# *Copyrights*

## 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 Patents and Licensing

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

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.

## Disclaimer of Warranty

**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.

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.

## Confidential Information

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

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.

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

## Sample Code Copyright

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.

## IBM Trademarks

For a list of IBM trademarks, visit this Web site www.ibm.com/legal/copytrade.html. This contains a current listing of United States trademarks owned by IBM. Please note that laws concerning use and marking of trademarks or product names vary by country. Always consult a local attorney for additional guidance. Those trademarks followed by ® are registered trademarks of IBM in the United States; all others are trademarks or common law marks of IBM in the United States.

Not all common law marks used by IBM are listed on this page. Because of the large number of products marketed by IBM, IBM's practice is to list only the most important of its common law marks. Failure of a mark to appear on this page does not mean that IBM does not use the mark nor does it mean that the product is not actively marketed or is not significant within its relevant market.

## Third-party Trademarks

Adobe, the Adobe logo, Acrobat, the Acrobat logo, FrameMaker, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions.

AIX and Informix are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

HP and HP-UX are registered trademarks of Hewlett-Packard Corporation.

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

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

# *Table of Contents*

## UML to Applications 829



## UML for C Code Generation 1031



## UML and Java 1327



## UML and C# 1481



## UML and C++ 1513

## UML and Requirements 1713



## Testing UML Models 1759



## UML Modeling with System Architect 1793



## UML and Web Services 1809



## XML Schema Modeling with UML 1891



## Exploring UML Models 1899



## Customizing Tau 1979

## Common Reference 2369

# *Introduction*

Tau 4.2 is a tool for the analysis and development of service-oriented architectures as well as other advanced software systems. Included in Tau 4.2 are tools for modeling applications using the UML notation. For an introduction to UML see chapter Chapter 8, UML Language Guide.

To fully take advantage of Tau and be able to start working quickly, it may prove useful to start with one of the tutorials included in this installation, like the Java Tutorial and the UML Tutorial. Additional and updated tutorials are available at this webpage.

Tau 4.2 includes many capabilities for analysis and development of service-oriented architectures, but of specific interest for this application area are the following chapters:

- Chapter 65, Web Services Support, that describes the web service modeling support in Tau 4.2 ,
- Chapter 69, Modeling XML Schemas, that describes how to model the XML data used by the web services,
- Chapter 63, Using Tau with System Architect, that describes how to use Tau 4.2 together with System Architect to enable an integration of the web service development with an enterprise architecture analysis.
- Chapter 67, WSDL/XSD Importer Reference, that describes how to import existing service descriptions in WSDL or XSD and how to generate WSDL/XSD from UML models.

In addition, the following sections can provide useful information:

- Chapter 84, Useful Shortcut Keys will provide a listing of possible shortcuts, this chapter can provide you with information on how to work faster and more efficient once you are familiar with what Tau can achieve.

- Chapter 85, Setting Up the Tool Environment contains descriptions of how to set up Tau together with configuration management and requirement management tools.

# 4

# *Introduction to Tau 4.2*

**UML**

Tau contains a set of model-driven tools based on UML 2.1 which are backwards compatible with UML 1.x. There is support for the following diagram types:

- Use case diagram
- Sequence diagram
- State machine diagram
- Activity diagram
- Interaction overview diagram
- Class diagram
- Package diagram
- Component diagram
- Deployment diagram
- Composite structure diagram (formerly called Architecture diagram)
- Text diagrams (in UML syntax)

You can easily verify your implementations by simulating real-time behavior using the Model Verifier.

The UML tool set contains support for handling Java code. UML and Java being very similar languages it is possible to switch between the two in text diagrams and text symbols. The Java support and Eclipse integration in Tau are only supported for Windows operating systems.

There is support for full C or C++ code generation and code generation for target integrations for market leading operating systems and real-time operating systems.

To be able to start working quickly with UML, the topics listed below may prove useful to start with:

- Description of Workflow
  Provides you with a road map how to use the UML tools in different stages in your project.
- Working with Models
  Describes the basics behind model-based development. It provides you with instructions and introductory information.
- UML Language Guide
  This section is a guide to the UML language.
- Java Tutorial
  A basic tutorial that allows you to work with the supported diagrams and to learn how to verify your model.
- UML Quick reference guide
  Examples on common constructs in graphical and textual UML.

# Overview of Tau User Interface

The complete Tau user interface contains several different areas that can be switched on and off at the will of the user: the **Desktop**, the **Workspace** window and the Output window. In addition there is a status bar in the lower part of the frame and a menu and Toolbar area at the top of the interface frame. The windows are all possible to dock according to the user's preferences. It is also possible to drag-and-drop frequently used toolbars to a Shortcuts window.

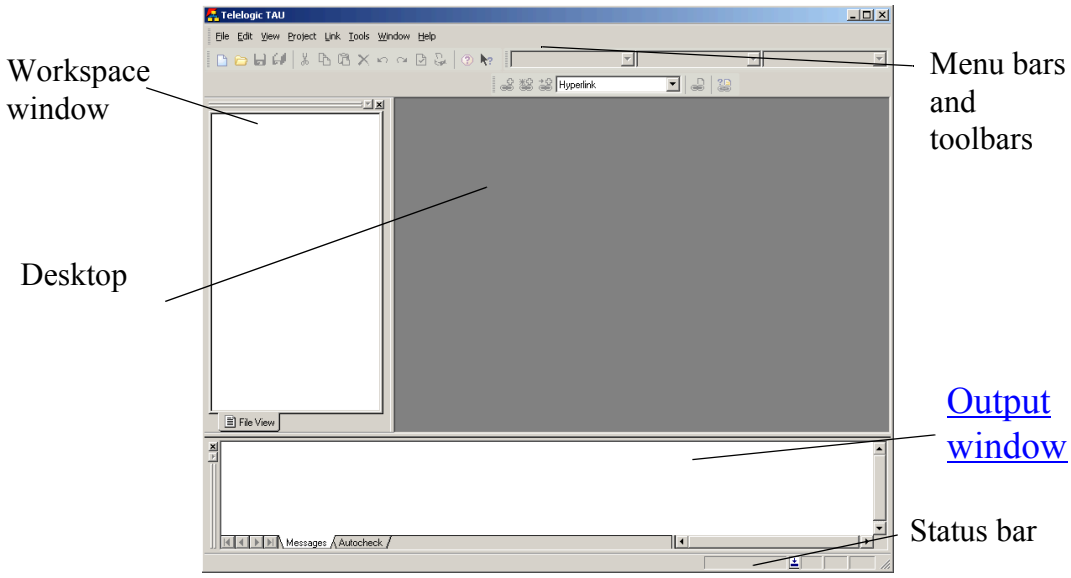



*Figure 1: The Tau Desktop.*

## Desktop

The Desktop, or editing area, is the area of your working documents. This is where the actual development takes place. Here you will see diagrams, documents, source files, etc. Once you have opened them for editing or viewing. Which editor or viewer that is displayed depends on the file types that are included in your project.

If you have more than one document opened on the desktop, you can move between with commands on the Window menu, or by pressing CTRL + TAB (forwards) or CTRL + SHIFT + TAB (backwards).

**Hint**

*If you want to have your editor expanded to full screen, select Full Screen in the View menu. To go back, press ESC or ALT + '1'.*

**See also**

"Working with windows" on page 23

## Workspace window

The Workspace window is a graphical tool that presents and manages the structure of the workspace information in a number of Views.

The Workspace window shows the information structure with expandable nodes. By collapsing and expanding these nodes, and by using different views, you can focus on different sets of the information in the workspace.

It is possible to move the Workspace window and to make it a floating palette. When not floating, it is docked at this left-most position. When docked, the window can only be resized horizontally. The vertical boundary is determined at the top by the toolbars and at the bottom by the Output window.

## Edit markers

### Gray bar

In the File View and Model View, elements that can not be edited are marked with a gray bar between the element symbol and the element name.

### Red bar

In the Model View, when you edit your model there will appear a red bar between the element symbol and the element name, indicating a change during the current work session.

### Asterisk

When a text file has been edited but not yet save an asterisk will appear in the window title bar after the file name.

## Views

In the Workspace window you have access to a number of views. They are each accessible via separate tabs. The views show different aspects of the model.

### File View

The File View shows your workspace with all elements that are represented as files. This could be project files, UML models, text files or any other files that you have stored in your projects. However, since diagrams, classes, etc. are not represented as files, they will not appear. Instead, the files in the File View contain the textual representations of the model.

You select the File View by clicking the **File View** tab in the **Workspace window**. Here you can open, edit and save all files. However, if you delete a file it is only deleted from the File View. The file is still available in the file system of your operating system.

To get a better overview of your files, you can create folders. You can drag and drop files between folders. You can display properties by selecting an item, right-clicking it, and clicking Properties in the shortcut menu.

### Model View

The Model View contains all the data that you are working with in an abstract structure. All UML elements are found here. Since the Model View shows all the elements that are not represented as files, this is the view to select when you add elements to the model and create diagrams.

The elements in this view are considered as graphical representations of the model. You may use diagram editors for the design process, but you may as easily fully design a system just working with the nodes in the Model view.

You select the Model View by clicking the **Model View** tab in the **Workspace** window.

The shortcut menu on the project nodes or the Model nodes will contain a sub-menu called **Model View Filters**. This menu will contain check boxes that defines how a set of predefined filters are applied to the Model View.

A Metamodel may include one Metaclass that has its base set to **Resource**. This model element will map to Resource elements in the loaded element at run-time and will thus only be visible in the Model View if the Show Files model view filter is turned on. It is possible to show files and resource elements in the Model View. Use the shortcut sub-menu **Model View Filters** and select **Show Files**.

A metamodel may include metaclasses that have their base set to object model classes that are subclasses of Diagram. These model elements will map to diagrams and will thus only be visible in the Model View if the **Show Diagrams** model view filter is turned on.

For convenience the metaclasses are classified as either structural entities or as detailed entities. This will affect how the contents of the Model View is affected by the **Show Details** filter.

A metamodel may include metaclasses that have their base set to object model classes that are subclasses of Implementation. These model elements will map to implementation-oriented elements and will thus only be visible in the Model View if the **Show Implementations** model view filter is turned on

When the **Sort Definitions** filter is active (indicated by check-mark in the menu item), model elements in a given Model View node are sorted in a lexicographical order. Diagram nodes will not be affected by the sort operation.

The command **Reconfigure Model View** (from the **View** menu) allows you to filter the information in the Model View based on a predefined metamodel. This affects the project which the selected element belongs to.

In some situations it is possible to cause a node in the Model View to "disappear" when you have selected a filter different from the **Standard View**. This happens because some operations in Tau (like drag-and-drop) rely on the basic metamodel that is used for storage of UML information, while others (like showing the elements in tree form in the Model View) rely on the currently selected metamodel. When you switch to **Standard View**, this model is the same as the basic metamodel. The **Diagram View** will present the information based on model elements that can own diagrams and push the diagram nodes to directly below their owning elements.

Typically this can happen in the following situations:

- drag-and-drop
- cut and paste
- creating diagrams from the Model Navigator Diagrams tab

To restore a node that have disappeared in this way you have two options:

- Use undo to get the node back at its original place.
- Switch to **Standard View**. In that view, all nodes will be visible. Any nodes that may have been drag-and-dropped to a position where they did not appear in the Model View will now show in their new place.

### Instances

Instances is a tab in the Workspace window, which is only accessible when you start the Model Verifier. It shows:

- The agents that are being executed, with inheritances and instantiations flattened.
- Live agent instances with their instance number and attributes.
- Kernel dependent objects; the ready-queue and the system environment agent.

You select the Instance view by clicking the **Instances** tab in the **Workspace** window.

## Files

Files are represented by icons in the File View in the Workspace window.

You may insert any file into your projects that you feel belongs there. The inserted files will show up as icons in the File View of the Workspace window. The Workspace window will start the associated program for viewing or editing if you double-click the file icon.

You may insert other UML files into your model folders. This enables you to share and move information between models.

## Shortcuts window

The Shortcuts window may contain toolbars, but it will only do so if you put them there. To put a Toolbar in the Shortcuts window, right-click the toolbar and click **To Shortcut** from the shortcut menu. You can reverse the process by right-clicking on the Shortcuts shown and clicking **As Toolbar** from the shortcut menu that appears.

The **View** menu **Shortcuts** command controls if the Shortcut window is shown.

**Note**

*Not all toolbars will be possible to submit to the shortcuts window.*

## Output window

The Output window consists of a number of different tabs that records and displays information for the corresponding tool. This information is typically error messages, warnings, result of actions, logging of events, etc. Each tab represents a different tool.

For some of the tabs, it is possible to navigate from the located element (in the subject column) to a presentation in a diagram.

The **View** menu **Output** command controls if the window is shown.

### General tabs

#### Messages

The Messages tab shows general information regarding the project loading and other executed actions.

No navigation is available from this tab to other parts of the tool.

#### Search result

This tab displays the result of a Find operation.

#### Presentations

This tab displays the result of a List presentations operation.

#### References

This tab displays the result of a List references operation.

#### Script

The Script tab shows the result of scripts, for example Tcl scripts.

### UML tool tabs

#### Check

You can initiate a complete check of the model to detect errors and warnings. This tab displays the result of the check. The errors remain in the list even after they are corrected. The list is changed the next time you invoke the check procedure.

#### Autocheck

There is an automatic check for errors in the scope you are currently working in. The scope is determined from the diagram that is opened on the desktop. This tab displays the errors that are found. When the error is corrected, it is removed from the list.

#### Build

The Build tab shows the result of the build process including semantic and syntactic checks and code generation. When building an application, all warnings and error messages are presented here. It is possible to navigate from error messages or warnings directly to the source in the appropriate editor.

**Navigate**

This tab contains a tabular model navigation tool, the Model Navigator which is used to navigate through any given model.

**Model Verifier**

This tab shows the textual trace when you are verifying the behavior of your model. This tab also logs the commands that you type in the Model Verifier Console.

## Working with windows

**Arrange windows**

**Tile all document windows:**

- On the **Window** menu, click **Tile Horizontally** or **Tile Vertically**.

**Overlap document windows:**

- On the **Window** menu, click **Cascade**.

**To change the position of document windows:**

**Note**

*The docking state can not be changed for a window with tabbed documents.*

1. Right-click the title bar of a document window.
2. In the menu select:
   - **Docked** to dock the window within the application window. There are also options for where it should be docked.
   - **Floating** to be able to move it outside the application window.
   - **MDI Child** to make the window float within the editing area. There are also options for maximizing, minimizing and restoring the window.

**To view the active document in full screen:**

- On the **View** menu, click **Full Screen**

  Or

- Press **ALT + 1**

**To view the active document in normal size:**

To display the active document in normal size again after you have viewed it in full screen, do one of the following:

- Move the cursor to the top of the screen. When the menu bar appears, on the **View** menu, click **Full Screen**.

  Or

- Press **ALT + 1**

### Show and hide windows

**Show or hide the workspace window:**

- On the **View** menu, click **Workspace**

  Or

- Press **ALT + 0**.

**Show or hide the output window:**

- On the **View** menu, click **Output**

  Or

- Press **ALT + 2**.

### Close windows

**To close a document window:**

- On the **Window** menu, click **Close**.

**To close all document windows:**

- On the **Window** menu, click **Close All**.

### Create a new window

**To create a new document window:**

- On the **Window** menu, click **New Window**.

### Tabbed documents

If the option “Tabbed documents” is enabled in general options page, documents will be opened as tabs in a single window.

A document can be detached from a tabbed window by right-clicking the tab and selecting the Detach menu item. It will then function as a normal MDI child and the docking state can be changed.

### Docking windows

There are three different modes for editor windows in the Tau framework. These are applied to each diagram window individually by right-clicking the diagram title bar.

**Note**

*The docking state can not be changed for a window with tabbed documents.*

**Docked**

A docked editor window will align into the Tau framework like the workspace window or the Output window. It will be possible to move the windows around to arrange a suitable view.

**Floating**

A floating window is on top of the Tau framework. It will turn into a docked window if it is moved towards the framework frame.

**MDI child**

An MDI child window is positioned into the desktop area. Adjusting can be done manually inside this area or with commands from the **Window** menu.

### Auto-hide docked window (Windows)

A window that has a pin symbol in the gripper bar can set to auto-hide mode. If the pin is pressed, the window will be hidden and a label representing the window will be displayed instead. By hovering with the mouse over the label the hidden window will be displayed. Window can be docked again by clicking on the pin symbol again.

### Expand/Contract docked window

When two docked windows share the same side of the main window, a window can be expanded to take the whole side in possession by clicking on the arrow symbol on the gripper bar of the window (if available). By doing this the other windows on the side are minimized. The window can be contracted by clicking on the arrow symbol again.

### Stored workspace windows

All windows opened during a session will be reopened when the workspace is loaded again. The information will be saved in a .ttx file with the same path and name as the workspace.

**See also**

"Organizing the view" on page 175

## Menu bar and toolbar

When you first start Tau, the toolbars appear just below the menu bar.

Depending on your workspace preferences, and the size of your screen, you can display as many toolbars as you want, or none at all. You can add a button with a command to a toolbar, change the size of the buttons, and move the toolbars to different locations to suit your needs.

### Menu Bar

The menu bar contains well-known menus such as File, Edit, Project, and so on. Depending on the task you are performing the number of menus differ.

Most menu commands have a shortcut assigned. A list of Useful Shortcut Keys is found in Common Reference.

You can add commands to the Tools menu allowing you to easy access to non-Telelogic tools. This is done from the Tools tab.

As an example, the following procedure demonstrates how to add the Windows Notepad accessory to the Tools menu.

**To add a command to the Tools menu:**

1. From the **Tools** menu select Customize, and then click the Tools tab.
2. Click the New (Insert) button.
3. Type the name of the tool, as you want it to appear on the Tools menu, and press ENTER.

   For example, if you want to add a command for the Windows Notepad accessory, you might type Notepad.
4. In the **Command** box, browse or type the path and name of the program, for example, C:\Windows\notepad.exe.
5. In the **Arguments** text box, browse or type any arguments to be passed to the program. Leave this field empty for the Notepad accessory.

**Note**

*You can use the drop-down arrow next to the Arguments text box to display a menu of arguments. Select an argument from the list to insert argument syntax into the Arguments text box.*

6. The **Initial Directory** box is used to specify the file directory where the executable file for the command is located. For the Notepad accessory this field is left empty.

When the command appears on the Tools menu, you may click it to run the program.

You can add arguments to be passed to the program by typing them in the Arguments text box, or set the initial directory for your program by typing it in the Initial Directory text box.

If the program you are adding to the Tools menu has a `.pif` file, the startup directory specified by the `.pif` file overrides the directory specified in the Initial Directory text box.

### Toolbar

The toolbar allows you to set up a palette of your most common used tools in order to have quick access to them. Once you have made any changes to the toolbar, these changes are saved and retrieved for your next work session.

The standard toolbar corresponds to the operations available in the menu bar. The standard toolbar can be toggled on and off from the View menu (Standard command) or from the toolbar area's shortcut menu, the other toolbars only from the shortcut menu.

**Note**

*Not all toolbars and commands can be modified. This feature belongs to the Tau framework and is not supported for toolbars related to editors.*

**To add a toolbar button:**

1. Make sure that the toolbar you want to change is displayed.
2. From the **Tools** menu select Customize, and then click the **Commands** tab.
3. Add a button by clicking the name of the category in the Categories box, and then dragging the button or item from the Buttons area to the displayed toolbar.

**To delete a toolbar button:**

1. Make sure that the toolbar you want to change is displayed.
2. From the **Tools** menu select Customize, and then click the **Commands** tab.
3. To delete a button, drag it off the toolbar.

When you delete a default button from a toolbar, the button is still available in the Customize dialog box. However, when you delete a toolbar button with a custom appearance, its appearance is permanently lost, although the command is still available (Customize dialog box, Commands tab).

**Hint**

*To save a toolbar button with a custom appearance for later use, create a toolbar for storing unused buttons, move the button to this storage toolbar, and then hide the storage toolbar.*

**Show or hide toolbars:**

1. From the **Tools** menu select Customize, and then click the **Toolbars** tab.
2. Select and clear toolbars to show or hide in the **Toolbars** list.
3. Click Close.

alternatively

1. Right-click anywhere in the toolbar area in the user interface.
2. Click the toolbar you want to show or hide. The menu closes automatically.

**To change the appearance of toolbar buttons:**

1. From the **Tools** menu select Customize, and then click the **Toolbars** tab.
2. Select the following options:
   - **Show Tooltips** to enable tooltips to be displayed when the cursor moves over a button or field in the toolbars.
   - **Large Buttons** to display larger sized buttons in the toolbars.
3. Click **Close**.

## Status bar

The status bar presents useful information about status of several different types of tasks, for example it lists errors and tooltips. Here will also be presented information about progress and current actions.

For text files the current line number and column position are shown in the right most corner of the Status bar.

### Line navigation

Navigation to a specific line in a text file is done by pressing CTRL + SHIFT + G. Enter the wanted line number in the dialog that opens.

### Progress bar

There is one progress bar displayed showing the overall progress when opening a workspace to the right of the status bar.

There can also be one displayed for the progress of the separate parts of the loading process displayed in the message field. In this case there will also be a message explaining the current action in progress.

# Options

Tool options affect Tau, not just the current project or workspace. There are different ways of changing these options:

In the Options dialog, there are different tabs for different options that you can change. The number of tabs differs depending on what type of project that is active. To see a description of an option in the Options dialog box, click the question mark in the dialog box title bar, and then click the option.

In the Advanced tab you may use a tree view for all options available in the Options dialog. The Advanced tab is activated from a check box in the General tab. To change the option values in the Advanced tab, select the value and click F2.

### Options file

The option settings can be saved in an options file, `.tot`. This file can later be edited.

If the options file is added to a project, you can double-click it in the File View to open it in an options editor. It is possible to save several options files in a project and select which one should be used. This is useful if you often switch options: Instead of changing the options directly, you just change the priority of the options files.

In the installation there will be a number of files with a `.tot` extension containing internal framework settings and options. These files should normally not be edited by the user. Editing a file with a `.tot` extension may cause loss of data and incorrect behavior of the tool set. The options controlled should be edited from the Tools menu Options dialog.

### Change options

**To change options:**

1. On the **Tools** menu, click **Options**.
2. In the **Options** dialog box, select and clear options in the different tabs. In the Advanced tab press F2 to access an option value.

3. Click **OK**.

### Work with options Files

**Save the current options in a new options file:**

1. On the **Project** menu, click **Options** and then **Save As**.
2. In the Save As dialog, select a name and location for the options file (.tot).
3. Click **Save**.
4. Click **Yes** when you are asked to include the options file in your active project.

   By including it in your project, you will be able to open it from the File View and edit it.

**To edit an options file:**

1. Make sure that an options file is included in your project, that is, visible in the File View.
2. Double-click the options file. It will be opened in the options editor.
3. In the options editor, expand the options tree until you are able to see your options.
4. Select an option.
5. On the selected option, click the Value field and press F2. This will make the field editable.
6. Enter a new value.
7. Close the file when you are finished. You will be prompted to save your changes.

**Select which option file to use:**

1. This is only possible when there are more than one options file included in your project, that is, visible in the File View.
2. On the **Project** menu, click **Options** and then **Files**.
3. In the Option Files dialog box, select an options file in the list and click the arrow buttons to move it up or down. The options file on top is the options file that will be used.
4. Click **OK**.

## Customizing

It is possible to customize the appearance of the user interface. In the **Customize** dialog box (**Tools** menu), there are options for customizing toolbars, the Tools menu, window layouts, and add-in modules. This is further described in Chapter 73, Customizing Telelogic Tau.

## Local setup (UNIX)

### Windows directory

In your home directory a new directory named **windows** will appear containing a set of files used to align the properties for Tau between Windows and UNIX. The information stored in these files is not to be edited by the user.

### Copy and Paste

Selection of text and using the middle mouse button for directly pasting it in another terminal window is supported only from the tabs in the Output window.

The dedicated buttons for Cut, Copy and Paste commands found on Solaris native terminals are not supported.

### File dialogs

It is possible to filter displayed files in file dialogs with for example “*.u2” to show only this file type.

## Generate support request

A tool for sending information to support can be opened by clicking **Generate Support Request** on the **Help** menu.

From the support tool it is possible to create screen dumps and video clips of Tau and send information directly to Telelogic support.

# Working with Workspaces

## Workspaces - overview

A workspace is a personal working area, where you can add your projects. It allows you to organize your projects in a logical way. You can define a number of different workspaces, but you can only work in one workspace at a time. There is no upper limit regarding the number of added projects.

You cannot share your workspace with other users, since the content of your workspace may not coincide with the needs of other users.

Projects are stored with path names relative to the workspace. This allows you to move a workspace and all its contents from one location to another without losing any information.

The information included in a workspace is stored in a text file with the `.ttw` extension.

## Create a new workspace

Tau always starts with an empty active window. If you have not recently used workspaces, you may create a new one.

1. On the **File** menu, click **New**.
2. Click the **Workspaces** tab.
3. Type a name for your workspace.
4. Choose the location for your workspace. A folder with the same name as the workspace is created by default.
5. Click **OK**.

## Open a workspace

Workspaces may be opened, closed, and saved, just as in any other standard application. Workspaces that you recently have worked with are available in a shortcut list. The list can contain up to eight different workspaces.

**Open a workspace:**

1. On the **File** menu, click **Open Workspace**.
2. In the Open dialog, select or browse to the file you want to open. Click **Open**.

**Open a recently opened workspace**

1. On the **File** menu, point to **Recent Workspaces**. A list of the eight most recent workspaces is displayed.
2. Select a workspace.

## Save and close a workspace

- Save a workspace: On the **File** menu, click **Save Workspace**.
- To close a workspace: On the **File** menu, click **Close Workspace**.

## Add a project to a workspace

There are two methods of adding a project to your workspace:

**From the File View:**

1. In the **File View**, right-click your workspace, and click **Insert project...** from the shortcut menu.
2. In the Open dialog box, select the project you want to add.
3. Click **Open**.

**From the Project menu:**

1. On the **Project** menu, click **Insert Project into Workspace...** The Open dialog box appears.
2. In the Open dialog box, select the project you want to add.
3. Click **Open**.

**See also**

“Create a new project with a new workspace” on page 36.

# Working with Projects

## Projects - overview

A project contains a number of references to source files that together with instructions on how to compile them produce a program or final binary files.

A project must be inserted in a workspace and you can work with many projects within the same workspace, allowing for diagrams and documents to be moved within and between projects. The same project can also be included in different users' workspaces. Projects that you add to a workspace can be located on other paths, on different drives, or directly in the root directory. This enables your team to work collectively with the same projects.

If there is no workspace open when you create a project, a directory for the workspace and a workspace file are also created. Alternatively, you may add a project to an existing, open workspace.

The files that are referenced in a project can be of any type and for convenience they can be organized in user-created folders.

A project is not individual and can therefore be shared between users. Some settings are global: if one user adds a file, it is added to the projects of other users as well. Some settings, such as which font to use, are not global.

The information included in a project is stored in a text file with the `.ttp` extension. Files included in a project are stored with relative paths in the project file.

### Active project

When you add a project to a workspace, it will become the active project. All commands on the Project menu will be applied to the active project. If you have more than one project in the open workspace, you must actively choose which project that should be active in the Active Project list that is available in the toolbar.

## Recommendation for Windows users

It is generally recommended to place projects, and all of the included files in a logical drive (such as P:, Q:, etc.). This enhances interchangeability of models and projects between different members of a team. For instance,

when using the hyperlink or traceability link mechanisms the links will store an absolute location of the project. Also, for using the DOORS integration or other third-party-tool integrations this practice is recommended, and will allow navigation from DOORS to models without exactly replicating the file structure. For information on how to achieve this, see the `subst` DOS command, or consult your system administrator for more information. It is recommended to always open the tool through this location to achieve a consistent logical model repository.

## Starting to work with projects

When using the tool for the first time, you can, for example, create a file first and later add this to a project. But the recommended workflow is to create a project in a workspace, and subsequently add files. By doing this, you are in a better position to control your project.

You can either create a new workspace or work with an existing one for each project. Files in a project may be stored locally, or on other locations. The objective is to offer you a working environment that optimally suits your, and your project's, needs.

As you create and modify a project, you can view the components included in the various views in the workspace window.

You have multiple choices to change and set your preferences, either on tool level or for your current project. It is for example possible to create custom toolbars, menu commands, and buttons.

**Create a new project with a new workspace**

1. On the **File** menu, click **New**.
2. Click the **Projects** tab.
3. Select the type of project you want to create.
4. Type a name for your project. Using the browse button, you can also change the location of the project. The option **Create new workspace** is selected by default.
5. Click **OK**.
6. If you want, you can change the project settings. Click the help button to receive more information about the settings.
7. Click **Next** and **Finish**.

### Create a new project in an existing workspace

1. Open the workspace you want to add the new project to.
2. On the **File** menu, click **New**.
3. Click the **Projects** tab.
4. Select the type of project you want to create.
5. Type a name for your project and click **Add to current workspace**.
6. Click **OK**.
7. If you want, you can change the project settings. Click the help button to receive more information about the settings.
8. Click **Next** and **Finish**.

### Insert an existing project in a workspace

A project is handled within the workspace it is located in. The workspace can be opened, closed, and saved, just as in any other standard application. A project file (.ttp) can be inserted in any workspace, not just the one it is created in.

### To insert a Project in an existing workspace:

1. Go to the File View tab in the Workspace window.
2. Right-click the workspace icon, on the **shortcut** menu select **Insert File**. The Open dialog box appears.
3. Find the desired project, and click **Open**.

**Note**

*To be able to view project files it is sometimes necessary to change the file filter in the Open dialog*

**See also**

"Menu bar and toolbar" on page 26.

"Add a project to a workspace" on page 34.

# Add files and folders to your project

### Add files to your project

You can add any type of files to your project, including project and workspace files. A workspace added to a project is treated like a normal text file, without any functionality. Using drag and drop you can move your files to and from folders.

There are two ways of adding files to your project.

**From the File View:**

1. In the **File View**, right-click the project or a folder, and click **Insert files...** from the shortcut menu.
2. In the Open dialog box, select the files you want to add
3. Click **Open**.

**From the Project menu**

1. On the **Project** menu, point to **Add to Project** and click **Files**.
2. In the Open dialog box, select the files you want to insert
3. Click **Open**.

**Note**

*When you add file from the project menu, you cannot decide the target folder. All files will be added at the root level of the project.*

### Open a recently accessed file

1. On the **File** menu, point to **Recent Files**. A list of the most recent files is displayed.
2. Select a file.

### Add folders to a project

You can add folders to a project to logically organize your files. These folders are only defined in the project files – they are not represented in your file system. The file path in the operation system will remain unchanged.

When adding folders you can define a list of file extensions. These indicate what type of files that will be included in the folder. The list will be used as a filter when you add files to a folder. Only the files with the listed extensions will be displayed in the add file dialog by default.

There are two ways of adding a folder to your project:

**From the File View:**

1. In the **File View**, right-click the project, and click **New Folder** from the shortcut menu.
2. In the New Folder dialog box, type a name in the **Folder name** box.
3. Type one or more optional extensions in the **Folder extension** box in the form of `*.<extension>`. If you type more than one extension, separate them with a semicolon (;).
4. Click **OK**.

**From the Project menu:**

1. On the **Project** menu, point to **Add To Project** and click **New Folder**.
2. In the New Folder dialog box, type a name in the **Folder name** box.
3. Type one or more optional extensions in the **Folder extension** box in the form of `*.<extension>`. If you type more than one extension, separate them with a semicolon (;).
4. Click **OK**.

## Activate a project

To enable any functionality for a project, it must be activated. Only one project in your workspace can be activated at a time. To activate a project do one of the following:

- In the **File View**, right-click the project. Click **Set as Active Project** from the shortcut menu.
- In the **Project** toolbar select the desired project in the **Active project** list.
- In the **Project** menu open the **Configurations** dialog and use the Set active button in the Configurations dialog.

**Note**

*If you only have one project in your workspace, this project will automatically be set as active.*

**See also**

"Project settings and configurations" on page 41

## File and folder properties

You can view, and in some cases edit, properties for selected workspace items. Properties will only be available when a single item is selected in File View.

Properties for files, projects, and workspaces are view-only. Properties for folders are fully editable.

**To view file and folder properties:**

- In the **File View**, right-click an item and select **Properties**. You can also press ALT + ENTER.

**Set or change folder properties:**

1. In the **File View**, right-click a folder and select **Properties**.
2. In the Properties Editor box, change the **Folder name** and **File extensions**.
3. Optional: press ENTER to apply and verify changes.
4. Unless you want to view or change preferences for other workspace items, close the dialog box by clicking the close button.

**To close the dialog box without saving any changes:**

- Press ESC.

## Create, open and close files

**To create a new file:**

1. On the **File** menu, click **New**.
2. Select the file type you want to create.
3. Specify File name and File location.
4. If you have an open project, you can decide if you want to include the file in the project.
5. Click **OK**.

**To open a file:**

1. On the **File** menu, click **Open**.
2. In the Open dialog box, select or browse to the file you want to open.
3. Click **Open**.

Or:

1. On the **File** menu, click **Recent Files**. A list of the 8 most recent files is displayed.
2. Select file.

**To close a file:**

- On the **File** menu, click **Close**.

### Move files

Moving around files (and also project and folder nodes) using drag and drop will not affect the current state of the loaded UML model entities, unless the file is moved between projects. If the file is moved between projects the contained model elements will be removed from the source project and added to the target project. The Undo queue will be disabled after this move operation.

## Project settings and configurations

Project settings, available in the Settings dialog box, include options for analysis, code generation, building, execution, and logging.

When you change options in the Settings dialog box, they will be saved in the currently active configuration. A project may contain several configurations. You can Activate a project to set its configuration to be active in the Configurations dialog box, or in the corresponding lists in the toolbar.

### Project settings

There are two ways of changing the settings for your project:

**From the File View:**

1. In the **File View**, right-click the project, and click **Settings** on the shortcut menu.
2. The Settings dialog box appears. Change the settings according to your needs.
3. Click **OK**.

**From the Project menu:**

1. In the Project toolbar select the desired project in the Active configuration list.
2. On the **Project** menu, click **Settings**.
3. The Settings dialog box appears. Change the settings according to your needs.
4. Click **OK**.

**Project Configurations**

**To add a new configuration:**

1. On the **Project** menu, click **Configurations**.
2. In the Configurations dialog box, click **Add**.
3. In the Add Configurations dialog box, type a name for the configuration, select an existing configuration to copy the settings from, and select which project the new configuration should be associated with.
4. Click **OK**.

**To remove a configuration:**

1. On the **Project** menu, click **Configurations**.
2. In the Configurations dialog box, select a configuration.
3. Click **Remove**.
4. Click **OK**.

# Discovery based storage

## Introduction

Discovery-based storage is necessary to support scenarios where the user does not wish to have one file (the project file) to have explicit knowledge of all files contained in a model.

## Discovery Path

A discovery path, is a property of a project, and contains a list of locations in which Tau will search for files. Searching is recursive.

This property is edited through a text field called “discovery location” on the folder “Discovery” property page.

A discovery location may be a URN reference or a relative path, or a full path to a u2 model.

- If a path is entered manually (without using the browse button) then the path will not be altered by Tau.
- If the path is entered using the browse button the path will be adjusted with respect to the Tau->Options->General->URN map. That is, if the file is located under a URN location, then a URN reference will be calculated and saved.
- If the path is entered using the browse button, and the path is not located under a URN location but is relative to the project, then this relative path is calculated and saved.

If a path is entered manually (without using the browse button) then the path will not be altered by Tau.

However, if the path is entered using the browse button the path will be adjusted with respect to the Tau->Options->General->URN map. That is, if the file is located under a URN location, then a URN reference will be calculated and saved.

If he path is entered using the browse button, and the path is not located under a URN location but is relative to the project, then this relative path is calculated and saved.

If the path is not in the same filesystem as the project then path is left as-is and saved.

A Shallow modifier may be added to a discovery location and has the same semantics as the `.u2shallow` file - see below.

Wild cards are not supported in names of discovery locations.

### Resource Match Rules

Resource Match Rules are regular expressions which specify which files should be covered by Discovery-based storage.

The set of Match Rules may be either inclusive or exclusive.

The rules are editable through the “discovery file filter” text field on the folder property page.

The rules are a semicolon separated list of file matching patterns.

If the list is empty (or non-existent) then everything is included

The entries are applied in the order they appear in the list - first match wins.

### Directory Match Rules

Directory Match Rules are regular expressions which specify which directories should be covered by Discovery-based storage.

The set of Match Rules may be either inclusive or exclusive.

The rules are editable through the “discovery directory filter” text field on the folder property page.

The rules are a semicolon separated list of directory matching patterns.

If the list is empty (or non-existent) then everything is included.

The entries are applied in the order they appear in the list - first match wins.

**Directives**

- Ignore Directives

  If a file named `.u2ignore` exists in a directory being searched, the directory and all files within it will be ignored.

  If a file named "`File.u2.u2ignore`" exists then this means that the file "`File.u2`" will be ignored.

  If a file named directory.u2ignore exists then this means that the directory "directory" will be ignored.

- Shallow Directives

  If a file named `.u2shallow` exists in a directory being searched, all subdirectories within it will be ignored.

  If a file named `directory.u2shallow` exists in a directory being searched, then all subdirectories within "`directory`" will be ignored.

- Discover Directive

  A file with the extension `.u2discover` may be used to provide an additional list of discovery locations, as well as a resource inclusion and exclusion list.

**Note**

*Files with the extension *.u2x are always ignored.*

- .u2discover file format

```
#This is a .u2discover file
DiscoveryPath:
#Do not recurse into subdirectories of the following
location
Loca*tion1(Shallow)
Loca*tion2
ResourceInclusion(inclusive):
*.foo
*.boo
DirectoryInclusion(exclusive):
#Skip CVS subdirectories
CVS
```

The Shallow modifier can be added to discovery locations. This is interpreted in the same way as the `.u2shallow` file extension (see above).

A `ResourceInclusion` section may be qualified by either “inclusive” or “exclusive” (default is “inclusive” if not present). Lines starting with '#' are ignored. All whitespace-only lines are ignored. Sections can come in any order. Sections must start in first column. All sections are optional.

A `DirectoryInclusion` section may be qualified by either “inclusive” or “exclusive” (default is “inclusive” if not present).

Case is not significant in keywords and directives.

### I18N

The `.u2discover` file is assumed to contain valid UNICODE characters in a UTF-8 stream. BOM or similar magic numbers are ignored.

### Interpretation Of Directives

Directory/Resource Inclusion rules are applied per directory/resource. The inclusion rules present in the project are added to the match rules first, then when each directory is visited list of match rules present in the `.u2discover` file (if present) is appended. When a directory/resource is applied to the match rules, the first match wins. The additional `.u2ignore`/`.u2shallow` files are used to override any of the previously named matches.

### Generic SCC/Synergy

The Generic SCC/Synergy integration works for files that are discovered in the same way as for normal files.

### Making Discovered Files Explicit In A Project

To make a discovered file explicit in a project, simply drag it from its discovery folder (in the File View) and drop it on the project node.

### Allowing An Explicit File To Be Discovered Instead

Delete the file reference from the File View (choose “remove elements” from the subsequent pop-up). And make sure that the directory, in which the file is contained, is in the `DirectoryPath` property. Save and reload the project.

### Filter syntax

The only file matching wildcard supported is “*”.

### Manual Rediscovery

It is possible to force a discovery after the project is loaded. The project context menu has an entry called “Discover Files”. This only adds new files, it does not remove files that no longer exist.

# Model and Diagrams

## Models

The model comprises all diagrams that describe your system. Different diagrams describe different aspects of the application. When modeling a system in UML, class diagrams describe the entities and the relationships between these entities.

Use case diagrams and use cases in form of sequence diagrams allows you to specify external interaction and an overview of a systems behavior.

Activity diagrams and Interaction overview diagrams can be used to describe parallel behavior in a model.

State machine diagrams describe the behavior of each active class and composite structure diagrams describe the external behavior of an entity and how the entity interacts with other entities.

The application is compiled from the model. Different diagram types show different views of the model. This means that entities that are available in the diagram exist in the model, but not necessarily the other way around.

### Model elements

Removing a symbol, a presentation element, from a diagram will normally not result in the deletion of the corresponding entity in the model since an entity might be represented in more than one diagram.

However, deleting an entity in the model results in the deletion of the equivalent symbols in the diagrams since it is the model that represents the application and the diagrams only presents different aspects of the model.

Deletion of the presentation element will also delete the model element when there is a one-to-one relationship between the model element and the presentation element. A one-to-one relationship exists when a model element can only have one editable presentation, for example this is the case with many state machine symbols.

The model elements are shown in the Model View of the Workspace window.

### Importing a DOORS Analyst model

A model created in DOORS Analyst can be imported into a UML package in a Tau project by following these steps:

1. Select a UML package in the **Model View**.
2. Right-click and choose the command "Import Analyst Model..." from the shortcut menu.

   If DOORS is not already running, it will be started now. After providing your login information, you should be connected to DOORS and ready to import a DOORS Analyst model.
3. In the **Module selection** dialog, select the formal module which contains the DOORS Analyst model you wish to import, and click OK.

The DOORS Analyst model will now be available in a new package. For UML elements to be correctly imported from a DOORS Analyst module, **Edit in Analyst** must have been performed on the module, and the model in the DOORS Analyst editor must be saved prior to the import.

**See also**

Chapter 6, Working with Models

Chapter 8, UML Language Guide

"Views" on page 18

"Files" on page 21

## Diagrams

A diagram is a representation of the model you are working with in UML. Depending on the type of diagram, you will be able to define different properties and actions.

Diagrams in general represent different views of a single model. There are a number of different types of diagrams. Their names are derived from UML concepts. Supported diagram types are:

- Activity Diagram
- Class diagram
- Component diagram
- Composite structure diagram
- Deployment diagram
- Interaction overview diagram
- Package diagram
- Sequence diagram
- State machine diagram
- Text diagram
- Use case diagram

### Using the diagrams

As there are several different types of diagrams available to describe the model, this section gives some hints how they can be used.

The order to perform activities when building a UML model is optional. A possible workflow is displayed in Figure 2 on page 50.

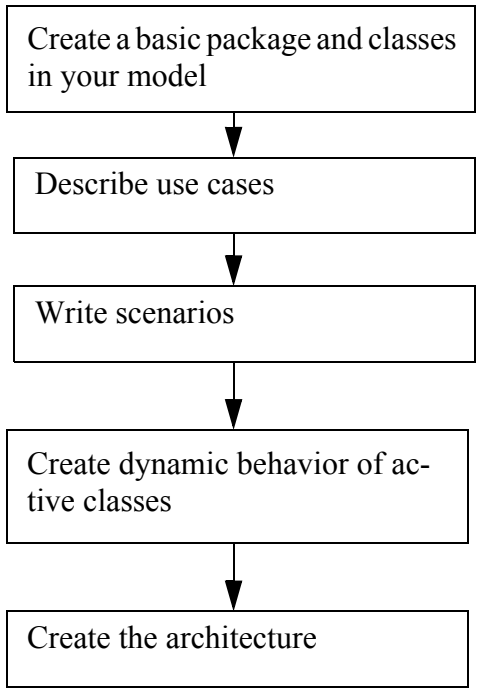



*Figure 2: Diagram creation workflow*

### Create a basic package and classes in your model

When you create a new project a package is automatically inserted, unless you disable this feature during project creation. It is possible to create new elements directly in the Model View by right-clicking on a package and from the shortcut menu select New and then choose the desired element from the submenu. To add a class diagram you right-click the package in the Model View and from the shortcut menu select **New** and then **Class diagram**.

### Create use cases

A use case diagram can exist directly in a package, in a class or be grouped in a collaboration. A use case can be inserted directly in a package, in a class or in a collaboration.

### Write scenarios

Scenarios in form of sequence diagrams is very easy-to-understand way of illustrating use cases. Syntax is simple and intuitive to understand, sequence diagrams is also a good basis for a dynamic behavior design.

### Create dynamic behavior of the classes

The next step is to define the behavior of classes that you have set to be active. This is done using the state machine diagrams. For each Active class, add a state machine diagram to the model and when opened, build the internal state machine that defines the behavior of the class using the symbols available in the active toolbar.

### Create architecture

The next step is to define how objects communicate. Instantiation of objects (parts) and communication between parts can be described with composite structure diagrams. Composite structure diagrams describe the internal structure of classes, attributes of classes and instantiations of classes.

### The next step

To continue working, create a test project and get to know how Tau lets you work with different diagram types, elements and symbols.

See also

"UML Language Guide" on page 199

"Working with Diagrams" on page 169

# Description of Workflow

The intention of this workflow description is to provide the users with a generic and simple road map on how the tool can assist in different project activities. In a project this description needs to be tailored for the specific project organization and application domain.

It may be simplified, but there might also be a need for a more complex process, for example by applying an iterative design approach or taking product maintenance into account.

**Workflow**

A series of activities performed by people involved in a project in order to obtain a result, normally a working product, according to a process.

Workflows appear as a result of tailoring a generic development process to the needs of a specific project. Workflows are thus strongly dependent on both the generic development process as well as the tailoring that take place in the project start-up. In this chapter the activities are in focus rather than the workflow as such.

Normally the roles for the people in a project are specialized to focus on one or a few specific activities, for example architect, designer etc.

This workflow description is divided into:

- Requirements analysis activities
- System analysis activities
- System design activities
- Detailed design activities
- Implementation activities

When moving from one phase to another, it is recommended to make a baseline, to freeze the model for that phase and continue the work in a new model in a new project, copying all information from the previous model that is useful in the new phase.

It is also worth noting that the division of work into clearly separated development phases should only be a means to organize the work and information into practically manageable pieces. The number of phases and the distinction between them may also vary in different projects and organizations. Another factor that reduces the difference between the phases is the capabilities of advanced modeling languages, such as UML, and the availability of advanced tool support.

With the strong simulation capabilities of UML models, a requirements model may actually be executable, thus effectively reducing the gap between requirements and implementation.

| Phase | Mostly used diagrams | New project wizard |
|---|---|---|
| Requirements analysis activities | Use case diagram<br>Sequence diagram<br>Interaction overview diagram<br>Activity Diagram<br>Class diagram | UML for Modeling<br>or<br>UML for Model Verification |
| System analysis activities | Use case diagram<br>Sequence diagram<br>Interaction overview diagram<br>Activity Diagram<br>Package diagram<br>Class diagram<br>Composite structure diagram<br>State machine diagram | UML for Model Verification |
| System design activities | Use case diagram<br>Sequence diagram<br>Interaction overview diagram<br>Activity Diagram<br>Package diagram<br>Component diagram<br>Class diagram<br>Composite structure diagram<br>State machine diagram | UML for Model Verification |

| Phase | Mostly used diagrams | New project wizard |
|---|---|---|
| Detailed design activities | Package diagram<br>Component diagram<br>Class diagram<br>Composite structure diagram<br>State machine diagram<br>Text diagram | UML for Model Verification |
| Implementation activities | Deployment diagram | UML for C Code Generation<br>UML for C++ Code Generation |
| System test activities | Sequence diagram<br>Composite structure diagram | UML for Model Verification<br>UML for C Code Generation |

See also

"Projects tab" on page 2450 in Chapter 91, *Dialog Help*.

# Requirements analysis activities

## 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.

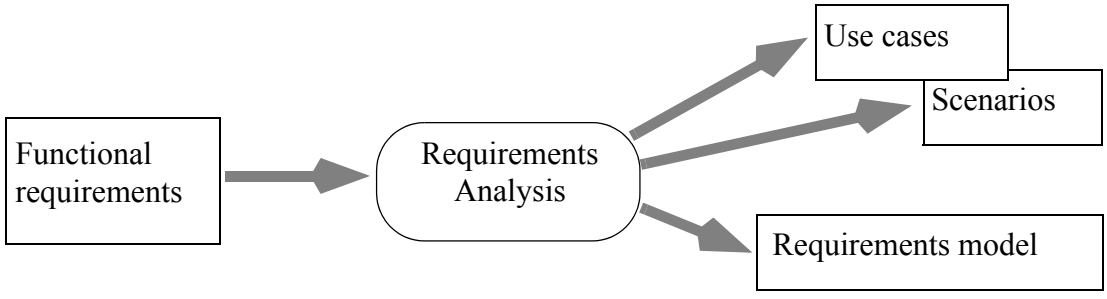



*Figure 3: Overview of the requirements analysis activity*

Functional requirements from a requirements' specification is normally the main input for the Requirements Analysis together with the knowledge and experience of domain experts.

### Identify use cases

Identify a set of use cases that are considered to be sufficient to realize the functional requirements of the application within the application domain.

### Create a requirements model

The purpose of the requirements model is to identify and document all the concepts found during the requirements analysis and to relate these concepts to each other.

- Identify sets of classes that are sufficient to realize the functional requirements within the application domain.
- Group the identified sets of classes into appropriate separated domains if needed and describe the relations between the classes.
- Implement an executable requirements model for each identified domain (optional).

### Create scenarios

Establish a set of Interactions (Sequence diagrams, Interaction overview diagrams) for the use cases that demonstrate how the class instances identified interact to realize the requirements covered by the use cases. A scenario can be regarded as an implementation of a use case.

### Verification and validation activities

- Verify the requirements by executing the requirement model. (Optional)
- Store the test results as Interactions (Sequence diagrams) (Optional).

### Use cases

From the functional requirements, identify a set of important Use Cases and the Actors involved in the use cases. Describe this in Use Case diagrams.

### Requirements model

A Requirements Model consists of Class diagrams with the main Classes (in a domain context). Include:

- Important active entities, including Actors from the Use Cases
- Important messages as Signals
- Important data (Classes)

Optional: add a simple behavior description in a state-oriented state machine for each active entity so that the requirements model is executable. Simulate using the Model Verifier.

### Scenarios

Detail each use case with a simple Interaction, described in a Sequence diagram.

**See also**

"Use Case Modeling" on page 216

"Scenario Modeling" on page 223

"Class Modeling" on page 261

## System analysis activities

### 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.

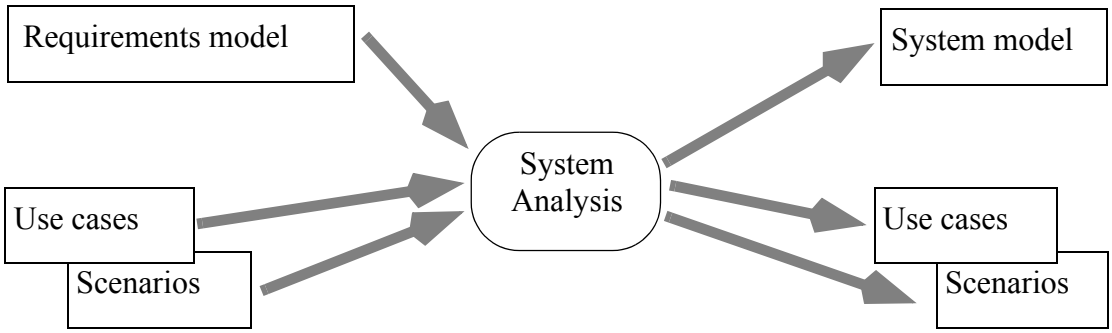



*Figure 4: Overview of the system analysis activity*

### Create a system model

The intention with the system model is that it is a means to describe a simple logical architecture, that is the main objects that need to be implemented in the completed system.

- Determine the classes from the requirements models to be reused in the system model.
- Establish a set of new system analysis classes sufficient to realize the requirements concerned with application architecture and high-level technical architecture within the platform.
- Combine the new system analysis classes and the re-used classes from different requirement models to form one or more system models.
- Implement one or more executable system models for each high-level architectural part (optional).

### Refine use cases and scenarios

The purpose of the use cases and scenarios of the system model is to document how the logical architecture is capable of implementing the requirements.

- Determine requirement analysis use cases that need to be modified as a function of the inclusion of high-level technical architecture and application architecture considerations.
- Establish a set of new use cases that are needed in addition to the reused use cases due to the inclusion of the system analysis requirements within the platform.

- Establish a set of Interactions (Sequence diagrams, Interaction overview diagrams, Activity diagrams) for each use case that demonstrates how the identified classes interact to realize the requirements covered by the use case.

**Verification and validation activities**

- Verify the use cases and scenarios by executing the system model.
- Store the test results as Interactions (Sequence diagrams) (optional).
- Validate the system model.

**System model**

A System Model typically consists of:

- Package diagrams for Package structure and Package Dependency visualizing the organization of the model.
- Class diagrams for the most important active entities (Active Classes) in a system context, that is to say that the system architecture should now be considered.
- Class diagrams related with messaging, Interface and Signal definitions for the most important messages as well as important message data (passive Classes and Datatypes).
- Class diagrams for passive data modeling, that is the most important passive Classes including important Operations and Attributes.
- Composite structure diagrams visualizing a simple architecture and communication structure (Parts and Connectors) for modeled Active Classes.
- State machine diagrams for Active Classes, typically not too detailed, of the state-centric kind providing an overview of each state machine without defining all details on the transitions.
  It is a good idea to make these State machines complete, that is to say executable so that the model can be simulated using the Model Verifier.

It is not necessary to strictly separate class diagrams into clear categories as mentioned above. Often it is more important that the view in each diagram makes sense, for example by showing relations between active and passive classes. To visualize one definition in several contexts is also a frequently used UML technique and the strong, model-driven approach in the UML tool set makes it simple to maintain such models.

### Use cases and scenarios

Update the Use Cases according to the system context of the System Model rather that the domain context of the Requirements Model. Add use cases so that all functional requirements are covered by Use Cases.

Scenarios (sequence diagrams) should be attached to each Use Case, providing a description of how all the active entities (Active Classes) interact in order to achieve the goal of the Use Case. If the architecture is complex, it is a good idea to have scenarios both on system level (showing the system and external Actor communication) as well as detailed level (showing how active entities within the system interact with each other and external Actor).

**See also**

"Scenario Modeling" on page 223

"Package Modeling" on page 255

"Class Modeling" on page 261

"Behavior Modeling" on page 330

## System design activities

### Overview

The major task of the system design activity is to define the precise architecture of the system.

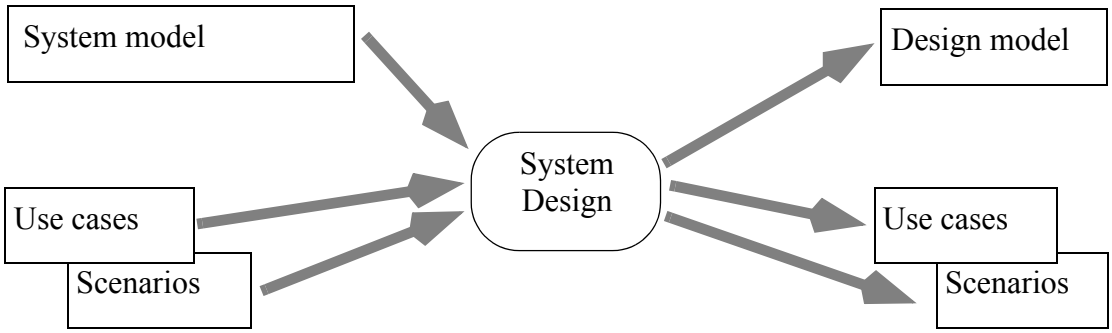



*Figure 5: Overview of the system design activity*

### Create a design model

The intention with the design model during system design is to describe a precise architecture.

- Determine the classes from the system models to be reused in the design model.
- Establish a set of new system design classes that are sufficient to realize the detailed architectural and technical requirements within the technical domain.
- Combine the new system design classes and the re-used classes from different system models to form one or more design models.
- Implement one or more executable design models.
- If necessary, establish one or more target models in order to optimize performance.

### Refine use cases and scenarios

The purpose of the use cases and scenarios of the design model is to specify how the precise architecture is capable of implementing the requirements in a more detailed way compared to the system model.

- Determine system analysis use cases that need to be modified as a function of the inclusion of detailed technical requirements, technical architecture and application architecture considerations.
- Establish a set of new use cases that are needed in addition to the reused use cases due to the inclusion of the system design requirements within the platform.
- For each use case, establish a set of Interactions (Sequence diagrams, Interaction overview diagrams, Activity diagrams) that demonstrate how the identified classes interact to realize the requirements covered by the use case.

### Verification and validation activities

- Verify the use cases and scenarios by executing the design model.
- Store the test results as Interactions (Sequence diagrams) (optional).
- Validate the design model.

### Design model

A design model typically consists of:

- Package diagrams for Package structure and Package Dependency visualizing the organization of the model.
- Component diagrams for active entities (Active Classes). Now all active entities should be modeled.
- Class diagrams related with messaging, that is all Interfaces with the most important Signal definitions as well as important message data (passive Classes and Datatypes).
- Class diagrams for passive data modeling, that is the most important passive Classes including important Operations and Attributes.
- Composite Structure diagrams visualizing a detailed application architecture and communication structure (Parts and Connectors) for modeled Active Classes.
- State machine diagrams for Active Classes, typically not too detailed, of the state-centric kind providing an overview of each state machine without defining all details on the transitions. It is a good idea to make these State machines complete, that is to say executable so that the model can be simulated using the Model Verifier.

### Use cases and scenarios

Update the Use Cases according to the completed application architecture.

Update the Scenarios (Sequence diagrams) according to the completed application architecture. If the architecture is complex, it is a good idea to have scenarios both on system level (showing the system and external Actor communication) as well as on a detailed level (showing how active entities (Active Classes) within the system interact with each other and external Actors).

## See also

"Package Modeling" on page 255

"Class Modeling" on page 261

"Architecture Modeling" on page 299

"Behavior Modeling" on page 330

# Detailed design activities

## Overview

A detailed design activity is often a part of the System Design.

However, there is a point in separating these activities, since many functional requirements can be verified before the detailed design has started, by using the Model Verifier on the “high-level” System Design.

The purpose of the detailed design is to complete the design model by adding detailed behavior and detailed data modeling.

## Refining the design model

Now, the design model should be fully specified:

- Complete passive data.
- Complete all operations of passive classes.
- Complete interfaces, messages, message data.
- Complete behavior design, now with transition-centric, detailed transitions.

The detailed scenarios completed in the system design is a very effective source of information for completing the behavior design.

## Verification and Validation activities

- Validate the detailed design model
- Make sure that all functional requirements (use cases) now have a detailed design.

## Design model

A detailed Design Model typically consists of:

- Package diagrams for Package structure and Package Dependency visualizing the organization of the model.
- Component diagrams for active entities (Active Classes). Now all active entities should be modeled.
- Class diagrams related with messaging, that is all Interfaces and Signal definitions including all message data (passive Classes and Datatypes).

- Class diagrams for all passive data, that is passive Classes including all Operations and Attributes.
- Composite Structure diagrams visualizing a detailed application architecture and communication structure (Parts and Connectors) for modeled Active Classes.
- State machine diagrams for Active Classes, detailed with complete transition behavior. Most often this means that a transition-centric presentation of the State machines is preferred.
- Operation Body or State machine diagrams for all Operations
- In some cases Text diagrams can be used for Operations.

See also

"Class Modeling" on page 261

"Behavior Modeling" on page 330

## Implementation activities

### Overview

Although this activity is placed after the Detailed Design activity in this workflow description, it is recommended to start the activity as early as possible, in order to get implementation feedback that may affect both the selected application architecture as well as the detailed design.

An incremental approach can assist in getting early implementation feedback and is generally recommended.

The Implementation Activity in a project is highly automated. A few steps can be identified, though:

### Planning the code architecture

- Use deployment diagrams to illustrate relations to hardware or fundamental software layers.
- Import external C/C++ APIs into the Model.
- Decide the file structure of the generated code (makefile).
- Select a target environment and application area (example: WIn32, threaded application).

- Tailor Code Generator settings.

**Implementation**

- Implement the signalling interface to the environment (“Environment Functions”).
- Tailor the target integration.

**Verification and validation activities**

- Debug the signalling interface and all other hand-written code.
- Test.

**See also**

Chapter 26, Building and Code Generation Overview and Examples

Chapter 27, Building Applications Reference

Chapter 35, C Code Generator Reference

Chapter 15, C/C++ Import

Chapter 52, C++ Application Generator Reference

## System test activities

### Overview

The purpose of the System Test activity is to use the UML Testing Profile to test if the requirements are fulfilled by the UML system created in previous activities and that the system behaves as expected.

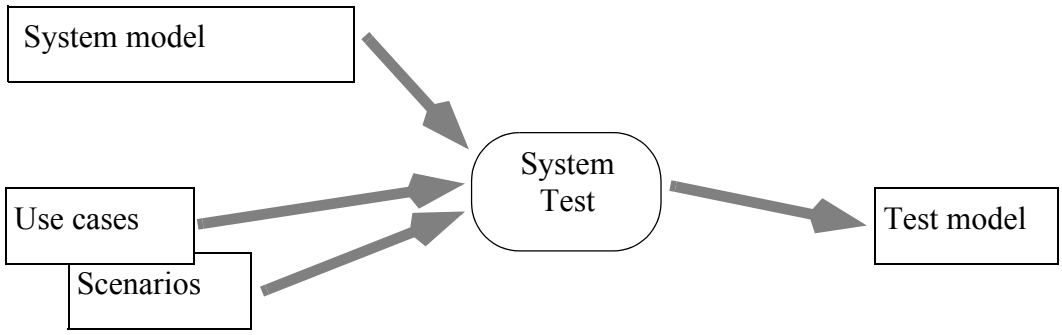



*Figure 6: System test activity*

### Create test model

A test model typically consists of

- A Test Context class containing
  - composite structure diagram with parts describing the SUT (System Under Test) and an instantiation of the test component
  - a set of test cases
  - the test component
- Dependencies to the UML model produced in previous activities

### Creating test cases

If test results (Interactions described in Sequence diagrams) from previous activities are saved, these interactions are candidates for test cases. Also the scenarios from previous activities are candidates. These interactions should be translated to Test Cases and (most likely) refined to be able to be executed.

### Test model

A Test Context class is a «Test Context» stereotyped class. The test Component is «Test Component» stereotyped class. The parts described by the Composite Structure diagram are an instance of the Test Component and a «SUT» stereotyped part. The SUT is typically an instance of the top level class (the system) produced in the Detailed Design activity. Signals and datatypes are typically reused (imported or accessed) from the UML system created earlier.

### Test cases

Test cases are «Test Case» stereotyped operations on the Test Context class. They are typically implemented in Sequence diagrams and specify how the SUT should communicate with its environment.

### Testing

Testing is done by building the test context class using the C Code Generator and running the tests in batch. Failing test cases are debugged using the Model Verifier.

**See also**

"Creating a test model" on page 1765 in Chapter 61, *UML Testing Profile*

# How to Use Help

This help file includes basic and advanced topics covering the supported functionality.

Additional product documentation is available in the section "Additional Resources" on page 2471. There you can find tutorials, language descriptions, the installation guide and links to external sites, for instance the Tau Support site.

Additional documentation in Adobe PDF format is also available on the installation CD.

## Navigate in the help file

The help file contains functionality that helps you to easier find the information that you are looking for:



### Search

**To perform a full-text search:**

1. In the help viewer, click the **Search** tab.
2. Type your search string in the **Type in the word(s) to search for** field. You may use regular expressions, operators, and nested expressions when searching.
3. Optionally, you may check some of the following options: **Search previous result**, **Match similar words** and **Search titles only**.
4. Click **List Topics**.
5. To open a topic, double-click the topic in the Select topic list or select a topic and click **Display**.

**Example 1:** ———————————————————————————————

To search for words beginning with “link”, type the following in the search field:

```
link*
```

———————————————————————————————————————

### Search highlighting

The words that you are searching for are highlighted on all pages where they are found. If you want to, you can turn off this functionality.

**Turn off search highlighting:**

1. In the help viewer, click the **Options** button and then click **Search Highlight Off**.
2. If you have already performed a search, click the **Display** button in the help viewer and the search highlighting disappear.

The search highlighting functionality is now turned off until you enable it again.

**Turn on search highlighting:**

1. In the help viewer, click the **Options** button and then click **Search Highlight On**.
2. If you have already performed a search, click the **Display** button in the help viewer and the search highlighting re-appear.

The search highlighting functionality is now turned on until you disable it again.

### Index

To see the list of index entries, select the Index tab. To find the entry you are looking for, type the first letters of the word or scroll the list. To view the entry, double-click the entry or select the entry and click **Display**.

### Locate search or index results

When you are using the search or the index functionality, the topic you are looking for will be displayed in the right-hand window. To locate where this topic is listed in the table of contents, click the **Locate** button. This allows you to easily find related topics or to learn where this topic is located the next time you are looking for it.

### Bookmark topics in the help file

If you know that there are topics that you will refer to often or that there are topics that you consider important for your work, you can bookmark them as you would do in a regular web browser.

**Bookmark a topic:**

1. Find your topic using the Contents, Index or Search tabs.
2. Click the **Favorites** tab. The name of the topic is listed in the **Current topic** field.
3. Click **Add**. The topic is now displayed in the topics list.

### Print help topics

You can print a single topic or you can select to print several topics within the same chapter.

**Print an active topic:**

- Right-click the displayed topic in the right-hand window and click **Print**. The print dialog opens.

**Print a single topic from the table of contents**

1. Right-click the topic window in the table of contents and click **Print**. The **Print Topics** dialog opens.
2. Click **Print the selected topic** and click OK. The Print dialog opens.

**Print multiple topics:**

1. Right-click a book icon in the table of contents and click **Print**. The **Print Topics** dialog opens.
2. Click **Print the selected heading and all sub-topics** and click **OK**. The print dialog opens.

# Search syntax in help

The help viewer supports full text search, and you can search for any combination of letters (a-z) and numbers (0-9). Words like “the”, “a”, “and”, “but”, are reserved and cannot be searched for. In addition, you cannot search for punctuation marks such as colon (:), semicolon (;), hyphen (-) and period (.).

You can group search elements by using quotes and parenthesis.

### Match similar words

The **Search** tab in the help viewer includes a **Match similar words** option. If you select this, you will be able to find all occurrences of a word, including common suffixes. For example, if you search for “run”, the words “run”, “running”, and “runner” will be found, but not “runtime”.

### Regular expressions

The following regular expressions may be used when searching the help:

- * for matching 0 or more characters.
- ? for matching 1 characters.
- A string within quotes (“ab cd”) for matching the string literally.

| Search for this: | Type this in the search field |
|---|---|
| Topics containing “analyze”, “analysis”, “analyses”, “analyzed”, and “analyzing” | analy* |
| Topics containing “analyzer” and “analyzed”, but not “analyze” or “analyzers” | analyze? |
| Topics containing the literal phrase “analyze and generate” | “analyze and generate” |

### Operators

You may use the following operators to refine a search in the help: AND, OR, NOT, and NEAR. The search string is evaluated from left to right. See table below for examples:

| Search for this: | Type this in the search field |
|---|---|
| Topics containing both “work-space” and “file” | workspace AND file<br>**or**<br>workspace & file<br>**or**<br>workspace file |
| Topics containing either “work-space” or “file” | workspace OR file<br>**or**<br>workspace \| file |
| Topics containing “workspace” but not “file” | workspace NOT file<br>**or**<br>workspace \| file |
| Topics containing “workspace” and “file” close together, that is “work-space” within 8 words of “file” | workspace NEAR file |
| Topics containing “workspace” but not “file”, or topics containing “workspace” but not “directory” | workspace NOT file OR directory |

### Nested expressions

By using parentheses, you may nest expressions to perform a complex search in the help. An expression within parentheses will be evaluated first, before the rest of the search expression. Expressions may not be nested more than 5 levels.

| **Search for this:** | **Type this in the search field** |
|---|---|
| Topics containing “workspace” without either of “file” or “directory” | workspace NOT (file OR directory) |
| Topics containing “workspace” with and “file” and “project” close together; or topics containing “workspace” with “directory” and “project” close together | workspace AND ((file OR directory) NEAR project) |

# *UML Modeling*

The chapters that are listed under UML Modeling describe functionality that is exclusive to UML projects.

# 6

# *Working with Models*

This chapter is intended to give an introduction to model-based development. It contains the background to how model bindings are maintained. It explains the syntax color scheme for text information.

**See also**

"Description of Workflow" on page 51

"Working with Diagrams" on page 169

"UML Language Guide" on page 199

# Models and Model Elements

### Model-based development

The model-based nature of the UML tool set offers strong mechanisms to aid you in creating and maintaining complex models.

There are two different ways of working:

- **diagram-centric**, where you create your model as you are creating and editing the diagrams of the model.
- **model-centric**, where you create your model in the Model View browser and afterwards define your diagram views.

It is of course also possible to combine these two paradigms.

### Diagram-centric workflow

The diagram-centric workflow is well known for users experienced with graphical languages. An example how this is carried out can look like the following:

- Create a diagram.
- Create the entities in that diagram.
- For the defined entities, create new diagrams that describe these entities in further detail.

A benefit with this approach is that you have a graphical context when you create new entities, which makes it easier to get it right.

### Model-centric workflow

The model centric workflow is not dependent on the graphical presentations that may or may not exist for the definitions in a model. Example of workflow:

- Define model elements in the model browser.
- New model elements are placed within this model structure.
- Diagrams are created whenever needed or wanted to visualize relevant parts of the model.

- Visualizing entities in the diagrams is easily done by dragging an element from the Model View browser to the diagram.
- Model elements may be visualized several times, and in different diagram views.

One consequence of model-based development is that it is sometimes optional to describe entities within diagrams. If completeness of the diagram representation of the model is important, the tool can be configured to check for entities that are not represented graphically.

## Model element and Presentation element

### Model element

When creating a new definition by entering a new name on either a new object or an existing object, the tool will recognize that it does not exist in the model. This will create a new Model element. This model element will be visible in the Model view of the Workspace window.

### Presentation element

A symbol in a diagram is a Presentation element, which is based on a model element. There can in many cases be any number of presentation elements to a given model element.

### Element properties

Changing properties on a presentation element, like for example the name of an attribute in a class symbol, this change will also take place in other presentations of the class. The change has been done on the model element, and all presentations that show the affected property will be updated.

If you add a new attribute to the class (either in the model browser or in one of the presented class symbols), this will not automatically appear in all presentations. The attribute is of course available in the model so that it can be conveniently added in the class symbols where it is wanted to visualize this property.

### Delete

If you delete an attribute in a class symbol, this will only remove the presentation of the attribute in that symbol.

### Delete from Model

When you delete the attribute in the Model View, this will delete the model element for the attribute and subsequently all presentation elements of the attribute will disappear (it is also possible to right-click on a presentation element in a diagram and use the shortcut menu command **Delete from Model**).

If you delete a class that is referenced in other places, for example as an attribute type in other classes, these references will become unbound when the class is deleted. This is immediately reflected in the diagrams (by a red wave).

## Model element

### Binding

When creating a new definition by entering a new name, the tool will recognize that it does not exist in the model. This is indicated by the text color, which is gray. If you want to use this name for a definition of a new model element, you have to **commit** the name (by simply pressing return).

When referencing an existing definition, the tool binds the typed name to the model element. This is indicated by the text color changing. If the name cannot be resolved (the name can be misspelled or the definition may not be visible according to the visibility rules), this is indicated by a red wave below the symbol.

### GUID

A UML entity is by default given a unique randomly generated identifier, called a Globally unique identifier, GUID. A GUID remains unchanged for the entire lifetime of an entity.

### Automatic naming of new elements

When adding new symbols that can define model elements, a default name is created for the symbol so that the name is unique in the current scope. You can just start typing (without selecting the text) to change the given name to your wanted name.

### Copying and moving model elements

A model element can be copied and pasted.

If you copy a symbol that references a model element and paste this symbol, this will just give a new presentation of the existing model element. Changing the name in one of these two symbols will also change the name in the other symbol: they are both presentations of the same element.

If you copy a model element in the browser, you can paste this in another place (here it is a copy of the model element itself). If you paste into the same scope you will have two conflicting definitions with the same name, something the checker will report. If you change the name of one of the model elements, the conflict will be resolved.

You can also copy model elements between projects. A structured way to reuse definitions between projects is to put them in packages, possibly defined in the same model (.u2) file.

Just as model elements can be copied, they can also be moved, typically by dragging from one scope to another or from one project to another.It is also possible to do a save of an element in separate file, see "Save in New File" on page 172 in Chapter 7, *Working with Diagrams*.

### Model in several files

It is possible to split the model in several files automatically by selecting a package or class in the browser and using the command **Auto create files** in the context sensitive menu. The model will now be split into several files with class as the finest granularity.

To enable this you go to the **Tools** menu and point to **Options**. Go to the **UML Basic Editing tab** and set the option **Default Create file mode** to **Package and Classes**.

**Auto create files** is active for elements that can be saved to a new file. If **Auto create files** is selected all classes and packages below the selected element are stored in separate files. The files are given the same name as the root element. If a file with this name already exists in the project then a number is added to the file name.

**Auto create files** is also available for project nodes (.ttp) and model nodes. **Auto create files** is not active for Libraries and Predefined packages.

# Text Highlighting

## Syntax highlighting

These are the main categories of syntax colored text tokens:

| Category Name | Default color palette |
|---|---|
| Names | Black, dark red and gray |
| Literals | Brown |
| Keywords | Blue (slightly violet) |
| Comments | Green |
| Errors | Red |
| Brackets, Braces, Parenthesis, etc. | Black |

In these categories there are subcategories that are based on the actual contents of the text or information added in the later passes (detailed below). Pointing on a token with the mouse gives a tool tip indicating semantic information about the token, such as the token type, or the type of object represented by the token. For instance: "'`IfAction`', class X, 2 references are currently bound to this object, CTRL + Click to navigate".

```
const Integer i = 3;
const Duration DefaultTimeout = 10;
signal ack(Boolean); /*  comment */
timer tim () = DefaultTimeout;
```

*Figure 7: Example of syntax text coloring*

## Syntax error markers

If the text contains a syntax error, the text contains markers at the position of the error. If pointing to the position of the marker, a tool-tip provides the syntax error message, for example:
'`Syntax Error, found token?, expected Name`'.

The markers are in the shape of upwards-pointing triangles of a height not exceeding half the height of the text, and with a center near the position of the text base line. The color of the marker depends on the severity of the error:

| Severity | Color |
|---|---|
| Fatal error | Orange |
| Error | Red |
| Warning | Yellow |
| Information | Light blue |

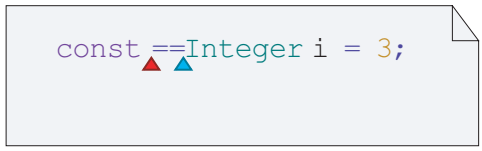



*Figure 8: Syntax error markers*

### Semantic highlighting

Names in text are subject to additional highlighting rules, extending the Syntax highlighting.

- Rules for name text coloring:

| Color Rule Description | Highlighting |
|---|---|
| A name represents a definition | Name is in black color |
| A name represents an identifier bound to a Type | Name is in dark green color |
| A name represents an identifier bound to an Event Class (Signal, Timer and Operation) | Name is in dark blue color |
| A name represents an identifier bound to an Attribute, Variable or Parameter | Name is in dark red color |
| A name cannot be determined to represent either (syntax error etc.) | Name is in dark gray color |

- Rules for name underlining:

| Underline Rule Description | Highlighting |
|---|---|
| Unbound identifier | A red saw-tooth-shaped underline |
| Obsolete identifier | A green saw-tooth-shaped underline |

Pointing the mouse at a highlighted name indicates the current status of the text in a tool tip. In particular this diagnoses the situation in some way if pointing to a name with one such underline. For example: “This reference is currently not bound, see AutoCheck log for details.”

### Object Location

In several places in a UML model, it is possible to locate a definition, for example by double-clicking an object in the Model View or in an output pane, or by choosing the Locate command for an object in an output pane. When this is performed, the correct diagram appears with the definition highlighted by yellow text background. Alternatively, if there are several possible presentations of the definition (or none), the Create Presentation is activated.

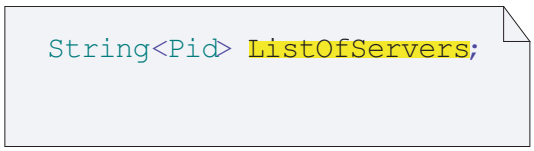



*Figure 9: Location marker*

### Name navigation

It is possible to navigate on names by holding CTRL and clicking on non-gray representations of names with no underline. The tool-tip also indicates this possibility.

## Properties

The complete set of properties of a model element is not always possible to edit through its presentation elements in diagrams. For this purpose there is a Properties Editor in which it is possible to view and edit the properties of a

model element. The Properties editor is opened from the shortcut menu (right-click on an element and point to **Properties...**), or press ALT + ENTER.

## Model checking

### Autocheck

As you are defining your model, it is continuously being checked by the tool. The Autocheck tab in the Output window is updated, whenever you modify the model, and presents a list of Errors and warnings.

Due to the design of the checking functionality, you will not have to wait for your large and complex model to be analyzed.

### Editor feedback

You will get instant feedback as you are editing your diagrams. If you type the wrong syntax, a syntax error marker will appear at the position where the error is made. A tool tip will also indicate the nature of the error (for example syntax error).

If you enter a name that is not known (for example misspelled) or a name that is not visible according to the visibility rules, this is indicated by a saw-tooth-shaped red line below the name.

The normal behavior is that when a name **binds** to an existing definition a Semantic highlighting will occur, resulting in a color change of the text.

### Syntax parse

When you edit text in diagram symbols, if the text is parsed correctly, the text is added to the model. After that, the text will be written back to the diagram symbol again based on the model.

This Text parsing is a consequence of the tight model-based approach. In some cases it will be written to one specific (of several possible) syntax alternative, thus not preserving your exact formatting.

### Restore model (F8)

It is possible to restore a model during text editing when the changes are not found correct by the syntax parser. This is done with the command F8, while the selection for the syntax error is still active in text edit. This will restore the text from the model information.

This command should be used with care. It will erase any text that is not bound to the model, like comments. When the model is first created and no correct model has been parsed, this command will erase everything.

### Name support

There are different ways you can get help from the tool when you want to reference a definition.

- Create Presentation lets you browse and navigate quickly through your complete model.
- **Name completion**

  After typing the first letters of the name, for example `ca`, pressing CTRL + SPACEBAR the tool will try to complete the name to an existing name, for example `card`. If there are multiple matches a Name completion scroll menu will open. Some special cases can be identified

  – Typing after a period ('.'). Name completion will list candidates matching the written characters that are local or inherited members (structural features or event classes) to the type of the left side expression.

  – Typing after a scope qualifier ('::'). Name completion will list candidates that are in the namespace of the left side expression.
- **Reference existing**

  When creating a new symbol (that can define or reference a model element) using the Diagram element creation toolbar and pressing the right mouse button in the diagram, the shortcut menu appears, with a submenu called Reference existing. This submenu contains a list of all visible definitions of the symbol kind, so that the wanted identifier can be chosen.

### Checking a complete model

Apart from the Autocheck, a model check can also be invoked manually. To check a complete model use **Check all** quick button (in toolbar Analyzing).

### Checking a part of a model

Select the part in the Model View to be checked. Use **Check selection** quick button (in toolbar Analyzing).

### Errors and warnings

If any problems are detected during a check of the model, this will be reported in the Check tab in the Output window. Each problem (warnings and errors) can normally be traced back to its origin, either in a diagram, or in the Model View browser. This is done by double-clicking the message or by selecting the message and right-click, then choose **Locate** in the shortcut menu.

# Models and Diagrams

## Diagrams

### Different views of the model

Diagrams are presentations of a model, typically focusing on one particular aspect and part of the model. One of the powers of UML is the capability to give different views of a model. This means that model elements are referenced in several places. Normally, this could be a problem when maintaining the model, but with the strong model-based tool support, all references are automatically kept up-to-date, that is if properties of a model element change, these changes will be reflected in all places where the element is referenced.

## Presentation element

### Symbols

Symbols are **presentation elements** that differ from model elements. If a symbol is deleted, the model element is still present in the model. The model element will be deleted when one of the following applies:

- the element is deleted in the Model View browser
- the command Delete from Model is performed on the symbol.

If you change the name of an attribute in a class symbol, this change will also take place in other presentations of the class.

If you add a new attribute to the class (either in the model browser or in one of the presented class symbols), this will not automatically appear in all presentations. The attribute is of course available in the model so that it can be conveniently added in the class symbols where it is wanted to visualize this property.

If you Delete an attribute in a class symbol, this will only remove the presentation of the attribute in that symbol. If you delete the attribute in the Model View browser, all presentations of the attribute in different class symbols will disappear.

If you delete the class itself in the Model View browser, the symbols referring to this class will disappear from all diagrams. If the class is referenced in other places, for example as an attribute type in other classes, these references will become unbound when the class is deleted. This is immediately reflected in the diagrams (by a red wave) and also by Autocheck.

# Properties Editor

## Opening the Properties Editor

The Properties Editor is opened by selecting an element in the Model View or in a diagram, and selecting “Properties...” in the shortcut menu. The Properties Editor will open as a docked window. Similar to other editors it can be undocked, or docked at a different location by right-clicking in its title bar. The Properties Editor will stay open until you close it.

### Multiple windows

It is possible to open more than one Properties Editor. This can for example be useful when comparing the values of properties on different elements. To enable this you must deactivate Track selection for one of the Properties Editor windows.

## The Properties Editor View

The view of the Properties Editor consists of the following areas from top to bottom (see Figure 10 on page 88):

- In the top left of the window is shown the selected element, element name and icon.
- An “Options...” button for setting Properties Editor Options for the current window.
- A Filter selection menu that controls which properties of the element that are displayed.
- A “Stereotypes...” button for controlling which stereotypes that are applied to the element. The dialog that is opened when pressing this button is the same as is opened when the **“Stereotypes...”** menu item is chosen in the shortcut menu of an element.
- Controls for viewing and editing properties of the element. This area is dynamically populated with controls based on the edited element and the selected filter.

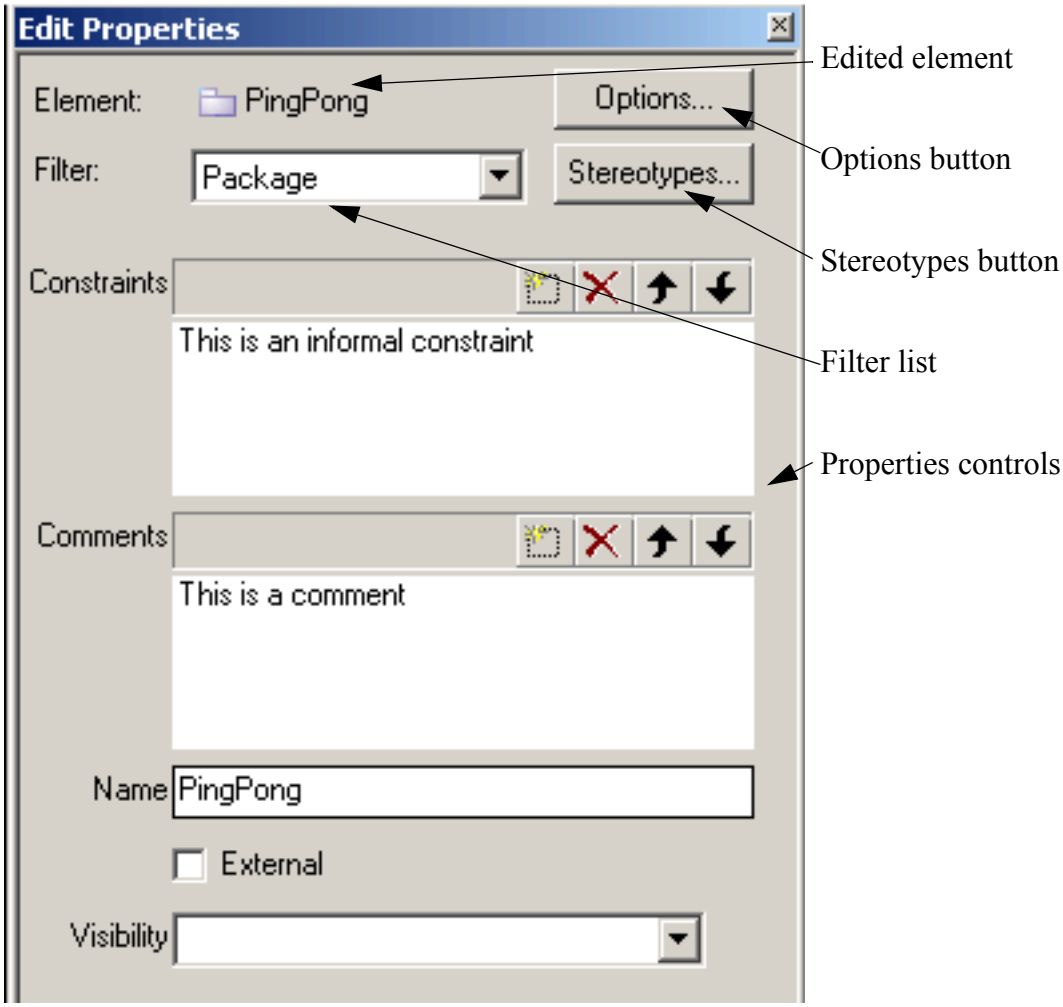



*Figure 10: Properties editor.*

The Filter list consists of the following items (not necessarily in this order though):

- Name of the Metaclass of the edited element.
  When this item is selected the Properties Editor will display the Metafeature values of the element (see "Different Kinds of Properties" on page 89).
- Name of each stereotype that is applied to the edited element.
  This includes both stereotypes with optional (`0..1`) and non-optional (`1`) extension to their metaclass. See "Extensibility" on page 386 for more information about stereotype extensions. However, hidden stereotypes (a stereotype which has the <<hidden>> stereotype applied) are not listed.

- "Comment"
  When this item is selected the Properties Editor will display the comment that is attached to the edited element. If no comment is attached, a button will appear that lets you create a comment for the element. If multiple comments are attached to the element, the first comment will be displayed.
- "All Properties"
  When this item is selected the set of properties is not filtered, and the Properties Editor will display all properties for the edited element. The order of the property controls will be the same as the order of the corresponding items in the Filter list.

### Properties Editor view when selecting an instance

When an instance is selected some of the standard controls of the Properties Editor view described above no longer make sense, and will therefore be removed:

- The Stereotypes button is removed, because it is not possible to apply stereotypes on instances.
- The Options button is removed, because some options are not applicable for instances.
- The Filter list is replaced with information about the signature (e.g. a class) of the instance.

The typical case when you will see this modified Properties Editor view is when an instance is selected in the Model View, for example a stereotype instance. However, an instance can also be selected when editing a tagged value for an attribute typed by a structured type, such as a class.

## Different Kinds of Properties

There are in principle two different kinds of properties that can be associated with the selected element, Metafeature values and Tagged values. The Properties Editor can edit both kinds of properties.

### Metafeature values

These are values for the metafeatures of the element's Metaclass.
The set of metafeatures for an element is fixed (and to some extent dictated by the UML standard), so it is not possible to add new metafeatures. The ex-

isting set of metafeatures can however be filtered so that only values for some metafeatures are displayed in the Properties Editor.
An example of a metafeature value is the “Active” property of a class.

**Tagged values**

These are values for attributes of the stereotypes that are applied to the element. Contrary to Metafeature values the number of tagged values on an element can be arbitrary large since it is possible to apply any number of stereotypes on an element and each of these stereotypes may have any number of attributes.
An example of a tagged value is the “Icon File” property that lets you specify an icon to be displayed for a symbol. Another example is shown in Figure 14 on page 96.

## Properties Editor Options

The Properties Editor Options dialog is reached from the “Options” button of the Properties Editor, see Figure 11 on page 91. The options that are set in this dialog only affects the current Properties Editor, and only as long as it stays open. This means that it is possible to have two different Properties Editors open each of which uses a different set of options.

**Note**

*Some of the options are also available in the general Options, allowing you to set and store options for all Properties Editors that are opened. The values of some options can also be modified using the General Shortcut Menu of the Properties Editor.*

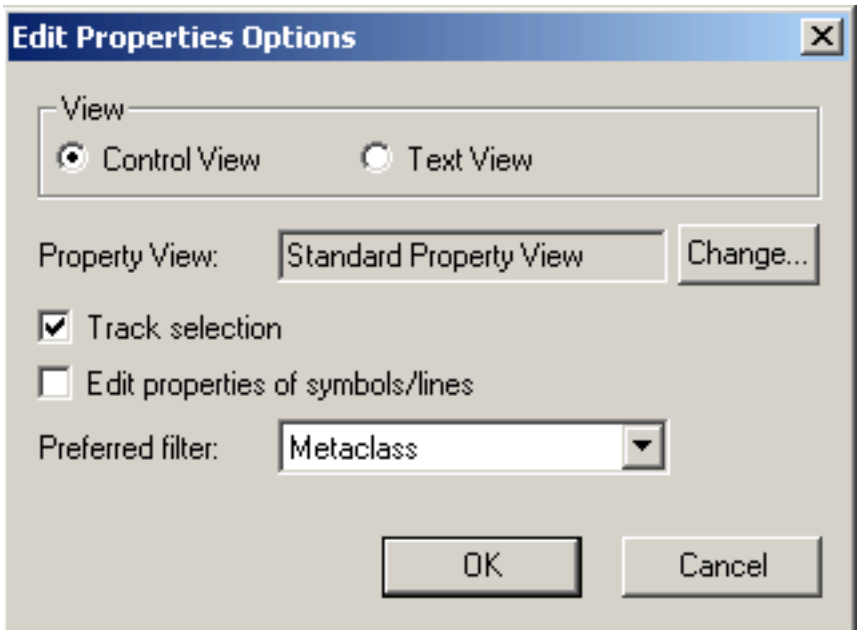



*Figure 11: Options dialog for Properties editor.*

### View

By default the Properties Editor uses the Control View for editing property values. This view contains controls for editing element properties in form of check boxes, pull-down menus etc. For Tagged values textual editing in UML syntax is also possible. This is supported by using the Text View.

Textual editing has the main benefit of giving a compact description of tagged values. Furthermore it is also easier to copy tagged values from one Properties Editor to another or from a text diagram or text symbol if Text View is selected. An alternative to using the Text View for editing tagged values of an element is to use a Stereotype Instance symbol in a diagram.

Text field values entered in the Control view will be committed to the model when you leave edit mode for the field.

### Property view

The Properties Editor can be customized using a Metamodel. Such a meta-model controls for example which metafeatures that are available for each Metaclass, and the Properties Editor will use this information when deciding which properties to display for an element. See "Customizing the Properties Editor" on page 98 for more information on how to use metamodels.

### Track selection

By default the Properties Editor will show properties for the element that is selected in the Model View or in the diagrams. Sometimes it is useful to turn off this selection tracking to be able to compare the properties of two different elements. This is done by opening the Properties editor for the element you want to be fixed. Then point to the **Options...** button and in the dialog make sure that **Track selection** is not selected. Now you can open another Properties editor for any other element, this new properties window will then track your selection in the model.

### Edit properties of symbols/lines

If a symbol or line is selected the Properties Editor will by default show the properties for the model element that corresponds to that symbol or line. In order to show the properties of the selected symbol or line instead this option should be selected.

For example, if a class symbol is selected, the Properties Editor will normally show the properties for the corresponding class. However, if the **Edit properties of symbols/lines** option is set, the properties for the class symbol will be shown instead.

### Preferred filter

This option controls which filter item that is the preferred when a new element is selected. Available items are Metaclass, Stereotype, Comment and All Properties. The option will take effect the next time the edited item is changed.

## General Shortcut Menu

The Properties Editor has a shortcut menu that will appear if the right mouse button is clicked in the editor view outside a control. The menu is shown in Figure 12 on page 93.

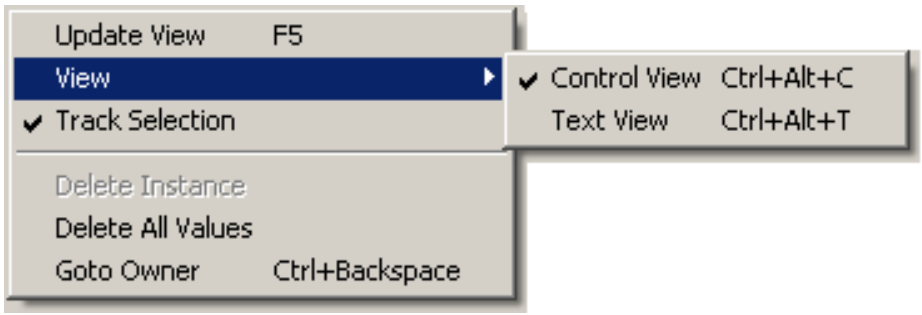



*Figure 12: Shortcut menu in Properties editor.*

### Update View

Refreshes the Properties Editor view. The Properties Editor will normally update its view automatically if values are changed from outside the Properties Editor. However, there are situations when it is necessary to force an update, for example if new attributes are added to an applied stereotype whose attribute values are currently displayed, or if the active property Metamodel is changed when the Properties Editor stays open.

### View

This menu item is a shortcut for the corresponding option in the Options dialog, see "Properties Editor Options" on page 90.

### Track Selection

This menu item is a shortcut for the corresponding option in the Options dialog, see "Properties Editor Options" on page 90.

### Delete Instance

This menu item is available when editing Tagged values for one single applied stereotype. It will delete the entire stereotype instance, effectively removing all tagged values contained in that instance. It can be seen as a shortcut for opening the "Stereotypes" dialog and removing the edited stereotype from the list of applied stereotypes.

### Delete All Values

This menu item will delete the values of all displayed properties. Those properties that have a default value will obtain that value, others will be unspecified. In the case of editing Tagged values, this menu item will remove all tagged values, but keep the applied stereotype instance.

### Goto Owner

This is a convenient shortcut for going to the property page of the edited element's owner. For example, if the properties of a class attribute is edited, "Goto Owner" will display the properties for the attribute's owner, i.e. the class.

**Note**

*Some menu items of the Properties Editor shortcut menu are not available when an instance is selected for editing.*

## Control Shortcut Menu

There is also a shortcut menu for each property control. The exact contents of this menu depends on the kind of property control. For example the edit controls have the standard Cut/Copy/Paste menu items. The menu items shown in Figure 13 on page 94 are common for all property controls.

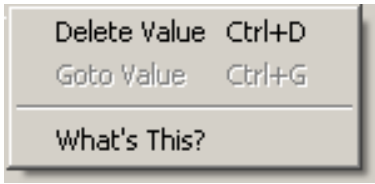



*Figure 13: Property control, shortcut menu example.*

### Delete Value

This menu item will delete the value of the property control. If the property has a default value, it will obtain that default value, otherwise it will get an unspecified value.

### Goto Value

The list controls may show a value that is a list of other elements in the model. For such controls the Goto Value menu item will navigate to the selected element of the list.

For example, most elements have a Comments list which display the list of all comments that are attached to the edited element. If one of these comments are selected, the Goto Value menu item will be enabled and if chosen the Properties Editor will display the properties of the selected Comment instead (it typically just has one property - the comment text).

### What's This?

If the attribute that corresponds to the property control (i.e. an attribute in a stereotype or in a Metaclass) has a comment attached, this menu item will be enabled. If chosen, that comment will be displayed in a tool tip. Stereotype and Metamodel designers should use the possibility to add comments to stereotype and metaclass attributes in order to help the user of the stereotype or metaclass to know which value that should be entered in the property control.

For certain controls (for example those showing Metafeature values) a standard What's This? text may be displayed even if the attribute has no comment attached. Such a text appears when the value to be entered in the control is text that is translated into model elements. The tool tip then displays the kind of text that should be entered into the control. For example, if a UML expression should be entered in the control the tool tip may say "Expression".

## Color Codes

When editing Tagged values (i.e. not Metafeature values) the Properties Editor uses a color coding scheme for showing the status of a tagged value.

A tagged value that has been specified explicitly in the applied stereotype instance is indicated by displaying the property control in a white color.

A tagged value that is unspecified in the stereotype instance, but for which the corresponding stereotype attribute has a default value, is indicated by displaying the property control in a green color.

A tagged value that is unspecified in the stereotype instance, and for which the corresponding stereotype attribute has no default value specified, is indicated by displaying the property control in a yellow color.

These color codes should be used by the designer of a stereotype, to express the intent of the stereotype attributes to the user of the stereotype. A green value signals that it is optional to specify a value for the attribute, since there is an appropriate default value. A yellow value signals that the user should specify a value, since no appropriate default value is available for that particular attribute.

**Example 2: Stereotype with colored attribute fields**

Consider a stereotype with three attributes. In Figure 14 on page 96 the stereotype `MyStereo` is applied to a class X. The user specifies a value for the second attribute, thus the color for this field will change from yellow to white.

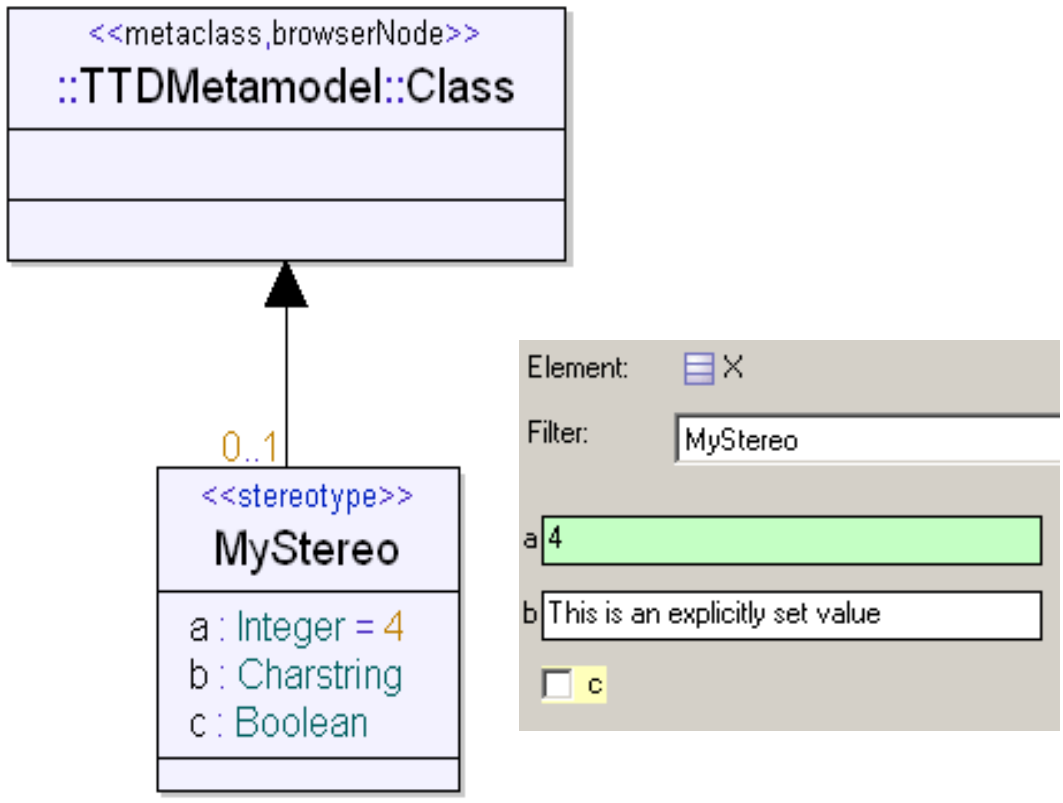



*Figure 14: Stereotype with attributes.*

Another colorization that is used is to show whether the text of a control contains a syntax error. Such syntax checks are made for all controls whose text must comply with the U2 textual syntax grammar. Text containing a syntax

error will be shown in red, while correct text will be black. If you leave editing while the text for such a control is red, the value will go back to its previously correct value. This colorization is thus a help to avoid accidentally loosing information while editing.

**Example 3: Syntax error colorization in the Properties Editor**

The ‘Realizes’ metafeature of a Port expects a list of identifiers. The current text (see Figure 15 on page 97) for the metafeature contains a syntax error since ‘signal’ is a UML keyword. Hence, if leaving edit mode now, that value will go back to the previously correct value (whatever that is).


*Figure 15: Correct and incorrect metafeature values.*

# Customizing the Properties Editor

When designing a stereotype to be applied to an element, two user roles can be identified; the designer of the stereotype who decides which attributes the stereotype shall have, and the user of the stereotype who applies it to an element and specifies Tagged values for the stereotype attributes. Although these roles could be possessed by the same individual it is very common that the designer and the user of a stereotype are two different people.

This section will address the designer role, describing how to design a new stereotype or Metaclass. This also includes how to utilize the various possibilities for customizing the Properties Editor to edit instances of these stereotypes and metaclasses in the way the designer wants.

The Properties Editor uses a profile to control most of its customization, called the TTDExtensionManagement Profile, and is available in the Library folder of any model.

## Designing a Stereotype

The following steps should be taken in order to design a stereotype for use with the Properties Editor:

- Decide where to place the definition of the new stereotype. If the stereotype is only intended to be used locally within the current project, it could be added in the same file as the elements on which it should be applied. However, it is typical that a stereotype shall be used in multiple projects, and then it should be placed in a package that is stored in a file of its own. Such a reusable package with stereotypes is typically a so called profile package. See “Add-Ins” on page 1987 for more information on how to load such a package as a library in the tool.
- Give the stereotype a good name. The name of the stereotype will appear in the Stereotypes dialog, in the filter list of the Properties Editor and in some symbols in the diagrams. Sometimes it can be useful to use the `TTDExtensionManagement::instancePresentation` stereotype in order to specify a more user-friendly display name for the stereotype. Such a specified display name will be shown in the Stereotypes dialog and in the filter list of the Properties Editor. See “TTDExtensionManagement Profile” on page 103 for more information.

- Make a comment for the stereotype. This comment should describe the purpose of the stereotype, any constraints on elements onto which it can be applied and so on. The comment will be displayed at the bottom of the Stereotypes dialog, when the stereotype is selected. It will also be displayed as a tool tip for presentations of the stereotype.
- Add attributes with appropriate types and multiplicities to the stereotypes. A stereotype attribute may have any type and Multiplicity, but you should be aware of the subset of types and multiplicities that are supported by the Properties Editor when using its Control View for editing. If an attribute has a non-supported type or multiplicity, values for that attribute cannot be edited in the Control View. Instead the Text View has to be used.

The table below specifies the supported combinations of types and multiplicities, and which graphical control that will be used in each case. See also the table in section "Designing a Metaclass" on page 101 in Chapter 6, *Working with Models* for a listing of the supported combinations of types and multiplicities that are applicable for attributes in metaclasses only.

| **Attribute type and multiplicity** | **Property control** |
|---|---|
| Boolean<br>Single multiplicity | CheckBox |
| Charstring<br>Single multiplicity | EditControl |
| Charstring<br>Non-single multiplicity | EditList |
| Enumeration<br>Single multiplicity | DropDownMenu<br>(one item for each literal) |
| Enumeration<br>Non-single multiplicity | CheckBoxList<br>(one check box for each literal) |
| Structured type (e.g. a class)<br>Non-optional, single multiplicity (1)<br>Attribute is a part (composition) | Group<br>(with one subcontrol for each attribute of the structured type) |

| Attribute type and multiplicity | Property control |
|---|---|
| Structured type (e.g. a class)<br>Optional, single multiplicity (0..1) | InstanceEditControl |
| Structured type (e.g. a class)<br>Non-single multiplicity | InstanceEditList |
| Metaclass type<br>Single multiplicity<br>Reference | DropDownMenu<br>(one item for each visible definition of the metaclass) |
| Metaclass type<br>Non-single multiplicity<br>Reference | EditControl |
| All other types<br>Single multiplicity | EditControl (expecting a U2 expression) |
| All other types<br>Non-single multiplicity | EditControl (expecting a comma-separated list of U2 expressions) |

Naturally, a syntype of any of the above mentioned types are also supported.

- In case the default control for an attribute is not appropriate you may apply the `TTDExtensionManagement::extensionPresentation` stereotype to an attribute, specifying a custom control as a tagged value. See "TTDExtensionManagement Profile" on page 103 for more information.
- It is possible to add additional "non-value" controls to the property page of the stereotype. For example you could add a static text or a button to the property page. This is done by applying the `TTDExtensionManagement::instancePresentation` stereotype to the stereotype and specifying the additional controls as Tagged values for the `nonValueControls` attribute.
- Use the possibility to attach a comment to each stereotype attribute. The comment text will be visible to the user of the stereotype in the What's This shortcut menu item on the control that corresponds to the attribute.
- Consider the possibility of using inheritance between stereotypes. The property page for the derived stereotype will include all base stereotype attributes followed by the attributes of the derived stereotype.

- Specify the kind of elements onto which the stereotype shall be applicable. This is done by establishing an Extension between the stereotype and a Metaclass. The meaning of this is that the stereotype will be applicable to all elements of the specified metaclass. The UML semantics state that if a stereotype lacks extensions, it cannot be applied to any kind of element.
  If you want the stereotype to be automatically available for all instances of the extended metaclass, you should make the extension non-optional (type "1" on the extension line). Thereby the stereotype will be available in the filter list of the Properties Editor without first having to apply it to the edited element.
  If you want the stereotype to be manually applied, you should make the extension optional (type "`0..1`" on the extension line).
  It is allowed to use multiple extensions. The stereotype will be available for all elements that is of any of the specified metaclasses.

Now you are ready to test the new stereotype. Create an element of the correct kind, i.e. an element of a metaclass that is extended by the stereotype. Make sure the stereotype is visible from the location of the created element. Open the Properties Editor on the created element and take a look at the property page for the new stereotype. If you specified an optional extension you should first apply the stereotype, using the "**Stereotypes...**" button.

## Designing a Metaclass

The process of designing a Metaclass is almost the same as when designing a stereotype. The main difference is how to specify the elements for which the metaclass shall be available in the Properties Editor. For a metaclass this is done by applying the `<<metaclass>>` stereotype to the class that describes the metaclass. It is in fact this step that makes it a metaclass instead of an ordinary class. The tagged value for the `base` attribute shall specify the name of the built-in UML metaclass on which the new metaclass shall be based.

**Note**

*A good starting point for learning how to design a metaclass, is to study the TTDMetamodel profile that is available as a library in all models. Here you can find information about the names of the built-in metaclasses and metafeatures to be used as base for your own metaclasses and their attributes. You can also see example of use of the TTDExtensionManagement Profile for customizing the Properties Editor for elements of the specified metaclasses.*
*It is TTDMetamodel that is referred to as "Standard Property View" in the Options dialog of the Properties Editor.*

In contrast to a stereotype it is not possible for a metaclass to specify plain new attributes. All attributes of a metaclass must be based on already existing metafeatures of the base metaclass. This is done by applying the `metafeature` stereotype to the metaclass attributes. If the name of the metaclass attribute is the same as the name of the corresponding metafeature, the `base` tagged value can be omitted. Otherwise it has to be specified.

**Note**

*The careful user will find some metaclass attributes in TTDMetamodel which do not correspond to metafeatures of the base metaclass. These are so called query features, and they use the <<queryFeature>> stereotype to specify a query agent that computes entities from the model. Query features are not displayed in the Properties Editor - only in Model View.*

The table below specifies the supported combinations of types and multiplicities that are applicable for attributes in metaclasses only, and which graphical control that will be used in each case. Compare with the table in section Designing a Stereotype for a listing of combinations that are valid for all Stereotype attributes.

| **Attribute type and multiplicity** | **Property control** |
|---|---|
| Metaclass type<br>Single multiplicity<br>Composition | EditControl |
| Metaclass type<br>Non-single multiplicity<br>Composition | EntityList |

When your new metaclass is ready you will have to place it in a package and store the package in a file of its own. The predefined stereotype `<<propertyModel>>` should be applied on the package. Then you should follow the normal procedure for writing Add-Ins that loads the profile. When the profile has been loaded you can use the "Options..." button of the Properties Editor to specify the profile package as the property view to use with the Properties Editor.

# TTDExtensionManagement Profile

The TTDExtensionManagement profile contains stereotypes and classes that allows you to customize the property pages for your own stereotypes and metaclasses. Here are the details of this profile, and also some examples of how to use it.

### Stereotypes

The profile contains three stereotypes that are relevant for the Properties Editor: `instancePresentation`, `extensionPresentation` and `filterStereotypes`.

## instancePresentation

The `instancePresentation` stereotype may be applied on a stereotype or Metaclass to customize how instances of the stereotype or metaclass will be presented in the Properties Editor.

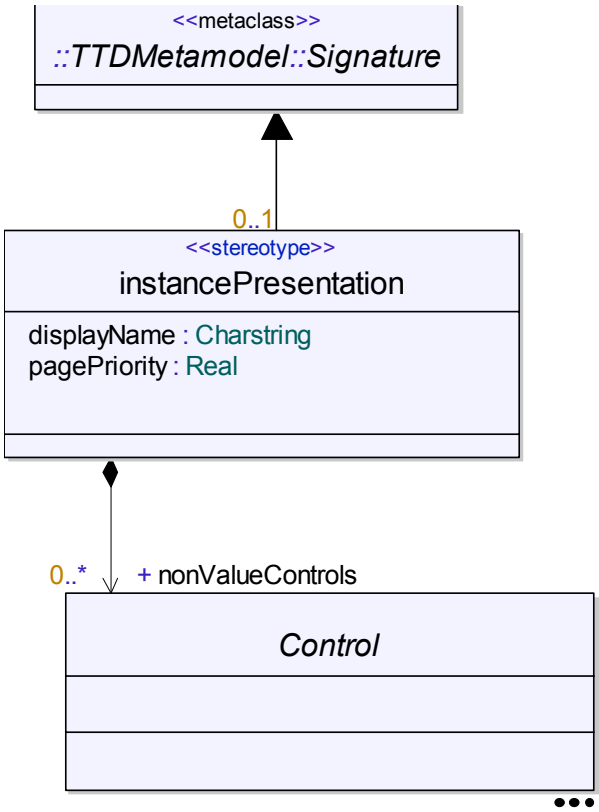



*Figure 16: The <<instancePresentation>> stereotype*

### displayName: Charstring

This attribute specifies the display name of instances of the stereotype or metaclass. It is used in the filter list of the Properties Editor and in the Stereotypes dialog. It is also used in some other places in the tool, such as in tool tips and in the Model View.

If no tagged value is specified for this attribute, the name of the stereotype or metaclass will be used as display name.

### pagePriority: Real

This attribute controls the relative order of two stereotype instances in the filter list of the Properties Editor and in the property page (if the All Properties filter is used). An instance of a stereotype with a higher page priority will be placed before an instance of a stereotype with a lower page priority number. Any specified page priority is considered to be a higher priority value than an unspecified page priority.

**Note**

*If you want to specify a page priority you must use a single numeric value. More complex expressions will not be evaluated.*

### nonValueControls: Control[*]

This attribute specifies a list of "non-value" controls, i.e. controls in a property page that do not correspond to a particular attribute. Examples of such controls are "adornments" such as static texts, but it could also be controls with some behavior, such as a Button.

## extensionPresentation

The `extensionPresentation` stereotype (Figure 17 on page 105) may be applied on an attribute of a stereotype or a Metaclass to customize how the Properties Editor will draw the control that corresponds to that attribute.

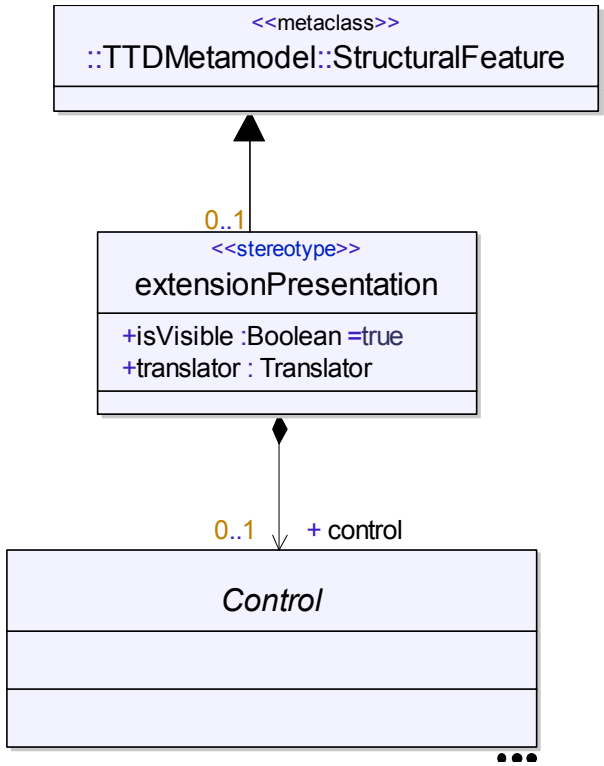



*Figure 17: The <<extensionPresentation>> stereotype*

### isVisible: Boolean

This attribute controls whether the control for the attribute shall be visible on the property page or not. You may set its value to false in order to hide the control for an attribute completely.

### translator: Translator

This attribute is used exclusively for parts typed by a metaclass. As mentioned in "Designing a Metaclass" on page 101 the Properties Editor uses an EditControl (in case of single multiplicity) or an EntityList (in case of non-single multiplicity) as the control for such an attribute. Since the text that is entered into these controls in this case is UML textual syntax, a parser (translator) is needed to interpret the text. The Translator enumeration contains one literal for each available entry point of the UML grammar. Although the Translator enumeration resides in a hidden (internal) profile, you can find out the names of its literals with the following procedure:

- Create an enumeration symbol in a class diagram by right-clicking and choosing Reference existing.
- In the list that appears, select the `U2ParserProfile::Translator`.
- Right-click on the enumeration symbol and choose "Show Literals" from the Show/Hide submenu.

**Note**

*Use the List References command (available in the shortcut menu) to find out how the Translator literals are used in the TTDMetamodel profile. For example, listing the references for the literal PEP_Multiplicity shows that it is used as the translator of the StructuralFeature::Multiplicity attribute. Thus, this translator is used for parsing the multiplicity syntax of* UML.

### control: Control[`0..1`]

As mentioned in "Designing a Stereotype" on page 98 the Properties Editor uses a default control based on the type and multiplicity, and sometimes also the aggregation kind, of an attribute. The `control` attribute makes it possible to specify that a non-default control shall be used for an attribute, or that some properties of the default control should be changed.

**Example 4: Specifying a custom control using the Text View**

```
extensionPresentation(.
  control = EditControl(.
```

```
        text = "My Control",
        autoLayout = GrowRight
    .)
.)
```

The Control class is an abstract class with one derived class for each control that is supported by the Properties Editor.

## filterStereotypes

The `filterStereotypes` stereotype may be applied on a package to reduce the number of stereotypes that will be shown in the Properties Editor when an element in that package is selected.

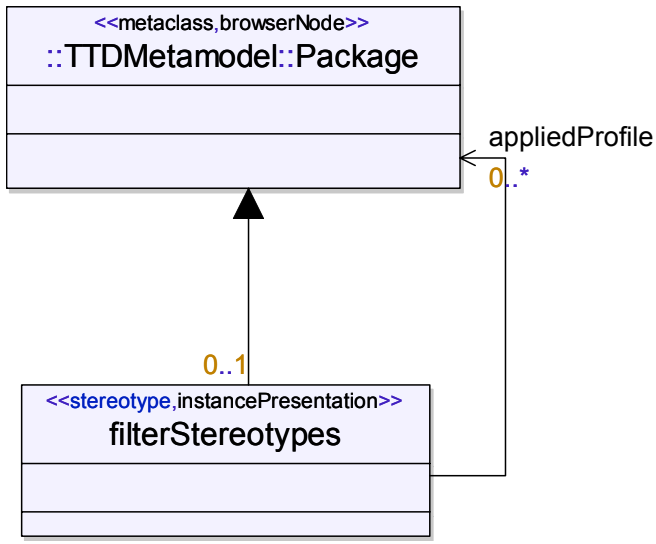



*Figure 18: The <<filterStereotypes>> stereotype*

### appliedProfile: Package[*]

If this list of profile packages is specified, the Properties Editor and the Stereotypes dialog will only show stereotypes defined in these packages for a selected element in the package on which the `filterStereotypes` stereotype is applied.

# Control model

The `TTDExtensionManagement` profile contains a number of Control classes representing graphical controls used by the Properties Editor. See the class diagram `Controls` for an overview of all available control classes.

### Control

The `Control` class is a common base class for all control classes.

### text: Charstring

This attribute controls which caption to use for the control. If it is left unspecified, the caption will be the name of the edited stereotype or metaclass attribute.

### isEnabled: Boolean

By default a control will be enabled, meaning that it can be used for editing the displayed value. If this attribute is set to false, the control will instead be disabled. In some situations the Properties Editor will force a control to be disabled, regardless of the value for this attribute. This happens if the file that contains the edited element is read-only, and also for attributes that are derived.

### onEnable: Operation

This attribute can be used to give a dynamic condition for when a control shall be enabled. If an agent operation is specified here it will be called each time the Properties Editor needs to decide whether the control shall be enabled or not. The model context of the agent call is the edited element. The call has the following parameters:

- `[out] enable : Boolean`
  The agent should set this out parameter to false if the control shall be disabled. By default the control will be enabled.
- `stereotypeInstance : Entity`
  The stereotype instance that is edited in the property page containing the control. This parameter is only passed when the edited instance is a stereotype instance.

**Note**

*When* `isEnabled` *is false the* `onEnable` *agent will not be invoked.*

**See also**

"Agents" on page 2025 in Chapter 75, *Agents*

### Button

The `Button` class represents a button that can be pushed. It is not used for editing a value, but can be used as a non-value control on a property page. `CheckBox` is a special kind of button, a toggle box control, which can be used to edit boolean values.

### onClicked: Operation

This attribute can be used to specify some behavior to be executed when the button is clicked. It may specify an agent operation, which will be invoked when the button is clicked. The model context of the agent call is the edited element. There are no parameters in the agent call.

### EditControl

An `EditControl` can be used for editing string values. There are two specialized versions of the class that can be used when the edited string is a directory name or a filename. They will add a browse button [...] for opening a directory or file selection dialog, as an alternative for manually typing the name in the control.

There is also a special kind of EditControl called `InstanceEditControl`. It is used for editing instances (for example instances of classes). The instance is shown using textual syntax in the control, but to edit the instance there is a browse button [...] which will bring up another Property Editor for editing the selected instance.

### isMultiLine: Boolean

By default an edit control shows exactly one line of text. By setting this attribute to true, the control will enable multiline editing. In order to see more than one line of text at the same time, the vertical size of the control may have to be extended. See PositionedControl for more information on how to do this.

### EditList

An `EditList` control can be used to edit lists of strings. The control contains buttons for creating a new string in the list and for deleting a selected string from the list. There are also two buttons for moving a selected string up or down in the list. A string can also be moved by drag and drop in the list directly.


*Figure 19: An EditList control with buttons for creating, deleting and moving strings.*

There are two specialized versions of EditList that can be used when the edited strings are directory or file names. They are called `DirectoryEditList` and `FileEditList` and will add a browse button [...] for opening a directory or file selection dialog, as an alternative for manually typing the name in the control.

There is also a special kind of EditList known as an `EntityList`, that can be used as the control for metaclass attributes (i.e. metafeatures) that are compositions typed by another metaclass. Each edited item in an EntityList is thus an element in the model. The string displayed in the control for such an element is its textual UML syntax.

Another special kind of EditList is the `InstanceEditList` which is used for editing a list of instances (for example instances of classes). The instances are shown using textual syntax in the control (one instance on each line). To edit one of the instance double-click on it, and press the browse button [...] which appears. Doing so will bring up another Property Editor for editing the selected instance.

### StaticText

A `StaticText` is a non-value control that can be used as an adornment in a property page. It can for example be useful to add a static text for giving instructions to the Properties Editor user on how to specify values in the supplied controls.

### EnumeratedList

This is an abstract class that is the common base for controls that edit lists of enumerated elements. There are two concrete specializations of this class; DropDownMenu and CheckBoxList.

#### items: Charstring[*]

If an enumerated list is used as the control for an attribute that is typed by an enumeration, it will contain one item for each literal of the enumeration. The name of each item will by default be the name of the corresponding literal. However, by specifying a list of strings as the value of the `items` attribute the names of the list items can be customized.

### DropDownMenu

A `DropDownMenu` is a list of items edited in a drop down menu.

#### isEditable: Boolean

By default the user can only select one of the existing items from a drop down menu. By setting this attribute to true, the drop down menu will be editable, allowing the user to type the name of the item manually.

### CheckBoxList

A `CheckBoxList` is a list of items edited in a list of check boxes. Hence this control allows multiple list items to be selected.

### Group

A `Group` is just a container control that can contain other controls. It is typically used as the control for a part attribute of multiplicity 1 typed by a structured type. There is then one contained subcontrol for each attribute of the structured type.

### ColorControl

A `ColorControl` can be used for attributes of integral type. The value of such a control is interpreted as a color reference, with three components; Red, Green and Blue (RGB).


*Figure 20: A ColorControl with green color as value*

The color value can either be edited using a standard color picker dialog (opened by clicking on the arrow button), or the RGB value can be typed directly using the syntax RGB(<red>, <green>, <blue>).

### QueryControl

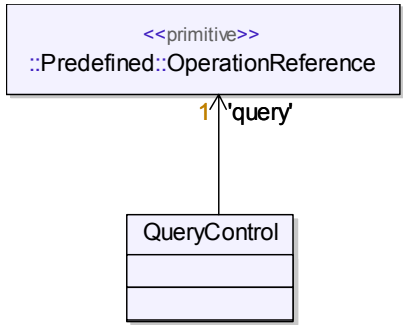



*Figure 21: QueryControl definition*

A `QueryControl` has a similar appearance as a DropDownMenu, but instead of having a fixed list of entities, the list is dynamically populated by executing a query (see Queries). The value of the control is a reference to the entity that is selected from the query result.

#### query: Operation

This attribute is a reference to the query agent to execute in order to populate the list.

### NavigationButton

A `NavigationButton` can be used as the control for metafeatures of single multiplicity that are typed by a metaclass. This means that the value of the control is a reference to another entity in the model. When the button is pressed the property page for that entity is shown.

Navigation buttons can be used when there is a relationship between two entities in a model to make it easier to reach the property page for one of the entities from the property page of the other entity.

### GotoOwnerButton

A `GotoOwnerButton` is a special kind of NavigationButton which always performs navigation to the composition owner of the edited element.

### ValueControl

Some control classes inherit the `ValueControl` class, representing controls that can display and edit a value.

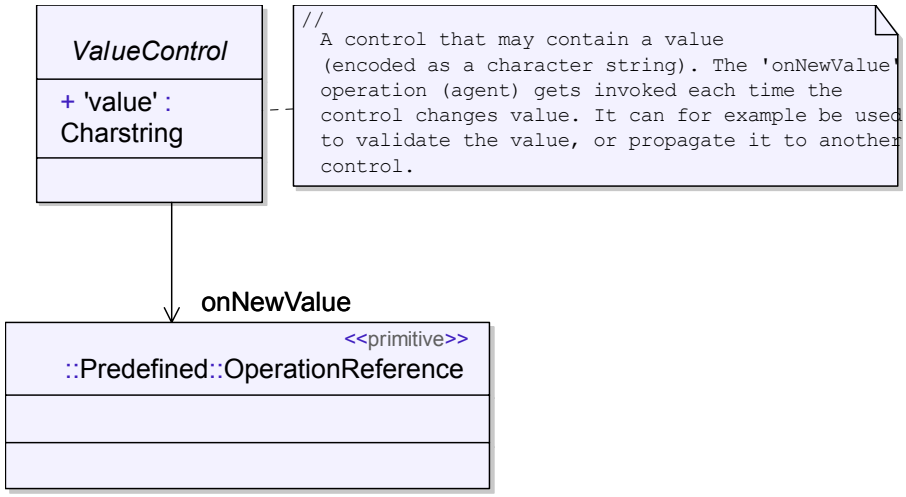



*Figure 22: ValueControl classes*

#### value: Charstring

This attribute is used internally by the Properties Editor to hold a representation of the control's value. However, it can also be explicitly specified to force a control to always show a particular value.

#### onNewValue: Operation

This attribute specifies an agent operation which will be invoked each time the control gets a new value. It can be used as a means for validating the entered value of a control, or to propagate a value to another control. The agent will be called just before the new value is set, with the edited element as its model context, and with the following parameters:

- `attribute : Entity`
  The edited attribute (stereotype or Metaclass attribute)
- `newValue : Entity`
  The new value to be set to the control.
- `stereotypeInstance : Entity`
  If the edited attribute is a stereotype attribute, this parameter is the stereotype instance that is about to be modified. Otherwise this parameter is not passed.

### PositionedControl

The `PositionedControl` class represents those properties of a control that are related to its graphical position and size. By default the Properties Editor applies a simple kind of autolayout for determining where a control shall be positioned. Attributes will be positioned left aligned and top-down, and autolayout position for a control is calculated relative to the preceding control. The attributes of the PositionedControl class makes it possible to customize this layout to some extent.

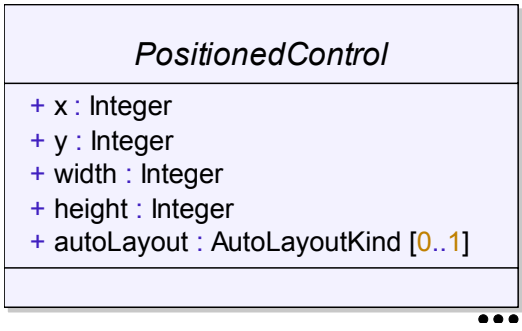



*Figure 23: PositionedControl.*

#### x: Integer

Specify a value for this attribute to override the default horizontal position of the control.

#### y: Integer

Specify a value for this attribute to override the default vertical position of the control.

**Note**

*To override the default placement of a control you must give a value both for the x and y attributes. The position you give to a control will affect a succeeding control that uses default layout.*

**width: Integer**

Specify a value for this attribute to override the default width of the control.

**height: Integer**

Specify a value for this attribute to override the default height of the control.

**autoLayout: AutoLayoutKind**

This attribute specifies an option to the autolayout algorithm that decides how a control is affected by resizing the Properties Editor window. The following values can be used for this attribute:

- `GrowRight`
  The size of the control grows at its right side when the size of the Properties Editor window is increased. This is the default behaviour for most controls.
- `GrowBottom`
  The size of the control grows at its bottom side when the size of the Properties Editor window is increased.
- `GrowRightAndBottom`
  This size of the control grows at both its right and bottom sides when the size of the Properties Editor window is increased.

# Create Presentation

The **Create Presentation** dialog provides a natural entry point to models, as an alternative to using the **New** command to Create diagrams from the Model View. This dialog is opened from the shortcut menu of any element.

### Create Presentation dialog

The **Create Presentation** dialog has a title and a set of tabs. The dialog title shows the type and name of the current entity that the Create Presentation is focused on. The tabs each contain a tab description and a list of alternatives.

A click on an alternative in a tab closes the dialog, creates model elements, symbols, lines or diagrams as needed, and navigates to them.

### New Symbol

With the **New Symbol** tab, you can create a symbol for the current entity either in an existing diagram or create a new diagram containing a presentation element for the entity.

### New Diagram

The **New Diagram** tab follows the Model View creation rules. From this tab you may create a diagram below the current entity. This is equivalent to using the Model View shortcut menu **New** for creating diagrams.

### Location column

The location of the alternative in the model.

### Diagram Name column

Name of the alternative.

### Item Type column, Diagram Type column

The type of the described entity. For instance: `Class`, `ClassSymbol` or `ClassDiagram`.

**See also**

"Model navigation/creation" on page 118

"Add symbols" on page 177 in Chapter 7, *Working with Diagrams*

# Model Navigator

The Model Navigator is a tab, named Navigate, in the Output window that allows you to browse and navigate through various aspects of any entity in a model.

The key purpose of the Model Navigator is to provide a natural and powerful tool for navigation in the model. While the Model View displays the model based on a hierarchical scope view, the Model Navigator has a number of different views allowing you to cross-examine the model based on the model's internal relations.

The model navigator also allows you to:

- Select and display diagrams.
- Navigate to a symbol or line representing the current entity.
- Take navigation shortcuts to entities related to the current entity.

If you select **Model Navigator** from the Model View shortcut menu or an editor shortcut menu, then the Model Navigator will be opened.

### Model navigation/creation

When you double-click an element in a diagram or in the Model view a model navigation/creation will be activated

- If there is any presentation element representing the double-clicked element the diagram with this element will become active and the Navigate tab will be activated.
- If there is no presentation element representing the double-clicked element the Create Presentation dialog will be opened.

## Model navigator tabs

The Model Navigator tab itself has a set of tabs. These tabs each contain a tab description and a list of alternatives. The set of tabs depends on the current entity. The start tab in the window will be selected by using the following criteria:

- Latest used tab
- Highest priority of applicable tabs

Column widths may be resized by dragging the vertical bar to the right of each column header.

### Sorting

Alternatives in tab lists are initially sorted in ascending order based on the name column. For tabs without a name column, the type or index number column is used instead.

Manual sorting is done by clicking on a column header. Repeated clicks will reverse the sort order.

## Tab categories

The Model Navigator tabs can be categorized into two main groups:

- Tabs that show the alternative in the Model View or in a diagram. In this group you find the **Presentation tabs** and the Links tab.
- Tabs that refocuses (on CTRL + click) the Model Navigator on a new model element. This type of tabs are called **Entity tabs**.

Below, you will find more information on the different tab groups.

- **Presentation tabs**

  A click on an alternative in a presentation tab navigates to a symbol or line in a diagram (the Symbols tab), or to a diagram itself (the Diagrams tab).
- **The Links tab**

  A click on an alternative in the Links tab closes the dialog and navigates to the other link endpoint.
- **Entity tabs**

  On CTRL + click on an alternative in an entity tab the Model Navigator refocuses on the clicked alternative, which becomes the new current entity. The new current entity is selected in the Model View, if possible. In this category, you find the **Package, Features, Bookmarks, Definitions, Shortcuts, References, Model Index** and **Recent** tabs.

The Model Navigator tabs are ordered according to the table below.

| Priority | Tab name | Category |
|---|---|---|
| 1 | Symbols | Presentation |
| 2 | Diagrams | Presentation |
| 3 | Links | Link |
| 4 | Package | Entity |
| 5 | Features | Entity |
| 6 | Bookmarks | Entity |
| 7 | Definitions | Entity |
| 8 | Shortcuts | Entity |
| 9 | References | Entity |
| 10 | Model Index | Entity |
| 11 | Recent | Entity |

## Navigation

Double-clicking on an alternative will show the alternative in both the Model View and a diagram (if this is possible to do).

Holding down CTRL while you click or double-click will refocus the model navigator on the clicked alternative, as well as show the alternative in both the Model View and a diagram (if this is possible to do).

Holding down SHIFT while you click or double-click will show the alternative only in the Model View, not in a diagram.

The tab and alternative shortcut menus in the Model Navigator contain a list of recent Model Navigator entities. This list allows you to refocus the Model Navigator on an entity that recently has been the current Model Navigator entity.

## Presentation tabs

### Symbols

The Symbols tab shows symbols and lines related to the current entity.

### Diagrams

The Diagrams tab shows diagrams closely related to the current entity.

## Links

The Links tab contains a list of incoming and outgoing hyperlinks for the current entity. Click on a link to navigate to the other link endpoint associated with the link.

## Entity tabs

### Package

The Package tab shows a complete list of definitions visible in the package containing the current entity.

### Features

If the current entity is a class or something similar (more precisely: If the current entity is a Classifier or is contained in a Classifier), then the Features tab lists the definitions in that class, together with any inherited definitions.

### Definitions

The Definitions tab shows a complete list of local and inherited definitions in the scope of the current entity.

### References

The References tab will list Model references to the current Definition, for quick navigation to the places where the Definition is used. This information is similar to the Model View shortcut menu choice List references.

### Shortcuts

The Shortcuts tab provides quick navigation through some commonly used relationships of a model. The most common shortcuts are described in the text about the Shortcut column.

### Bookmarks

The Bookmarks tab provides a method for setting and navigating through bookmarks, to select places in the model that you anticipate re-visiting. The contents of this tab will only be persistent over the current tool session. Adding and removing items from the list is done by clicking on the **Add/Remove** and **Remove all bookmarks** rows in the list.

From the shortcut menu for any model element in the Model View you can choose **Bookmark** to add the selected element to this list.

### Model Index

The Model Index tab contains an alphabetical list of all definitions in the model with the exception of unnamed parameters (return parameters). See also description of the Find dialog.

### Recent

The Recent tab keeps track of entities that the Model Navigator has been focused on, allowing you to refocus the Model Navigator on any of your recently visited entities. You can as an alternative to this tab use the shortcut menu, which contains the 5 most recent Model Navigator entities.

## Columns

Below is a list of the columns appearing in the Model Navigator and a short description of the listed information.

### Index column

This column can be found in the Recent tab and in the Bookmarks tab. It contains numbers indicating the order that entities were visited in. A lower number means a more recently visited entity.

### Links column

The number of incoming and outgoing links to and from the current entity.

### Location column

The location of the alternative in the model.

### Name column, Diagram Name column

Name of the alternative.

### Page column

The diagram page number. This column can be found in the Diagrams tab.

### Role column

The role the current entity plays in the listed reference. This column can be found in the References tab.

### Shortcut column

This column contains a list of shortcuts from the current entity to various related entities. This column can be found in the Shortcuts tab. Here are a couple of examples on shortcuts that may appear in the Shortcut column:

- The **Scope** shortcut: Refocus the Model Navigator on the scope entity that contains the current entity.
- The **Container** shortcut: Refocus on the entity that owns the current entity.
- The **Model Root** shortcut: Refocus on the model root for the current entity. This shortcut is especially useful when having a workspace with more than one model.
- The **Predefined Package** shortcut: Refocus on the internal library of predefined types.

### Type column, Item Type column, Diagram Type column

The type of the described entity. For instance: `Class`, `ClassSymbol` or `ClassDiagram`.

### Views column

Number of symbols and lines representing the current definition.

# Generate Diagram

Tau supports automatic generation of diagrams in order to visualize existing model elements. There are a number of built-in diagram generators available, for generating commonly useful diagrams, such as inheritance diagrams, composition diagrams, dependency diagrams etc. It is also possible to add additional custom diagram generators to support specific visualization needs.

To generate a diagram, follow these steps:

1. Select an element in the Model View. The selected element will be the context of the generated diagram. For example, if you want to visualize super- and sub-classes of a certain class, then you should select that class.
2. In the context menu select **Generate Diagram** and choose which diagram generator to use in the sub menu. For example, to generate an inheritance diagram, select “Generate inheritance view”.

The generated diagram is typically placed under the selected context element, but some diagram generators may place it elsewhere in the model, for example as a top-level diagram, or in a separate package. Afterwards you may move it to where you want it to be.

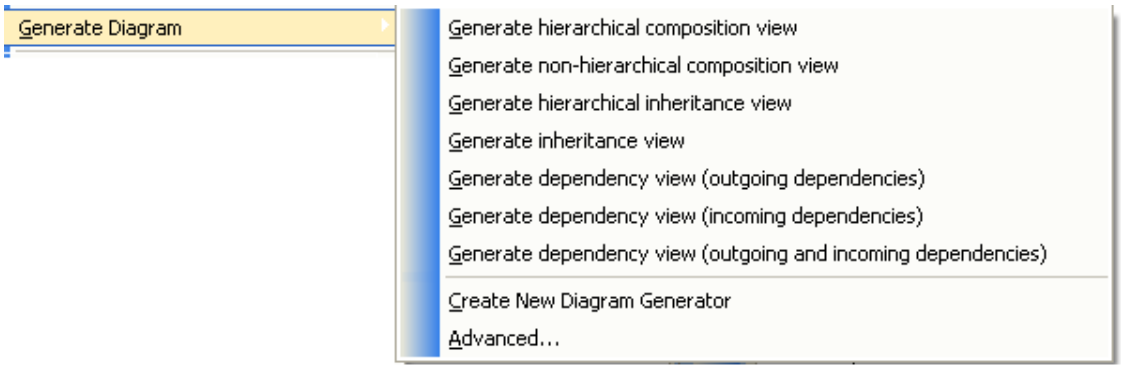



*Figure 24: The Generate Diagram context menu*

## Diagram Generation Parameters

Diagram generators may take actual parameters to control how the diagram shall be generated. For example, when generating an inheritance view for a class, a parameter controls whether the diagram only shall show immediate super- and sub-classes, or if all recursive super- and sub-classes shall be shown.

When a diagram generator is run from the **Generate Diagram** context menu, default values are used for these parameters. To change the actual parameters use the command **Edit Diagram Generation Parameters** available in the context menu of the generated diagram. You can also edit them by opening the Properties Editor on the generated diagram, and selecting the <<generated>> stereotype as filter. Parameters can then either be edited textually in the Parameters field, or you may press the **Edit Parameters** button.

## Regenerate Diagram

A generated diagram can be regenerated based on new information in the model. For example, you may want to regenerate an inheritance diagram when new super- or sub-classes have been added. You may also want to regenerate a diagram if you have modified the Diagram Generation Parameters.

To regenerate a generated diagram choose the **Regenerate** command available in the diagram context menu. It is also possible to regenerate all generated diagrams in the model by selecting the **Regenerate All Diagrams** command in the Tools menu. Only generated diagrams are affected by these commands.

**Important!**

*When a diagram is regenerated everything it contains will be deleted and regenerated. This means that if you have made manual modifications to the diagram, such as changing layout, colors etc., these changes will be lost.*

### Convert a generated diagram into an ordinary diagram

To avoid accidentally regenerating a generated diagram that has been modified, it is recommended to convert the diagram into an ordinary non-generated diagram if you want to maintain it manually. To do so follow these steps:

1. Select the generated diagram in Model View.
2. Select **Stereotypes...** in the context menu.
3. Uncheck the 'generated' checkbox and press **OK**.

After this it is no longer possible to regenerate the diagram.

## Using Diagram Generators in Existing Diagrams

A diagram generator doesn't have to always generate a new diagram. It is also possible to use a diagram generator in order to add information to an existing diagram. The steps to do so are:

1. Drag the context element from the Model View into a diagram using the right mouse button.
2. Drop the element on the diagram and select **Visualize in Diagram** in the context menu that appears.
3. In the sub menu select which diagram generator to use.

The symbols and lines generated by the diagram generator will be inserted in the diagram where the entity was dropped.

## Advanced Diagram Generators

In addition to the diagram generators you will find in the **Generate Diagram** context menu there are also a few more advanced diagram generators. To use these diagram generators select the **Advanced...** command in the **Generate Diagram** context menu. This will open the Advanced Generate Diagram dialog:

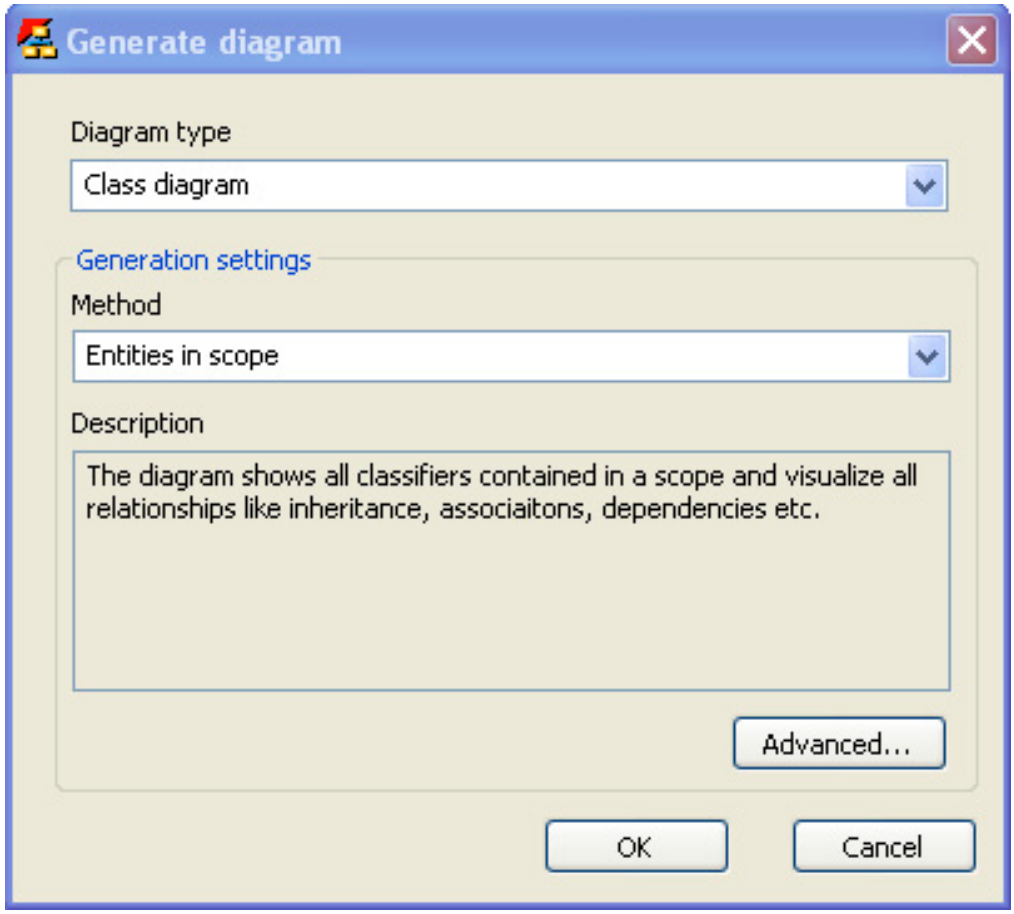



*Figure 25: The advanced diagram generator dialog*

This dialog only provide a limited set of diagram generators, but each of these diagram generators have several customizable layout options.

### Diagram type

First thing to do in the Advanced Generate Diagram dialog is to select the wanted diagram type. Only the diagram types where there is a method to generate a diagram is displayed.

### Generation settings

Secondly, the **Method** of generation can be selected. Only methods of generation that can be applied for the above selected diagram type is available.

A description of the selected generation method is displayed below the list of available generation methods.

Settings for the selected generation method are available by pressing the **Advanced** button. These settings will be associated with the generated diagram and can be edited after generation in the Properties Editor.

## Customization

It is possible to create your own diagram generators in order to generate custom diagrams. See Adding Diagram Generators for more information on this topic.

It is also possible to invoke diagram generators programmatically. This can be useful for example when implementing add-ins. See Invoking Diagram Generators Programmatically for more details.

# Queries

This section describes how to perform a query on a UML model in order to find entities that fulfill certain conditions.

Queries are useful for finding entities in the model that cannot be found by using the more basic search facilities of the Find dialog. Using a query is an alternative to using one of the standard APIs for finding the wanted information. Since a query may contain calls to many of the available API functions (COM, C++, Tcl), the expressive power of a query is equivalent to using the APIs.

### Concepts

A **query** is an operation that yields a collection of entities from the model.

A **predicate** is an operation that yields a boolean true or false.

Both a query and a predicate may take any number of input arguments. One input argument that always is implicitly present is the **model context**. This is an entity on which the query or predicate is invoked.

In order to be able to define query and predicate operations in a UML model, there is a built-in library called `TTDQuery`, which defines the stereotypes in Figure 26 on page 129. See also "The Query Dialog" on page 133.

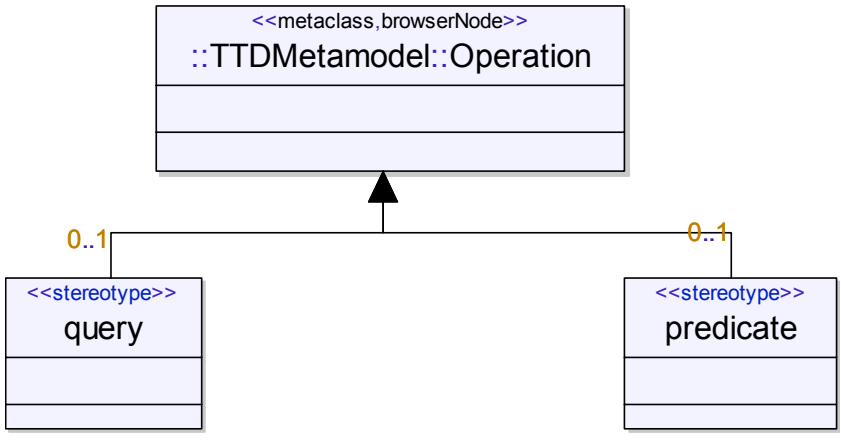



*Figure 26:* `TTDQuery` *library with stereotypes*

In addition to these stereotypes, the `TTDQuery` profile also contains a number of built-in queries and predicates, that are ready to use.

Calls to queries and predicates can be put together in a Query expression. This is an expression that can be interpreted by Tau, and just like when invoking a query operation, the result of the interpretation is a collection of entities from the model. A query expression may use boolean operators and literals, as well as a small subset of the collection operators that are found in OCL, in order to modify the result obtained by calling queries and predicates.

**Note**

*Many operations of the public APIs work as either queries or predicates. These operations are also available for use in query expressions. The UML definition of these API operations can be found in the library called* `u2`.

## Query expression

A query expression is written in textual UML expression syntax. The type of a query expression must be a collection of entities. This means that when a query expression is interpreted the result should be a collection of entities.

All sub-expressions that are contained in a query expression must be of either boolean type, or the type should be a collection of entities. For expressions of boolean type, the usual boolean operators can be used. The following boolean operators and literals are supported within a query expression:

```
and (&&)
or (||)
not (!)
true
false
```

Parentheses can also be used.

For expressions whose type is a collection of entities, a number of predefined Collection Operators may be used.

## Collection Operators

Certain predefined operators may be used on the collection of entities that result from executing an expression within a query expression. The names and semantics of these operations come from OCL (Object Constraint Language). In fact, a query expression is a legal OCL expression, except that periods (.) are used instead of the arrow notation (->) when invoking a predefined collection operation. However, only a subset of OCL is supported. This subset allows powerful queries to be performed.

### select

Syntax:

```
select(<boolean expr>)
```

Type: collection of entities

`select` projects one collection of entities into another collection of entities. The resulting collection will contain those entities in the input collection for which the boolean expression evaluates to `true`. Thus, `select` can be used to filter a collection through a predicate.

### exists

Syntax:

```
exists(<boolean expr>)
```

Type: boolean

`Exists` is a boolean operator that returns `true` if there exists at least one entity in the input collection for which the boolean expression evaluates to `true`, otherwise it returns `false`.

### isEmpty

Syntax:

```
isEmpty()
```

Type: boolean

This operator returns `true` if the input collection is empty. Otherwise it returns `false`.

### Examples

Here are some examples of query expressions that uses some of the available Built-in Queries and Predicates, combined with predefined boolean operators and collection operators.

**Example 5**

Find all active classes defined in a package.

[model context = the package]

```
GetAllEntities().select(IsKindOf("Class") and
HasPropertyWithValue("isActive", "true"))
```

**Example 6**

Find all attributes in the model that are directly owned by a class.

[model context = the model, i.e. the Session]

```
GetAllEntities().select(IsKindOf("Attribute") &&
GetOwner().exists(IsKindOf("Class")))
```

**Example 7**

Find all «access» dependencies in the model.

[model context = the model, i.e. the Session]

```
GetAllEntities().select(not
GetTaggedValue("access(..)").isEmpty())
```

This query will obtain the wanted result, but is quite inefficient since it will check for an applied «access» stereotype on each entity in the model. Performance will be greatly improved just by adding a check that the entity must be a dependency. For all entities that are not dependencies, there is no need to invoke the `GetTaggedValue` query.

```
GetAllEntities().select(IsKindOf("Dependency") and not
GetTaggedValue("access(..)").isEmpty())
```

You can rewrite the expression by using the `HasAppliedStereotype` predicate, which is the recommended way to check if a stereotype is applied on an element.

```
GetAllEntities().select(IsKindOf("Dependency") and
HasAppliedStereotype("access"))
```

Finally, it should be mentioned that the most efficient (and also the shortest) query expression for finding the «access» dependencies makes use of the built-in `GetStereotypedEntities` query:

[model context = the «access» stereotype, found in the `TTDPredefinedStereotypes` library]

```
GetStereotypedEntities()
```

As seen in this example there can be alternative query expressions that can be used to obtain the same result. There can be a great difference in execution performance between different semantically equivalent queries, so it is can be worthwhile to consider different alternatives before writing a query expression.

---

## The Query Dialog

The Query dialog allows you to construct a Query expression to execute. The dialog is opened by selecting an entity in the Model View or the diagrams, and selecting the menu item Edit -> Query. The selected entity will be the model context of the query expression.

**Note**

*The model context of the query expression may be a presentation element (e.g. a symbol or a line in a diagram). Thus, if you open the query dialog from a selected entity in a diagram, the selected presentation element will be the model context. Use the context menu "Show in Model View" to find the corresponding element in the Model View, in case you want to run the query on the model element instead.*

The Query dialog lists all available Queries and predicates that can be found in the current model. This list consists of all "built-in" queries and predicates that are supplied in the predefined `TTDQuery` and `u2` libraries, together with all queries and predicates that are defined elsewhere (for example user-defined queries and predicates).

The query expression is executed by pressing the **Execute** button. By default the result will be output in the "Search Result" tab, but this can be changed by typing another tab name in the drop down control.

You may construct the query expression either by writing the expression directly in the edit control, or you can double-click on entries in the list of available queries and predicates. If the selected operation (query or predicate) do not have any formal input parameters, a call to the operation will be added directly at the position of the cursor in the query expression text. If, however, the operation has at least one formal input parameter, a dialog (see Figure 27 on page 134) will pop-up which allows you to provide the corresponding actual parameter for the operation call.

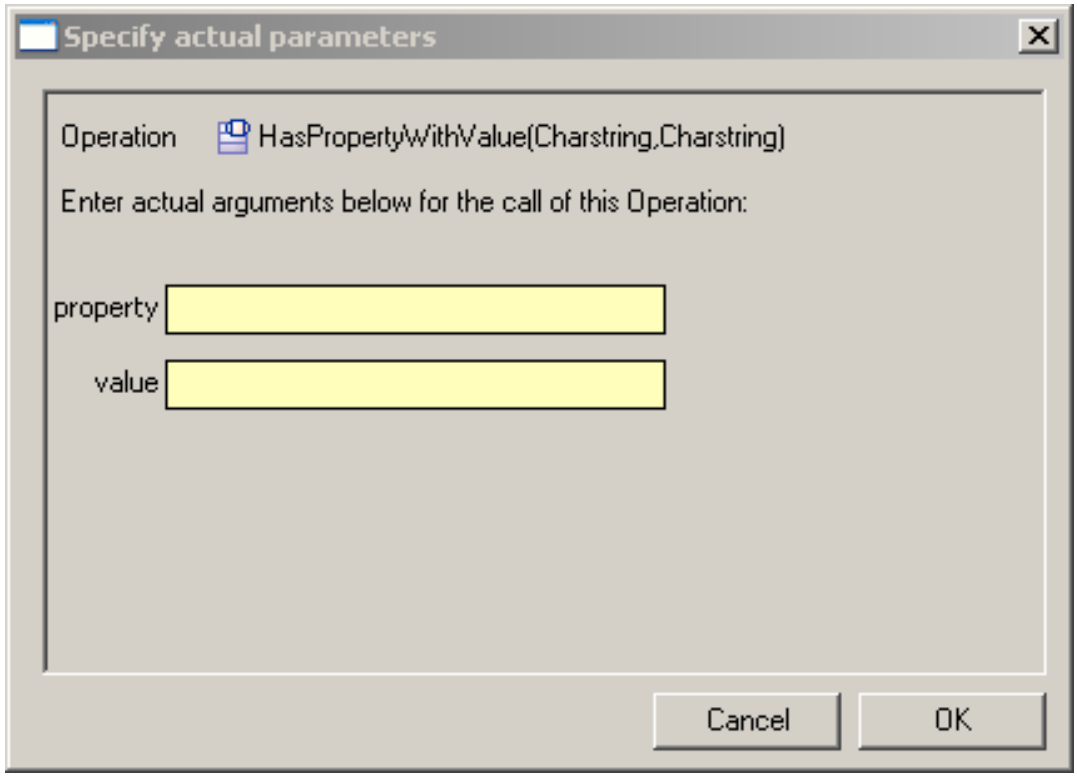



*Figure 27 Specify actual parameters*

This dialog is in fact a Properties Editor (the parameters are seen as properties of the operation) and edited values follow the same Color Codes as the Properties editor. Other features of the Properties Editor, such as obtaining "What's This?" help on the meaning of the parameters are also available.

### Saving a query expression as a new query

The Query dialog has a Save button that allows you to save a query expression as a new query in the model. Use this possibility if you have constructed a query expression that you want to save for the future. You will be prompted to specify a name and description of the new query, as well as a location in the model where it shall be stored. It can be a good idea to put all queries in a common place, for example in a profile package stored in a separate .u2 file. Thereby you can include and use your saved queries in multiple projects.

When you have saved a query expression as a new query, it immediately becomes available in the list of queries and predicates that are ready to use in new query expressions.

## Built-in Queries and Predicates

A number of built-in queries and predicates are available for use in query expressions. These are defined and documented in the profile libraries `TTDQuery` and `u2`.

In addition to these, it is possible to add user-defined queries and predicates as described in User-defined Queries and Predicates.

## User-defined Queries and Predicates

It is possible to define additional queries and predicates than those that are supplied as “built-in”. This is done by defining an agent which has the <<query>> or <<predicate>> stereotype applied. The implementation of such Agents must fulfill the signature of a query or predicate. Thus a query agent must return a list of entities, and a predicate agent must return a boolean value. This mandatory output parameter is passed as the first parameter to the agent. In addition the agent may take any number of input parameters. These parameters may have any type supported by Agents.

## Executing a Query Expression from the APIs

It is possible to execute a Query expression programmatically from all the public APIs, using the agent in Figure 28 on page 135.

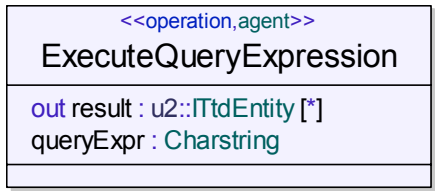



*Figure 28 Agent for query execution*

This agent is (like any other agent) invoked using the InvokeAgent operation.

**Example 8: Executing a query expression from the Tcl API**

This small example shows how to execute the query expression “`GetAllEntities()`” from a Tcl script. The script just prints the Tcl ids of the resulting entities.

```
set s [std::GetSelection]
set a [u2::FindByGuid $U2
```

```
"@TTDQuery@ExecuteQueryExpression"]
set p [lappend p {} "GetAllEntities()"]
u2::InvokeAgent $U2 $a $s p
output [lindex $p 0]
```

# Drag and Drop

This section describes how drag and drop can be used to work with the model.

A drag and drop operation can be done with three different variations of drag sources and drop targets:

- Within the model view.
- From model view to a diagram.
- Within and between diagrams.

A drag and drop operation can be done either with the left or the right mouse button. If a drag and drop operation is done with the right mouse button, a context menu is opened listing the possible operations to perform as a result of the drag from the source element to the target element. The context menu will always have a highlighted alternative. This is the operation that will be performed when a drag and drop operation is done using the left mouse button. There can also be modifier key within parenthesis next to the operation. If so, this operation can be performed by holding down this modifier key will doing a drag and drop using the left mouse button.

Next follows the different operations available using drag and drop.

## Within the Model View

### Move

Moves an element within the model view.

This is the default operation for drag and drop within the model view and will be performed if drag and drop is done using the left mouse button.

### Copy

Copies an element within the model view.

This operation can be performed by doing a drag and drop operation using the left mouse button while holding down the CTRL button.

### Link

Creates a link between the drag source element and the drop target element. The currently active link type will be used.

### Copy with Traceability

Copies an element (including subelements) in the model view and creates <<trace>> dependencies from the copy to the original.

The operation is performed by doing a drag and drop operation using the right mouse button and choosing the "Copy with Traceability" command in the pop-up menu.

The dependencies will be created for all definitions, like e.g. packages, classes, attributes and operations.

**See also**

"Working with links" on page 2409

## From Model View to a Diagram

### Create Presentation

Creates a symbol representing the drag source element in the context of the drop target element.

This is the default operation for drag and drop from the model view to a diagram and will be performed if drag and drop is done using the left mouse button.

### Create Presentation (include lines)

Does the same thing as Create Presentation with the addition that lines representing the drag source element connections to other elements in the drop target diagram will be created.

**Visualize in Diagram**

This is a sub-menu containing the possible diagram generation methods that are available for the drag source element and the drop target diagram. The drag source elements will be visualized in the diagram without affecting any already existing elements in the diagram.

**See also**

"Generate Diagram" on page 124

### Within and between Diagrams

Drag and drop within and between diagrams has the same operations as within the model view.

# Compare and Merge Versions

It is possible to compare versions originating from the same model. The comparison is done between model or project files (`.u2` or `.ttp` extension). Differences can be accepted automatically or manually. When using the compare and merge for more than two versions of the same model it is recommended to use a configuration management system to ensure complete version control.

**Note**

*Model elements will always have a GUID. To be able to use the Compare or Merge features it is necessary that the versions used really originate from the same model. Models created in parallel with identical entity names will have different GUIDs on model elements that otherwise appear to be identical.*

## Merge variations

There are three basic variations of a compare or merge operation. These variations will be referred to as 2- 3- and 4-way compare (merge). The 2-way operation is used to compare or merge any two versions of the same model. The 3- and 4-way merge is performed when working with versions of the same model that have developed on different configuration branches under version control.

- 2-way: Compare or merge two versions of the same model into one.

- 3-way: Compare or merge two versions of the same model into one, taking into consideration their (closest) common ancestor.
- 4-way: Compare or merge two versions of the same model. The 4-way compare (merge) is used to take into consideration a previous merge, where merge decisions from this previous merge will be propagated automatically to the resulting model.

### Configuration

Consider the configuration in Figure 29 on page 140. On the main configuration branch there is at the beginning version A1.

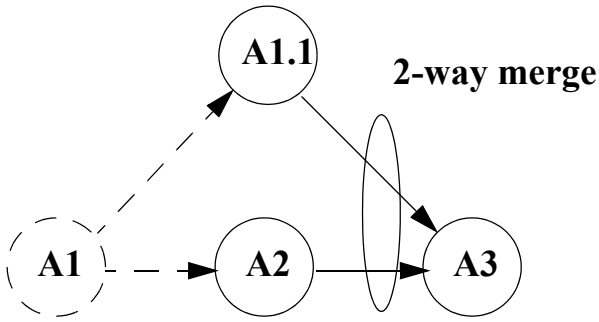



*Figure 29: 2-way merge version tree*

Version A1 is changed. The new version (version controlled) is called version A2.

Version A1 is branched to version A1.1 that later is to be merged with version A2 in order to get version A3. This merge can be done without taking into consideration the relation with A1 as closest common ancestor (**2-way merge,** Figure 29 on page 140) or it can be done with version information from A1 (**3-way merge,** Figure 30 on page 141).

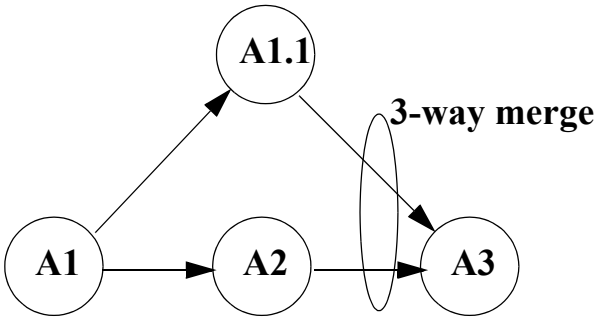



*Figure 30: 3-way merge version tree*

The 2-way merge and the 3-way merge will in the example given both result in version A3. From a configuration management perspective it would be recommended to always use the 3-way merge, thus always considering the common ancestor.

The 2-way merge is sometimes preferred over a 3-way merge. This could be for a number of reasons, some examples:

- The 2-way merge is simpler to run
- You do not have access to the common ancestor version or it is necessary to specify the name manually of the common ancestor and there is a large risk of errors if the wrong common ancestor is specified
- When you need to review all differences manually and there is no need to have any automated support for resolving the differences
- The 2-way merge is only an update where all modifications are checked through the compare feature, this would be the case if the development is not branched or if one of the branches has not changed

Work is continued on version A1.1 as well as on version A3, resulting in versions A1.2 and version A4 respectively. These versions could then be merged by a **4-way merge** into version A5, see Figure 31 on page 142. The 4-way merge will then consider the previous merge (resulting in A3) and the properties that were handled by this merge will not appear, allowing you to focus on merging the changes done for A1.1 to A1.2 and A3 to A4.

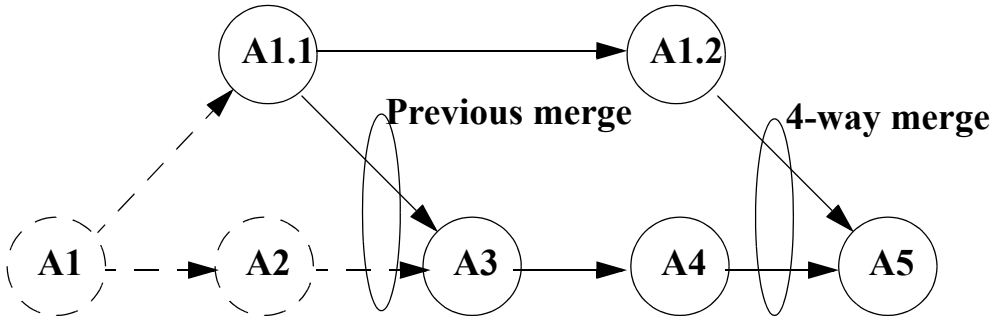



*Figure 31: 4-way merge version tree*

### Name conventions

Given the example above, the name conventions will then name the versions in these different situations to the following. The described 2-way merge and 3-way merge gives the same result. Which to use would depend on

- 2-way merge: The current version is called **Version 1**, A2 in Figure 29 on page 140. A1.1 is called **Version 2**. After the merge is completed A3 is the result.
- 3-way: Considering a merge towards the main configuration branch, A2 is **Version 1**, A1.1 is **Version 2** and A1 is **Common ancestor**. After the merge is completed A3 is the result.
- 4-way: Considering a merge towards the main configuration branch, A4 is **Version 1** and A3 is **ancestor to version 1**. A1.2 is **Version 2** and A1.1 is **ancestor to version 2**. After the merge is completed A5 is the result.

**Note**

*When the term **Common ancestor** is used it is always presumed that it refers to the **closest** common ancestor, as there may be several versions before A1 that are common ancestors to any pair of files in the example.*

### Project Merge

A project to project merge, which contains multiple files, can be performed simply by specifying `*.ttp` files for version 2 and for ancestor 1 and ancestor 2, instead of model (`.u2`) files. Since models normally contain quite a bit of hierarchy, using project merge will aid the automatic merge function in making the right decisions. Merging projects is preferred to merging indi-

vidual model files since it will assure that all the model elements are loaded during the merge. It also means that you can merge all model (`.u2`) files at one time instead of running the one merge for every model file.

### Compare/Merge considerations

There are some issues to think about with regards to Compare/Merge:

- An **ancestor** should be a real ancestor (whenever possible, do not use the same file as version 1).
- The **entire project** should be loaded in the tool (not just one of the files, because a change in one file might affect other files, and said change must be propagated, otherwise inconsistencies may result).
- It is recommended that all files are **saved** before a merge. This is to guarantee that canceling a merge operation returns the model to the state it was in before the merge operation was invoked.

## Compare versions

From the **Tools** menu, select **Compare Versions** to display the dialog window for this feature (see Figure 32 on page 144), containing options for the compare operation.

### Version 2 - read from file

This field contains the name of the file (`.u2` or `.ttp`) containing the version (of the same original model) to compare with. For a 2-way compare this is the previous version of the file. For a 3-way compare this is a version belonging to a configuration-controlled branch. For a 4-way compare this is a version belonging to a configuration-controlled branch, where a previous merge is taken into consideration.

### Common ancestor (3-way) or Ancestor to version 1 (4-way)

For a **3-way** compare this field is used to select a file name (`.u2` or `.ttp`) containing (closest) **Common ancestor**. This version must be the common origin of both Version 2 and the version currently loaded.

For a **4-way** compare this field is used to select a file name (`.u2` or `.ttp`) containing the closest ancestor from previous merge to the version currently loaded. This should be the version that was the result of the latest merge operation with **ancestor to version 2**.

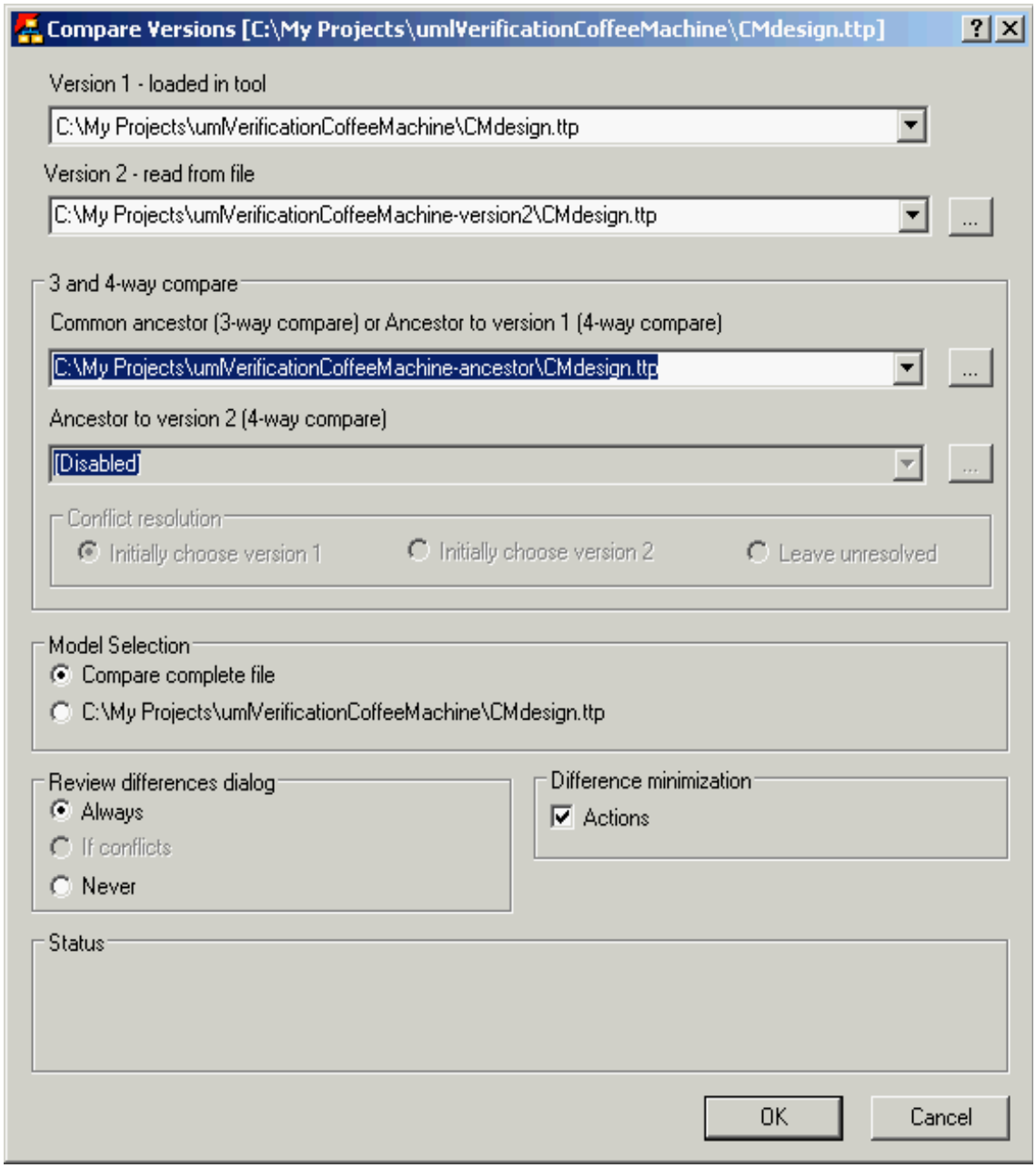



*Figure 32: Compare Versions dialog*

### Ancestor to version 2 (4-way)

A 4-way compare is done with a previous merge as reference. For a 4-way compare the field "Ancestor to version 2" is used to select a file name (`.u2` or `.ttp`) containing the closest ancestor from this previous merge to the version selected in "Version 2 - read from file" field. This would normally be one of the versions used in the previous merge operation that resulted in **ancestor to version 1**.

### Model Selection

- Compare complete file (default): This will apply the compare operation on the complete model. This is recommended in most cases.
- Compare only selected: This will apply the compare operation only on the selection made in the diagram or the Workspace window.

### Review differences dialog

- Always: This option will always present the operation result in a Review differences dialog.
- If conflicts: This option will present the operation result in a Review differences dialog if there are any conflicting definitions, for example no dialog will appear when the differences only consists of model elements that have been added.
- Never: This option will only present a summary of the operation in a result dialog.

### Difference minimization

It is recommended that the **Actions** option is set for most use of the Compare and Merge feature. When **Actions** is set the Compare operation will ignore *GUID* differences that originate from any intermediate model unbinding. This will for example happen when you disconnect a part of a flow to insert a new symbol. The disconnected flow will become unbound for a while and when reconnected the symbols will receive new GUID values.

## Merge versions

From the **Tools** menu, select **Merge Versions** to call up the dialog window for this feature (see Figure 33 on page 146).

### Version 2 - read from file

This field contains the name of the file (`.u2` or `.ttp`) containing the version (from the same original model) to merge with. For a 2-way merge this is the previous version of the file. For a 3-way merge this is a version belonging to a configuration-controlled branch. For a 4-way merge this is a version belonging to a configuration-controlled branch, where a previous merge is taken into consideration.

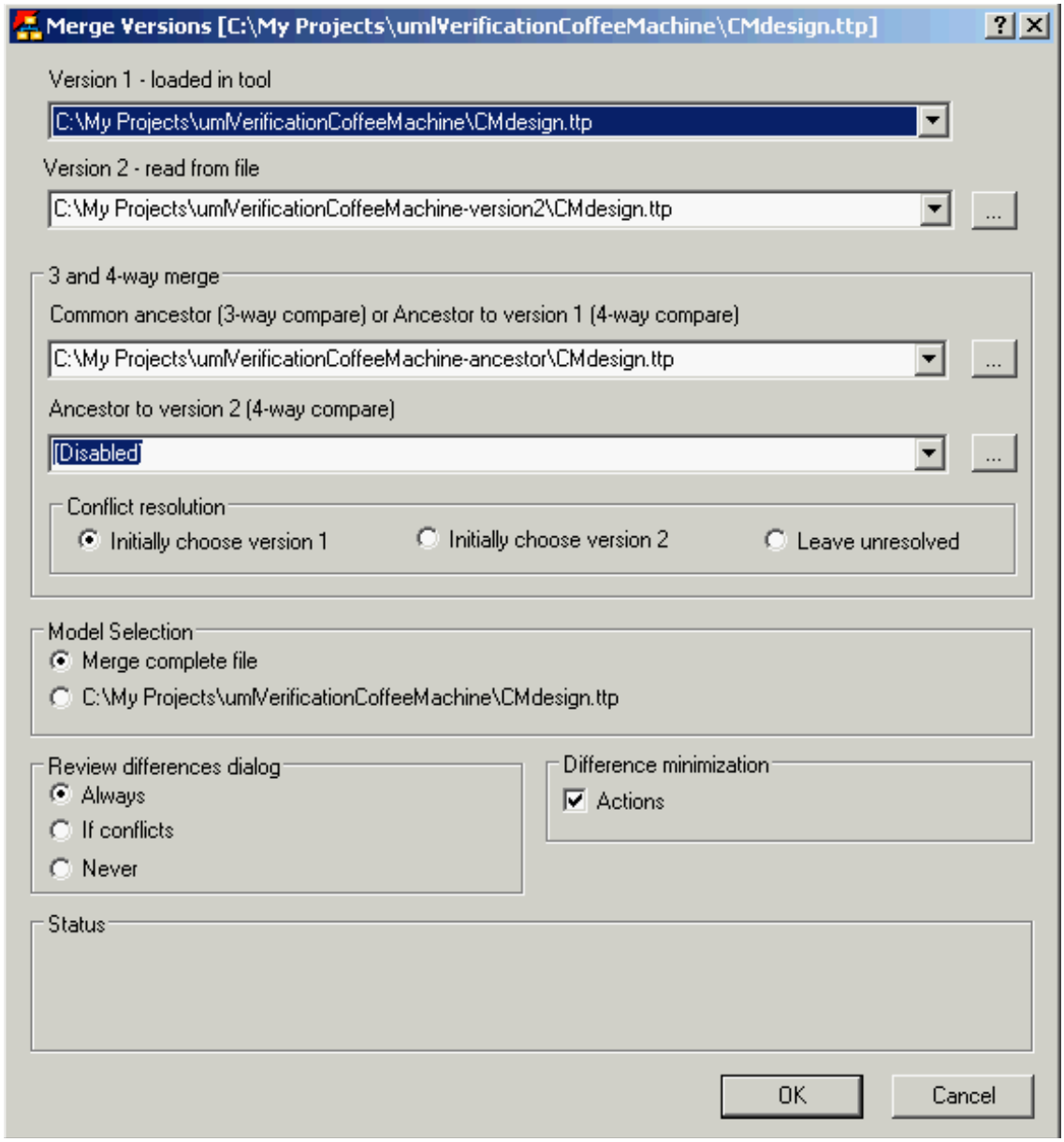



*Figure 33: Merge Versions dialog*

### Common ancestor (3-way) or Ancestor to version 1 (4-way)

For a **3-way** merge this field is used to select a file name (`.u2` or `.ttp`) containing (closest) **Common ancestor**. This version must be the common origin of both Version 2 and the version currently loaded.

For a **4-way** merge this field is used to select a file name (`.u2` or `.ttp`) containing the closest ancestor to the version currently loaded. This should be the version that was the result of the latest merge operation with **ancestor to version 2**.

### Ancestor to version 2 (4-way)

A 4-way merge is done with a previous merge as reference. For a 4-way merge the field "Ancestor to version 2" is used to select a file name (`.u2` or `.ttp`) containing the closest ancestor from this previous merge to the version selected in the "Version 2 - read from file" field. This would normally be one of the versions used in the previous merge operation that resulted in **ancestor to version 1**.

**Note**

*4-way merge: Revert to the version represented by **ancestor to version 2** is done by first performing **Accept 2** for the property and then **Reject 2**. This will revert the property to the status for **ancestor to version 2**.*

### Model Selection

- Merge complete file (default): This will apply the merge operation on the complete model. This is recommended in most cases.
- Merge only selected: This will apply the merge operation only on the selection made in the diagram or the Workspace window.

## Review differences dialog

See corresponding section under "Compare versions" on page 143 for a description of these options.

### Conflict resolution

This option is enabled when 3-way or 4-way merge is performed. It decides what should happen with conflicts not automatically merged by the merge operation:

- Initially choose version 1: selecting this option the merge operation will use the properties of Version 1, the version currently loaded, to create the resulting version.
- Initially choose version 2: selecting this option the merge operation will use the properties of Version 2, the version given as argument in "Version 2 - read from file" field, to create the resulting version.

- Leave unresolved: selecting this option the merge operation will leave the conflicts in an unresolved state. The merge operation can not be finished until all unresolved conflicts have been resolved by choosing the wanted version explicitly. The merge operation will use the properties of ancestor to Version 1, the version given as argument in "Common ancestor (3-way) or Ancestor to version 1 (4-way)" field, to create the temporary resulting version.

If there is a difference which is not desired then it can be rejected after the merge has been performed.

**Note**

*This option does not have any impact on non-conflicting differences. This means that non-conflicting differences of Version 1 and non-conflicting differences of Version 2 will be included to the resulting version anyway.*

## Command line usage

It is possible to launch the **Compare** or **Merge** dialog from the command line via the Tcl scripts `u2compare.tcl` and `u2merge.tcl` found in the `etc` directory in the Tau installation. The `u2compare.tcl` and `u2merge.tcl` scripts support two modes of operation:

- Single file mode, for operations on model files (.u2)
- Project mode, for operations on Project files (.ttp)

**Note**

*In the call to Tau (`VCS.EXE` on Windows or `tau` on `UNIX`) the tcl files must be specified with full path. I.e. on Windows the call should be*

```
VCS.EXE -script "C:\Program Files\Telel-
ogic\TAU_4.2\etc\u2compare.tcl" ...
```

### Single File Mode

```
<Tau> -script {u2compare.tcl | u2merge.tcl}
        {forceVersion1 | forceVersion2 | leaveUnresolved}
        {reviewDifferencesNever | reviewDifferencesAlways |
reviewDifferencesIfConflicts}
        {suppressSetupNever | suppressSetup}
        {true | false}
        [<version1>.ttp | <version1>.ttw]
        <version1>.u2 <version2>.u2
        [<ancestor1>.u2 [<ancestor2>.u2]]
```

### Project Mode

```
<Tau> -script {u2compare.tcl | u2merge.tcl}
        {forceVersion1 | forceVersion2 | leaveUnresolved}
```

```
        {reviewDifferencesNever | reviewDifferencesAlways |
reviewDifferencesIfConflicts}
        {suppressSetupNever | suppressSetup}
        {true | false}
        {<version1>.ttp | <version1>.ttw}
        <version1>.ttp <version2>.ttp
        [<ancestor1>.ttp [<ancestor2>.ttp]]
```

The tables below explains the attributes used in the commands.

| ForceVersion | Description |
|---|---|
| `forceVersion1` | initially choose version 1 in case of conflict. |
| `forceVersion2` | initially choose version 2 in case of conflict. |
| `leaveUnresolved` | leave conflicts unresolved. |

| ReviewDifferences | Description |
|---|---|
| `reviewDifferencesNever` | never show review differences dialog. |
| `reviewDifferencesAlways` | always show review differences dialog. |
| `reviewDifferencesIfConflicts` | show review differences only in the case of conflicts. |

| SuppressSetup | Description |
|---|---|
| `suppressSetupNever` | do not suppress the Compare/Merge setup dialog. |
| `suppressSetup` | suppress the Compare/Merge setup dialog. |

| ExitOnSuccess | Description |
|---|---|
| `true` | close Tau after successful operation |
| `false` | do not close Tau after operation |

**Example 9: Single File merge with one ancestor file on Windows**

```
VCS.EXE -script "C:\Program
Files\Telelogic\TAU_4.2\etc\u2merge.tcl" forceVersion1
reviewDifferencesAlways suppressSetupNever false
C:/work/version1/project.ttp C:/work/version1/file.u2
C:/work/version2/file.u2 C:/work/ancestor/file.u2
```

**Example 10: Project merge with two ancestors and relative paths on Windows —**

```
VCS.EXE -script "C:\Program
Files\Telelogic\TAU_4.2\etc\u2merge.tcl" forceVersion1
reviewDifferencesAlways suppressSetupNever false
version1/project.ttp version1/project.ttp
version2/project.ttp ancestor1/project.ttp
ancestor2/project.ttp
```

**Example 11: Single file compare with relative paths on UNIX ————————**

```
tau -script ${TAU_HOME}/etc/u2compare.tcl forceVersion1
reviewDifferencesAlways suppressSetupNever false
version1/file.u2 version2/file.u2
```

# Review differences dialog

## Output from compare/merge operation

When you have entered your selections the result will be shown either in Output window or in a Review differences dialog (see Figure 34 on page 151). When the result is only a summary of the compared information there will only be a summary (messages in the Output window) displaying the result from the operation. The options setting for "Conflict resolution" will together with the status of the version differences affect the appearance of the "Review differences dialog".

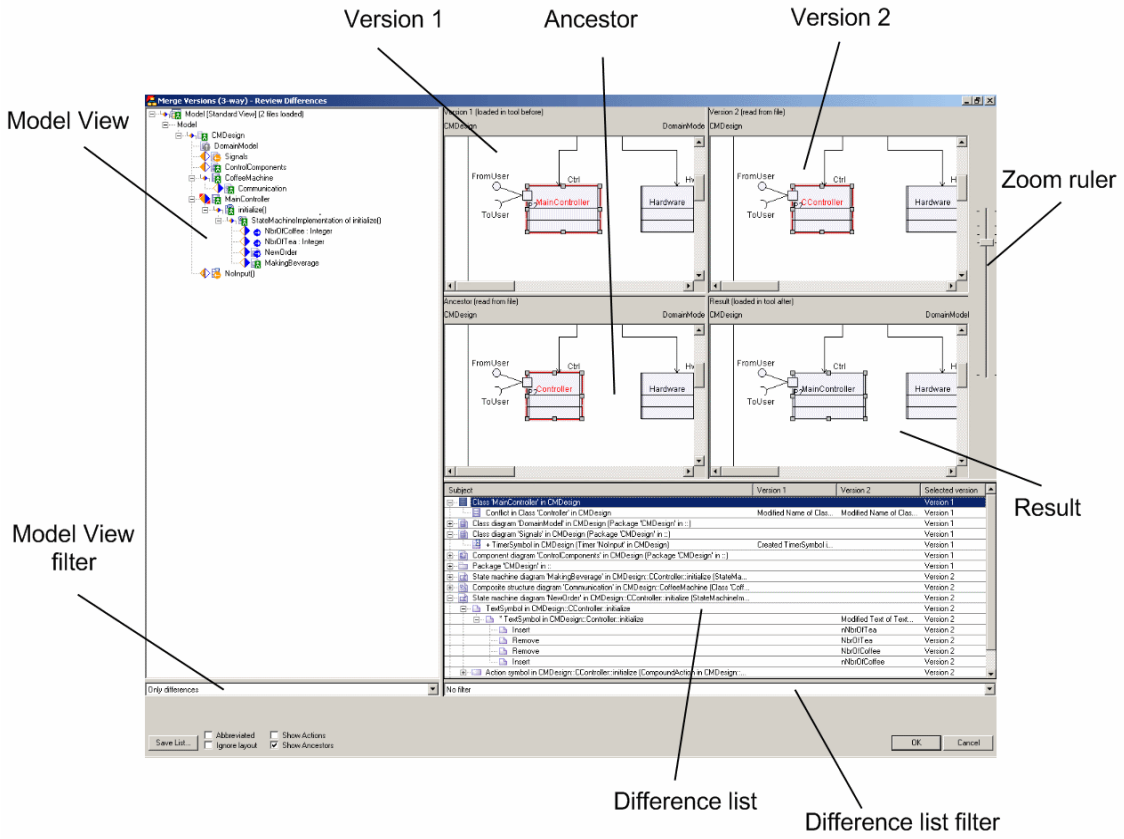



*Figure 34: Review Difference dialog*

## Model view

On the left side of the review difference dialog a model view displays all the differences in the model. Special icons provide additional information such as version where changes have been done, found conflicts and a current merge state for those changes. The meanings of the icons are listed below.

| Icon | Icon Meaning |
|---|---|
| | Model element has been added in Version 1 |
| | Model Element has been added in Version 2 |
| | Model element has been deleted in Version 1 |

| Icon | Icon Meaning |
|---|---|
|  | Model element has been deleted in Version 2 |
|  | Model element has been modified in Version 1 |
|  | Model element has been modified in Version 2 |
|  | Model element has been modified in both version, no conflict. |
|  | Conflict, model element has been deleted in Version 1 and modified in Version 2 |
|  | Conflict, model element has been modified in Version 1 and deleted in Version 2 |
|  | Conflict, model element has modified in both version |
|  | Model element does not contain direct differences, but has differences in children |

| Icon | Icon Meaning |
|---|---|
|  | Unresolved conflict |
|  | Version 1 is accepted |
|  | Version 2 is accepted |
|  | Modifications in Version 1 and/or Version 2 are rejected |
|  | Modifications in both versions are accepted |

### Model view filter

A filter can be applied on the model view. The following options are available:

- **No filter:** Everything in the model is displayed.
- **Only differences:** All entities affected by one or more differences are displayed.
- **Only conflicts:** All entities affected by one or more conflicts are displayed.
- **Only unresolved:** All entities affected by one or more unresolved conflicts are displayed.

### Version 1

This window displays a presentation of the currently selected **Difference** item or composite difference, with the properties of the version that was loaded in the tool before applying the compare (merge) operation.

### Result

This window displays the properties of the version that will be loaded in the tool after a merge operation.

### Version 2

This window displays a presentation of the currently selected **Difference** item or composite difference, with the properties of the version given as argument in the "Version 2 - read from file" field.

### Zoom ruler

The zoom ruler to the right of the graphic windows allows simultaneous zoom of the three displayed presentations.

### Difference list

The difference list displays all the differences for the entity selected in the model view and its children. See section "Difference Grouping" on page 157 for a description of the nodes visible in the difference list.

### Difference list columns

- **Subject**: This column contains an icon and an identifier for any item or group of items that have been identified to represent a composite difference between the two compared versions.
- **Version 1**: This column contains the version information on composite differences (number of differences in the group) or the items (item specific property) belonging to Version 1.
- **Version 2**: This column contains the version information on composite differences (number of differences in the group) or the items (item specific property) belonging to Version 2.
- **Selected version** (merge only): This column shows the version for the difference that currently is included in the merge Result.

### Difference list filter

A filter can be applied on the difference list. The following possibilities are available:

- **No filter:** All differences are displayed.
- **Only conflicts:** Only conflicts are displayed.
- **Only unresolved:** Only unresolved conflicts are displayed.

### Context Menu

By right-clicking an entity in the model view or an item in the difference list a context menu is opened (see Figure 35 on page 154) with the following operations available:

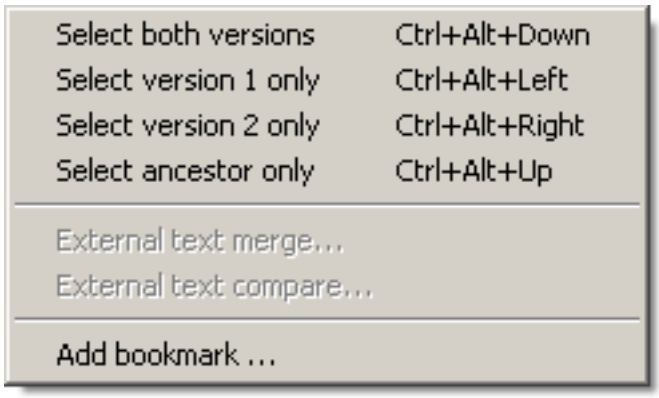



*Figure 35: Context Menu*

### Select both versions

Differences from both versions are merged into the model. If a conflict can be consolidated the merged result will be entered into the model. If the conflict cannot be consolidated then it will be resolved according to the option "Conflict resolution" and a corresponding difference(s) will be merged into the model. This operation is done hierarchically on the selected entity and all its children. If the changes have been done in one version only or the changes from both versions have been already accepted, then this menu item is disabled.

### Select version 1 only

Only differences from version 1 are merged into the model. If there are differences from version 2 currently in the model the ancestor version will be entered instead. This operation is done hierarchically on the selected entity and all its children. If there is no changes in the version 1 or the changes have been already accepted, then this menu item is disabled.

**Select version 2 only**

Only differences from version 2 are merged into the model. If there are differences from version 1 currently in the model the ancestor version will be entered instead. This operation is done hierarchically on the selected entity and all its children. If there is no changes in the version 2 or the changes have been already accepted, then this menu item is disabled.

**Select ancestor only**

Differences from version 1 and version 2 will be rejected from the model. This operation is done hierarchically on the selected entity and all its children. If the changes have been already rejected, then this menu item is disabled.

**External text compare/External text merge**

An external textual compare and merge tools can be used for comparing and/or merging comments, text symbols, task symbols and instance expressions. The “External text compare...” and “External text merge...” operations are available where applicable.

If an external textual merge is done, the result will be checked if it can be re-entered into the model. If it cannot be entered into the model, the result file from the external tool is saved and the path is reported together with an error message box.

Path and command line switches for the external text compare/merge tool are available via the Tools menu, Options dialog, under the Compare/Merge tab.

**Add bookmark**

A bookmark can be added on a selected entity. A comment can be added to the bookmark. The bookmarks can later be listed in the model navigator.

### Save List

This will save the difference list in XML format to an external file. To view the contents of that file in a more readable format, the XML file should be used together with an XSL style sheet file. In the installation there are two examples of such a style sheet. The files are called u2compare.xsl (difference centric style) and u2compare_diagrams.xsl (diagram centric style) and are found in the etc directory in the installation.

To view the XML file, copy the XSL file to the same directory as the XML file and give it the same name as the saved XML file (but with the file extension xsl). When that is done, it is possible to open the XML file in a web browser and the contents will be readable.

The size of the generated images and whether unparsed text should be saved can be modified with the options for save review information via the Tools menu, Options dialog, under the Compare/Merge tab.

### Abbreviated

The "Abbreviated" check box can be used to reduce the verbosity of the difference description text.

### Ignore layout

The **Ignore layout** can be used to temporarily hide differences having to do with layout only (position, size and line segment points changes). These differences are not removed from the list they are simply hidden and are taken into account even while hidden.

### Show Actions

With this option set the model differences will be shown rather than textual differences only. This concerns statements in action symbols, text symbols and text diagrams.

### Show Ancestors

With this option set the ancestor version(s) will be displayed as well.

### Cancel

The merge/the result will enter in Version 1, which is loaded in the tool.

## Difference Grouping

The model centric information view is used to group changes into sets of differences. Each model element has own set of properties. All differences related to those properties are added into the same group even if modifications have been done in different versions. The separate groups are created for differences in semantic and presentation models, see Figure 36 on page 157. Each group has a representative element. The representative element for top-level groups is one of those elements which are visible in the model view, i.e. “Definition”, “Implementation” or “Diagram” in terms of UML meta-model. The following rules are applied while doing the grouping:

- Difference related to created or deleted entity is added into the group where representative element is an owner of created or deleted entity, see Figure 37 on page 158.
- Differences related to moved entity are added into two groups. The “Moved (from)” difference is added into the group where representative element is an owner in ancestor version, i.e. the “old” owner. The “Moved (to)” difference is added into the group where representative element is an owner in modified version, i.e. the “new” owner, see Figure 38 on page 158.
- Differences related to modified attributes are added into the group where representative element is the corresponding modified model element, see Figure 39 on page 158.

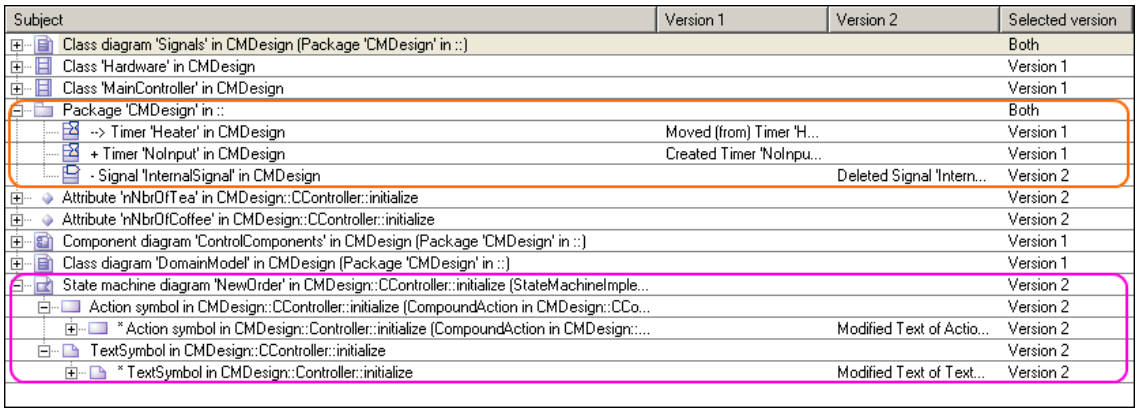



*Figure 36: Semantic and presentation model differences grouping*

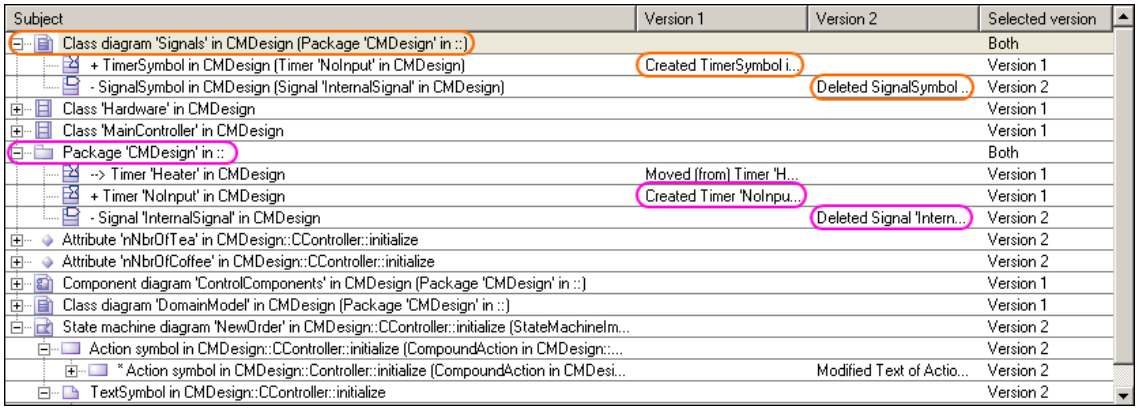



*Figure 37: Grouping of "created entity" and "deleted entity" differences*

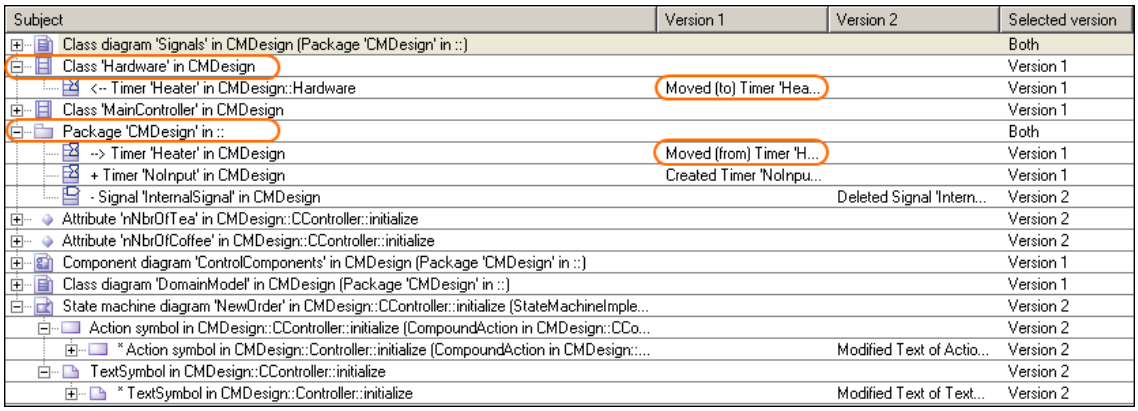



*Figure 38: Grouping of "moved entity" differences*

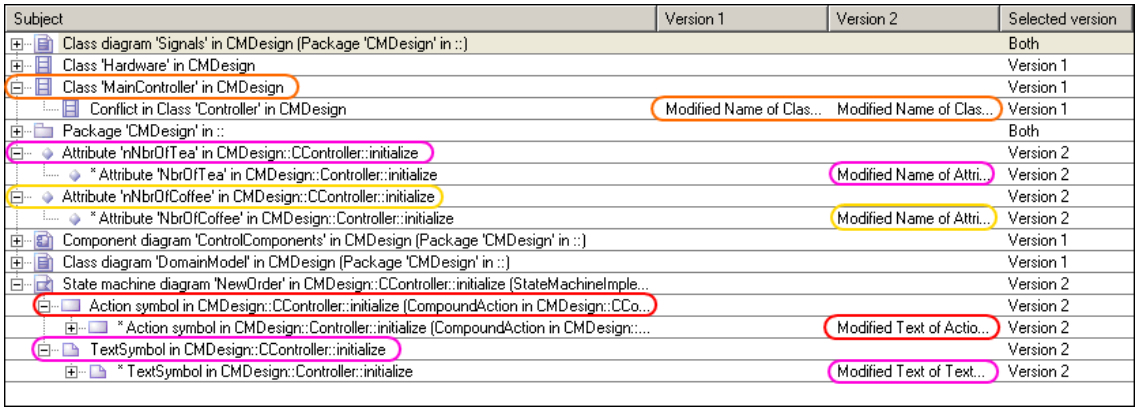



*Figure 39: Grouping of "modified attributes" differences*

In the Difference list each group is represented by a set of sub-nodes. There are five main kinds of nodes (see Figure 40 on page 159):

### Composite Group Node

This composite node contains the set of differences directly or indirectly owned by the representative element. This group can contain all other kinds of nodes.

### Composite Conflict Group Node

This composite node contains conflicting differences which are owned by different representative elements. For example, if entity has been moved in Version 1 and in Version 2 and the new owners of that entity are different in Version 1 and Version 2, then the Composite Conflict Group will be created. This group can contain Difference Nodes only.

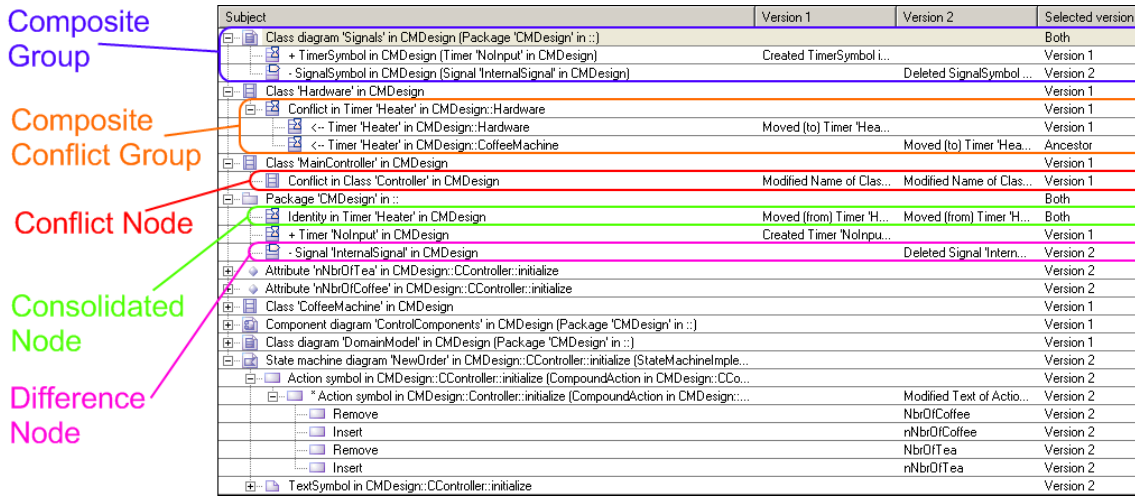



*Figure 40: Nodes in Difference list*

### Conflict Node

This node corresponds to conflicting differences which are related to the same representative element.

### Consolidated Node

This node corresponds to consolidated differences which are related to the same representative element.

### Difference Node

This node describes the simple change that has been made in Version 1 or in Version 2.

By using the Context Menu it is possible to accept or reject any individual difference as well as a group of differences. The merge tool sets up additional relations between differences. Thus the accepting of some individual difference may automatically lead to accepting or rejecting another difference(s). Such relations are used in order to preserve semantic consistency of the merged models.

## Textual merge

The merge tool uses two different strategies for merging, model-based merge and textual merge. The purpose of the textual merge is to enable merging within text symbols where model-based merge is not sufficient. Both model-based merge and textual merge is built into the merge tool and uses the same graphical interface. This means that the list of differences in the Review differences dialog could be model-based differences or textual differences.

### Dynamic differences

Some symbols, for example action symbols and text symbols can contain model elements, an example of this is a textual description of a class. Thus the text within these symbols can change during a merge operation depending on the versions of other differences in the model that are chosen. This leads to that a textual merge must be done dynamically during a model merge. The merge tool detects the differences in the symbols that act as a text container. Two additional nodes are used while representing the textual differences (see Figure 41 on page 161):

### Composite Textual Difference Node

This group node corresponds to a group of primitive textual differences. This group can contain Textual Difference Nodes only.

### Textual Difference Node

This node corresponds to a primitive textual difference. There are two operations that represent modifications in the text: **Remove** and **Insert**. **Remove** means that a part of the text has been deleted (in comparison with the ancestor version). **Insert** means that a new text has been added.

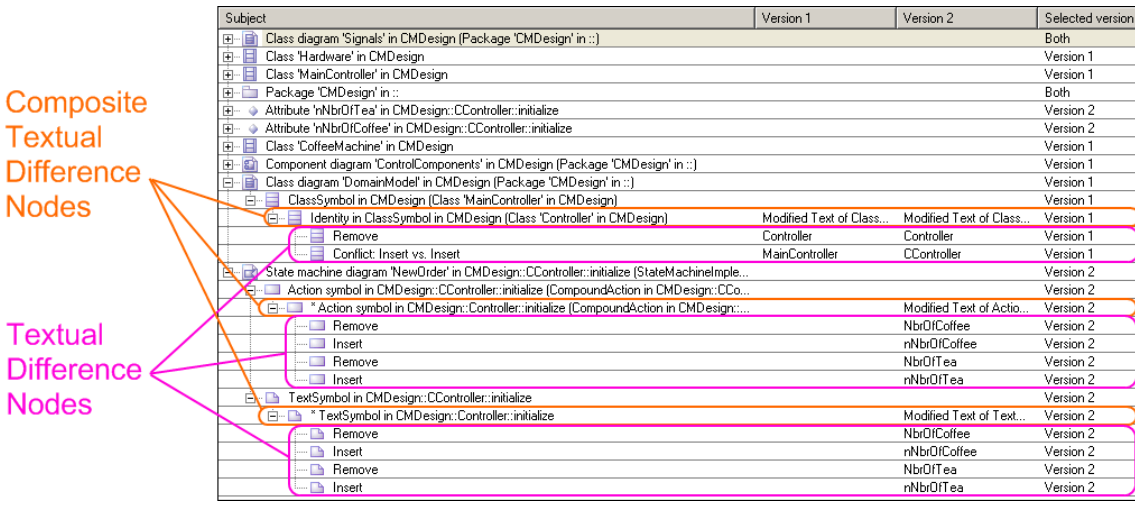



*Figure 41: Textual difference nodes*

### Selecting versions

As well as for the semantic and presentation model differences it is possible to accept or reject the group of textual differences or an individual textual difference. The merge tool sets up relations between textual differences and semantic model differences. As soon as textual difference is accepted/rejected the corresponding semantic difference is accepted/rejected. And vice versa, as soon as semantic difference is accepted/rejected, the corresponding textual difference is accepted/rejected.

When Textual Difference Node is selected in Difference list, the modified part of code is highlighted in the one or several windows which represent Ancestor, Version 1, Version 2 and Result models, see Figure 42 on page 162.

If the selected node corresponds to **Remove** operation, then the removed part of the text is selected in Ancestor window (and in Result window, if that operation is rejected). If the selected node corresponds to **Insert** operation, then the inserted part of the text is selected in Version 1 (and/or Version 2) window (and in the Result window, if that operation is accepted).

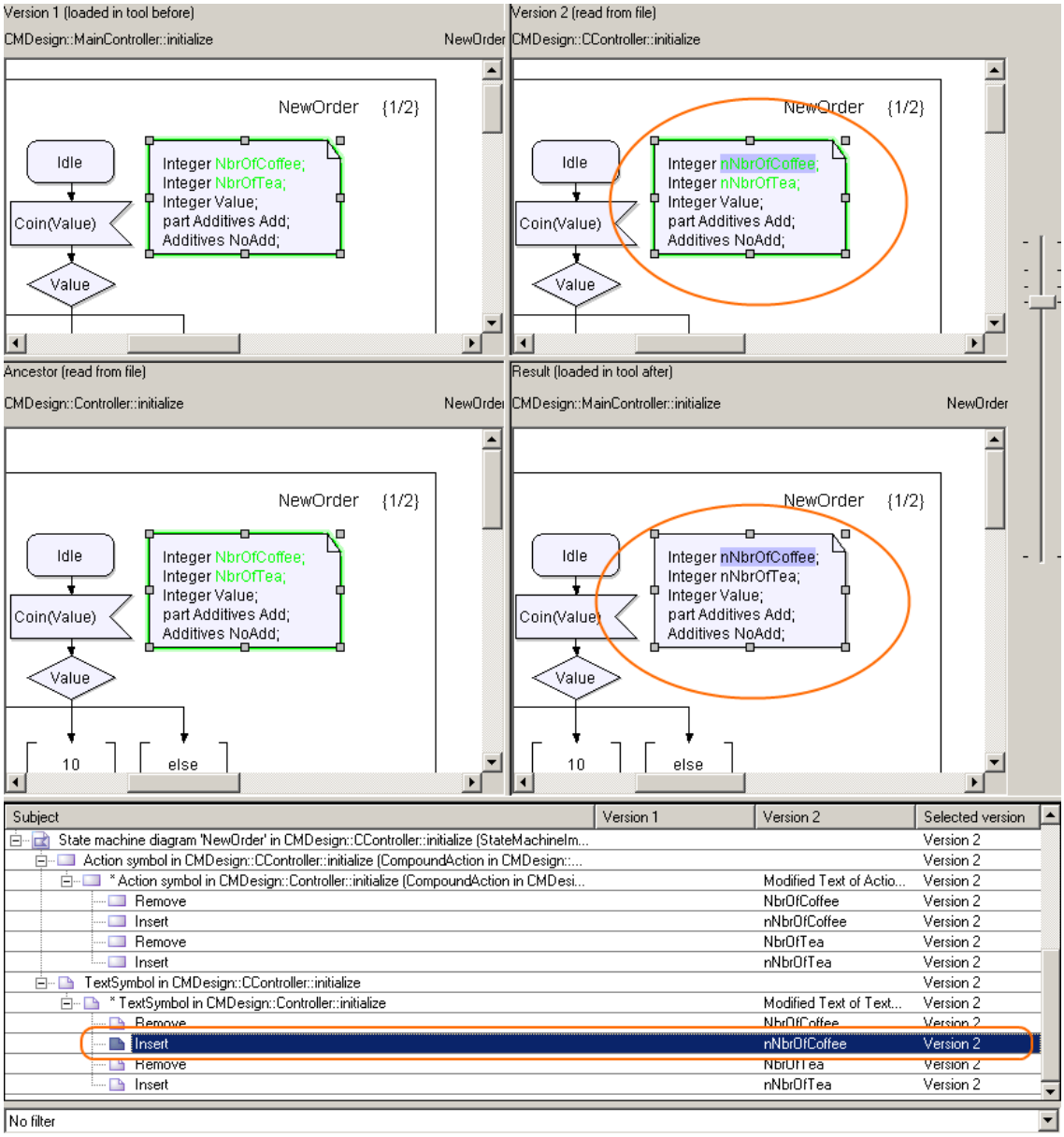



*Figure 42: Textual difference highlighting*

# Coloring

The colors are used in the Review differences dialog in order to simplify the understanding of changes that have been done in both versions. The **blue** color is used to mark presentation elements that correspond to model elements modified in version 1 only. The **green** color is used to mark presentation elements corresponding to model elements that have been modified in version 2 only. And presentation elements that correspond to model elements

modified in both versions are marked by **red** color. The same coloring is applied to modified presentation model elements, i.e. Symbols and Lines as well as to parts of the text.

# Basic Models to Get Started

When working with models it is necessary that your model has a certain level of completion for it to be executable with the Model Verifier. This level of completion may vary depending on your requirements and will not be entirely covered by the check functionality. The check functionality covers the syntax and semantic check from a model perspective, but not from a code generation perspective. The model must pass the check function, but there may be further restrictions that prevent code generation.

This section aims to explain some basic requirements to achieve completion with respect to generating code for a Model Verifier or a C Code Application. These are similar and build on the same mapping tables between UML and application code.

**See also**

Chapter 85, Setting Up the Tool Environment, for more information on how to create a new workspace with a project.

"UML Language Guide" on page 199 in Chapter 8, *UML Language Guide*

## Initial design

### Active classes and behavior

In most cases a UML design starts with defining a package as a container. For any system to be meaningful to run as a Model Verifier or application it has to have an active class. The dynamic behavior is normally modeled in a State Machine implementation, belonging to a State Machine model element (usually named **initialize** or named after the active class it belongs to). These encapsulating elements will be produced when creating a new State machine diagram. One of the fundamental stages in your workflow is to determine the top-level active class for your system.

### Model example

In order to create a Model Verifier it is required that there is a Build Artifact. You are prompted to specify build type and Build Root for this artifact if it does not exist. An executable model (Figure 43 on page 164) can be made up of:

- a package (optional)
- a class diagram (optional)
- at least one active class
- at least one State Machine, with a State Machine implementation (optional, in the sense that it can be implicit)
- at least one build artifact

The design of your model can be made in the Workspace window directly in the model, using the shortcut menu and creating element with the **New** submenu. This most natural way to start is however by creating a class diagram in the package.

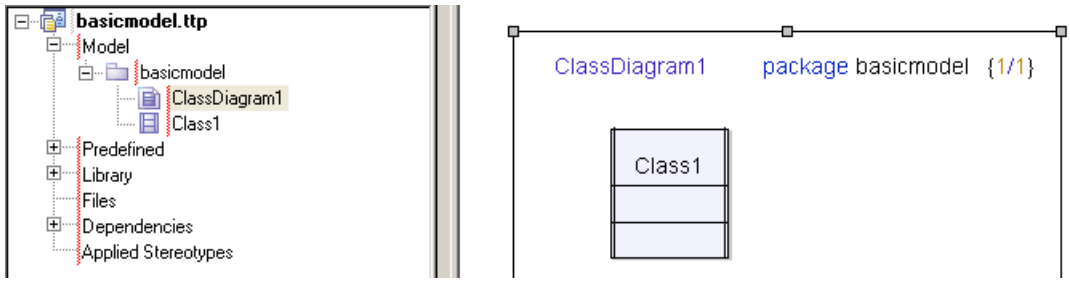



*Figure 43: Model with an active class*

Running the Model Verifier on the example above there will be an implicit State machine generated, with no actual behavior.

### Signalling example

An active class has to have a port to have any form of communication based on UML signals and interfaces. The port is defined with the signals or interfaces that it can transmit. A model with (internal or external) communication (Figure 44 on page 165) will have:

- at least one signal (which is used in the state machine defining the dynamic behavior)
- at least one port

- an interface (optional)

The use of interfaces to encapsulate signals is also a fundamental stage in the workflow when designing your UML model.

### State machine

This will be possible to generate code for, but there will be an implicit state machine that will not have any behavior that can be simulated.

Adding a simple state machine to the model makes the behavior more visible (Figure 45 on page 166).

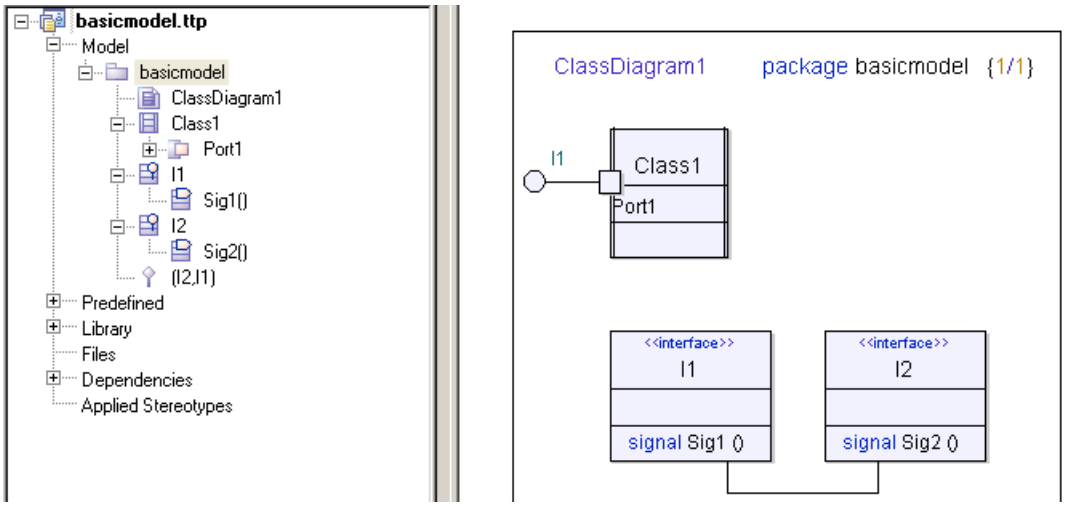



*Figure 44: Model with interfaces and signals*

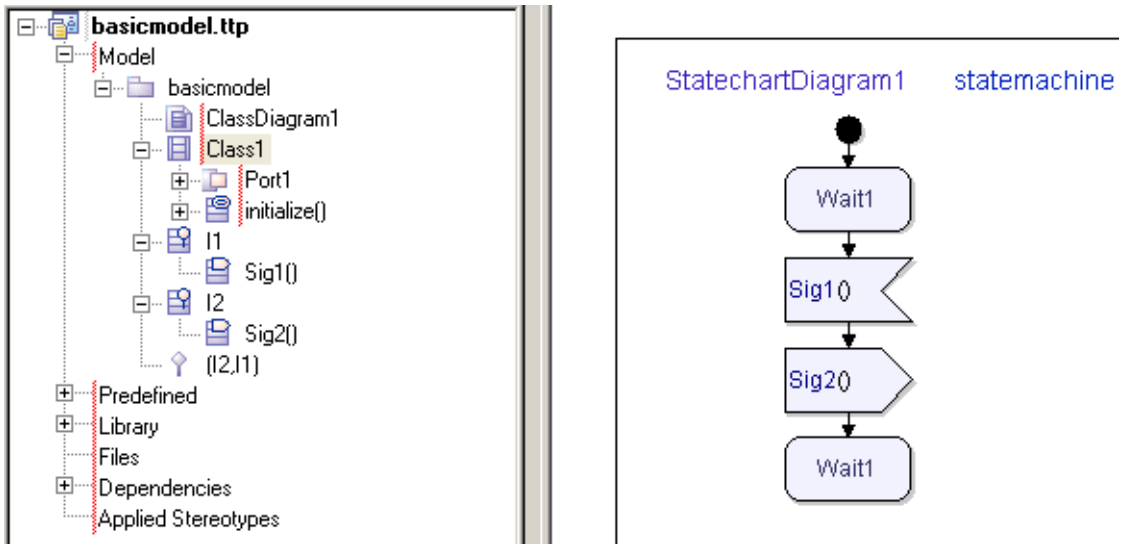



*Figure 45: State machine with signal*

# Internal communication

## Open and closed systems

The model in Figure 44 on page 165 is describing an open system. An open system is here referring to a system that interacts with its environment. A closed system may be designed from an open system where the interaction is built into the system.

The example will now be enhanced with another active class (Class2) which will be having Class1 as a part. A new interface (I3) and signal (Sig3) is added, Figure 46 on page 167.

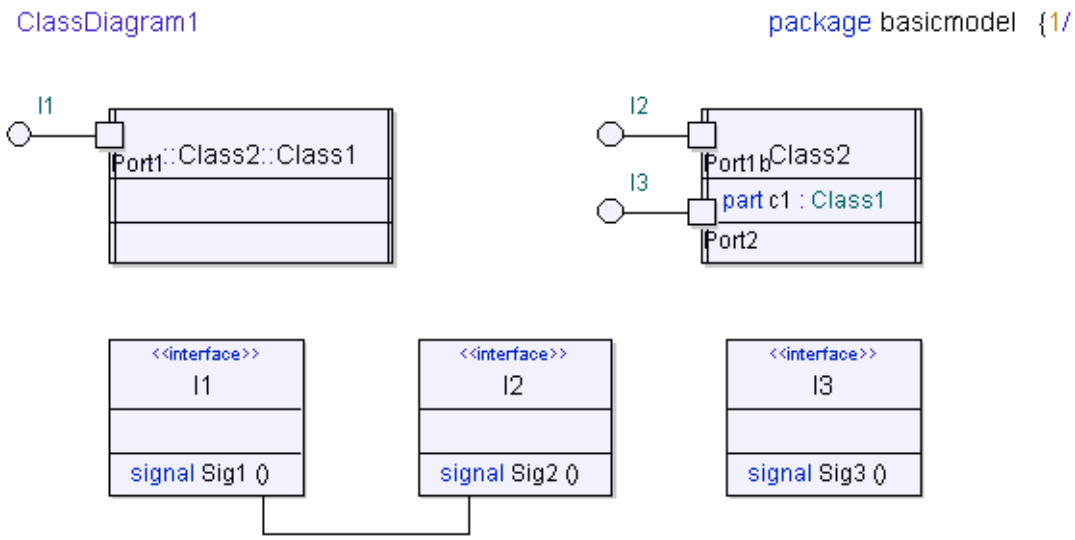



*Figure 46: Class2 and new interface*

## Ports and behavior ports

The port in the example is used for communication with the environment. Ports will also be used for communication with other parts of the model. In a larger design there can be an active class which in turn contains active classes. A behavior port will allow the design to show the communication between a part and the state machine in (an instance of) the class that owns the part.

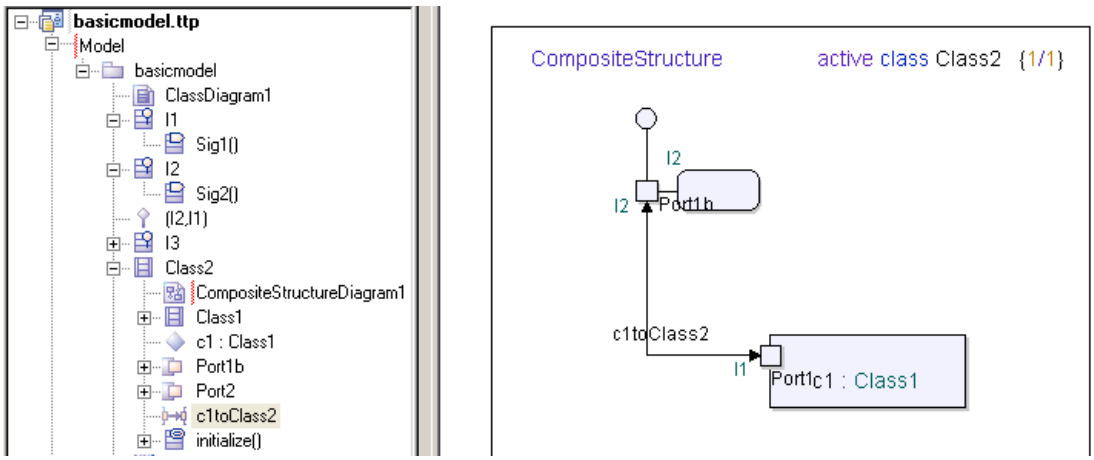



*Figure 47: Behavior port in composite structure diagram*

### Architecture

Composite structure diagrams are necessary only when there is an ambiguity in how signals can be transmitted in a model. Composite structure diagrams can be of great use to explicitly show how a complex model with active classes is structured. If the top class has a state machine that requires communication with its parts this can be modeled in a composite structure diagram with a behavior port, Figure 47 on page 167.

The state machine for Class2 is shown in Figure 48 on page 168.

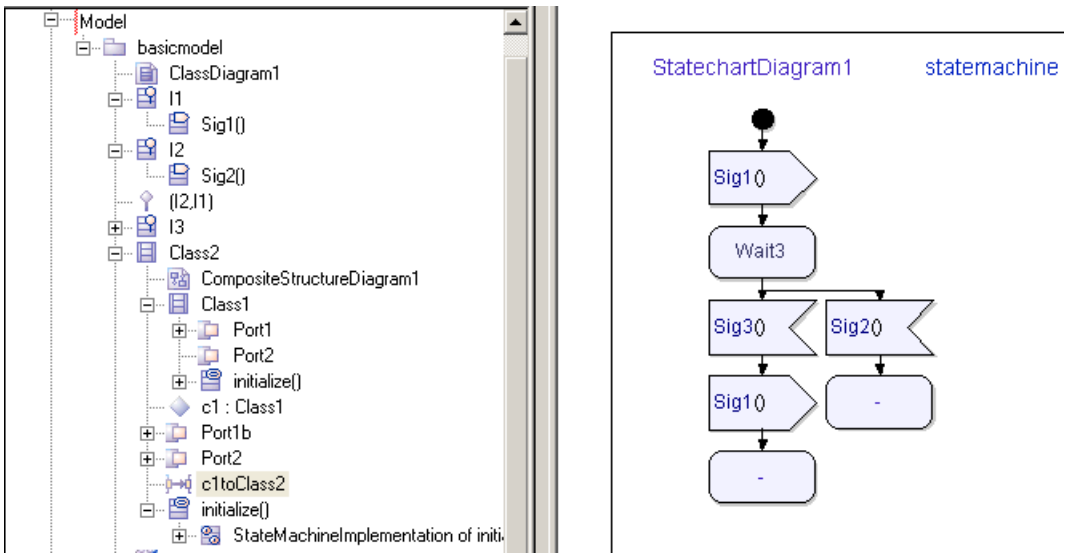



*Figure 48: State machine for Class2*

# 7

# *Working with Diagrams*

When you have opened a project you are ready to edit your model.

When editing the model, you are provided the best view if the Model View tab in your Workspace window is active. Here you can easily see how the changes you do in diagrams affect the model.

How you should use the diagram editors in combination with the model information is highly dependent on the application that you want to create. The recommendations below are meant to help you getting familiar with the tool and later you can adapt and change the workflow to fit your needs.

# Common Diagram Operations

Common diagram operations include:

- Create diagrams
- Open, save and print diagrams
- Move diagrams
- Resize diagrams
- Save in New File
- Find
- Text parsing
- Diagram auto layout
- Organizing the view

### Grid

The diagram drawing area will always have an active grid, which means that there will be a **snap to grid** which is always set to **on**. The grid spacing is set to 2 millimeters and cannot be changed. It is possible to show or hide the grid from the shortcut menu or by changing in the Options, default is hidden. All symbols, lines and text fields (except those fixed in symbols) will adhere to the grid. When a symbol is resized, it is done in grid steps. This also applies to auto resizing.

### Frame

All diagrams will have a frame enclosing all symbols, except port symbols which can be placed on the frame. The upper left corner of the frame symbol will be at (x=10, y=10) millimeters, measured from the canvas top left corner. The frame is sized according to the diagram size and layout. The position can be overruled if extra space is needed for symbols placed on or outside the frame.

The frame may be resized and moved in all directions, canvas space permitting.

### Heading

The diagram heading is positioned in the top left corner of the diagram. The text is calculated from the properties of the defining entity.

### Diagram Name

The diagram name is positioned in the top right corner of the diagram. The text is right aligned. The information is calculated from the model information, and thus only editable through the Model view.

## Create diagrams

A diagram contains a set of presentation elements representing elements from your model. Diagrams are managed through the Workspace window.

1. In the **Workspace** window, click the **Model View** tab.
2. Select or create a suitable package.
3. You are now able to choose **New** from the shortcut menu to create an appropriate diagram (or model element).

**See also**

"Create Presentation" on page 116 in Chapter 6, *Working with Models*

## Open, save and print diagrams

Information in the diagram is saved in a file when you save the project. If you do not specify a file, the information will reside in the data file (.u2 extension). This file will normally reside together with the project (`.ttp`) and workspace (`.ttw`) file you are currently using.

- To open a diagram: double-click the diagram icon in the **Model** View.
- To print an open diagram: on the **File** menu, click **Print...**

The diagram size when creating a new diagram is derived from the printer settings. For example, if the page layout is set to landscape on the printer, diagrams will have a landscape orientation.

### Save diagram image

A diagram can be exported as an image in a range of formats including JPEG, GIF, BMP and SVG.

A diagram can be saved as an image by opening it and clicking **Save As...** on the **File** menu. A save dialog will open where a file path and wanted image format can be selected.

### Save in New File

An operation is provided to store a UML element in a separate file. This is provided in the Model View of the workspace window as a separate shortcut menu choice “Save in New File...”. When selected you will be prompted for a file name and the element will be stored in the indicated file.

The following elements can be saved in new files:

- Definition
- Implementation
- Class diagram

**See also**

“Select diagrams to be printed” on page 2434 in Chapter 88, *Printing*

“Save” on page 2458 in Chapter 91, *Dialog Help*

## Move diagrams

You can move diagrams within projects and between projects in the same workspace.

- Click the icon corresponding to the diagram in the Model View and drag the diagram to its desired location.

## Resize diagrams

When a diagram is created the default is that the diagram is placed in an auto size mode. This mode can be turned off or on from the diagram shortcut menu or the diagram element properties toolbar. When the diagram is in auto size mode diagram elements can be dropped, inserted, pasted, moved etc. anywhere on the editor canvas. If the element is placed outside the frame the diagram will automatically be resized to fit the element. The auto size mode will always keep the minimum full pages to contain all diagram elements.

The initial diagram size and layout is determined by the Print settings (size and orientation). If no printer is installed the size is determined from the current Options set. Resize of the diagram is done in two steps. If the diagram is to be enlarged, it can be done in one of the following ways:

- Change the size of the diagram from the shortcut menu. Right-click on the diagram in the Model view and select **Diagram size...**, this will open a dialog that allows you to control the size manually or return to auto sizing.
- When auto size is off, you can increase the diagram size in steps of whole papers by holding down CTRL and clicking on the drag handles of the frame symbol. If CTRL + SHIFT keys are pressed when clicking on the handles the diagram will be decreased in size. The mouse cursor will change appearance according to which keys are pressed.

The frame symbol is automatically resized to match the resize of the diagram. When resizing the diagram to make it smaller the frame symbol may not be made smaller than the extent of all symbols (or lines) in the diagram. The frame must be at least 1 grid point apart from any symbol (line) inside. The diagram name symbol will be moved automatically when the diagram is re-sized. Ports and lines connected to the frame move with the frame. The size of the canvas may not be made smaller than the frame + margins. Resize of the frame should follow the grid.

## Find

Open the **Find Diagrams and Definitions** dialog by choosing **Find** from the Edit menu. Through the dialog it is possible to locate definitions. To find the usage of a definition it is possible to **Find text in diagrams too**. Results are shown in tab Search result in the Output window. To list where an entity is used, right-click the entity in the Model view and from the shortcut menu click List references.

**See also**

“Model Index” on page 122.

## Text parsing

In general, C++ style syntax is used in text symbols (and external files). UML style syntax is used in all other symbols. However, the parsers accept some minor deviation from UML (“public” instead of “+”, “output” instead of “^”, etc.) but will unparse everything in UML style. In text symbols, UML style deviations are accepted, but will be converted to C++ style (i.e. “+” as visibility operator is converted to “public”).

These changes fall into this category:

- Visibility: `+` converts to `public`
- Visibility: `#` converts to `protected`
- Visibility: `-` converts to `private`
- Signal Sending: `^` converts to `output`
- Decision alternative: `else` converts to `default`

There are some other properties of the unparse phase:

- "`in`" parameter direction kind is not unparsed
- quotation marks are removed from names that do not need it (for example `'Name1'` becomes `Name1` after unparse)
- "`in / out`" parameter direction kind is unparsed as "`inout`"
- data types only with literals become enumerated data types, i.e. `datatype colors { literals red, green; }` un-parsed as `enum colors { red, green }`

The unparser will expand shortcut notations. Several attributes, remote variables, signals, timers, exceptions or synonyms defined at once (for example, `Integer i, j, k;`) will be unparsed as several separate definitions (`Integer i; Integer j; Integer k;`).

The unparsed adds omitted parenthesis in signal and timer definitions (i.e. "`timer T`" becomes "`timer T()`").

Open ranges (if possible) are converted to UML style: "`>= n`" is converted to "`n..*`" and "`>=0`" is converted to "0..*".

### Auto-quote

The purpose of auto-quote is to assist you with typing quotation marks. If you add a whitespace in a name there should be quotation marks added wrapping the text. Only a limited set of symbols (labels) are auto-quoted. Typically labels that contains names are auto-quoted.

### Word wrapping

Word wrapping allows words to be broken up into several lines. This is applied to multiline labels and to non-autosized symbols. The algorithm for determining where to break a word looks for any of the following: '`:`', "`::`", whitespace, capital letter, comma ('`,`'), period ('`.`'), underscore.

## Diagram auto layout

The shortcut menu for diagrams (canvas background) includes an **Automatic layout** menu item for diagram types that has an auto layout algorithm associated with it. If this menu item is selected the diagram elements will be placed in a layout suitable for that specific diagram type. For example will Class diagrams and state machine diagrams have a hierarchical layout.

The auto layout algorithms will be used when placing diagram elements with the Show elements dialog.

When using the auto layout:

- It is only Generalization lines that are included in the layout algorithm in class diagrams.
- Flow lines and transition lines are included in the layout algorithm in state machine diagrams.

## Organizing the view

The editors have several features allowing you to organize your view. These include shortcuts for scrolling and zoom in/out.

When a diagram is closed, the current scroll and zoom settings can be saved to a separate file with the extension `.u2x` and the name `<project>_DiagramSettings`. This file is not added to the project (ttp file). Instead when a project is loaded there is a step that loads files with the extension `.u2s` and a corresponding name. This feature is controlled by the option **Remember scroll and zoom**.

### Scroll

When your diagram is sized so it is not entirely visible in the desktop area it is possible to scroll the view. Scrolling your diagram view can be done with the window scroll bars.

(**Windows**) It is possible to scroll with the IntelliMouse pointing device.

- Vertical scrolling is done by use of the scroll wheel.
- Horizontal scrolling is done by pressing CTRL and at the same time use the scroll wheel.

### Zoom

It is possible to zoom in fixed steps via the Zoom command on the View menu.

It is possible to do a continuous zoom via the shortcut menu. Right-click in a diagram, point to zoom and select the desired enlargement level. When not in text edit mode it is possible to use "-" (minus) for zoom out and "+" (plus) for zoom in.

(**Windows**) It is possible to zoom with the IntelliMouse pointing device.

- Press SHIFT and use the middle mouse button, the diagram will zoom.
- Double-click on the scroll wheel will zoom to 100%.
- SHIFT + Double-click on the scroll wheel will zoom to fit the current diagram width in the desktop area.

**See also**

"Docking windows" on page 25

"Workspace Operations" on page 2372

# Common Symbol Operations

- "Model references" on page 189
- "Update model" on page 191
- "Nested symbols" on page 191


## Symbol information

Symbols and lines in diagrams will present context sensitive information on tool tips when you let the cursor rest on them. This feature can be controlled from the **Tools** menu **Options** command via the UML Advanced Editing tab.

The selection of **Show symbol and line tooltips** will allow you to see contextual model information, for example stereotype and model bind information.

The selection of **Show edit mode tooltips** will allow you to see information from syntax parsing while in text edit mode.

### Show/hide model element details toolbar

The **Show/hide model element details** toolbar lets you toggle showing and hiding of certain features of symbols in diagrams. These settings are remembered per diagram element. If the setting is applied on the diagram, all diagram elements without a setting of their own, will inherit the diagram setting.

- Show/Hide qualifiers

  Toggles the qualifying parts of a label text. For example: `Package1::Package2::Class1` will be toggled to `Class1`.
- Show/Hide stereotypes

  Toggles the stereotype label. When the label is hidden the space occupied by it is considered minimal and can thus affect the symbol size.
- Show/Hide quotation marks

  Toggles the automatic quotation marks, thus a limited set of symbols are auto-quoted and those symbols are affected by this button. The text will still be considered quoted when the quotation marks are hidden. Auto-quoted text is typically names containing a whitespace.

## Add symbols

To add a symbol, you click its corresponding icon in the Diagram element toolbar, then click or right-click in the diagram to position the symbol.

It is also possible to generate symbols from the Model View. You can in many cases drag a Model element into the desired diagram.

In state machine diagrams it is possible to Insert a symbol in the flow by pressing CTRL and then click in the diagram element toolbar on the new symbol. The symbol will be positioned after the currently selected symbol.

### Reference existing

When you right-click to position a symbol a shortcut menu will appear for most symbols. Symbols that do not have this menu are:

- Transition line (used in state machine diagrams when designing in State-oriented view).
- State machine transition symbols not related to states, signals and operations.

The shortcut menu contains the following choices: **Create New <element>**, **Leave Unbound**, **Reference existing**.

- **Create New <element>**: a new symbol will be created and a corresponding model element will be created in the model.
- **Leave Unbound**: a new symbol will be created but no corresponding model element will be created.
- **Reference existing**: will display a drop-down box with existing model elements matching type and scope.

### Auto placement

In many cases it is desired to position a symbol in connection with the previous, for example a port on a class. For this purpose auto placement of symbols is supported.

- Hold down SHIFT and click the symbol toolbar. The symbol clicked will be connected to the currently selected symbol.
- Hold down CTRL and click the symbol toolbar. The symbol clicked will be inserted between the currently selected symbol and the next one.

The symbols in the toolbar will be dimmed if they cannot be connected in a syntactically correct flow to the currently selected symbol.

It is also possible to use the SHIFT + SPACEBAR and CTRL + SPACEBAR shortcuts to get a list of the symbols that can be auto placed.

**See also**

"Name support" on page 84 in Chapter 6, *Working with Models*

"Create Presentation" on page 116 in Chapter 6, *Working with Models*

"Model navigation/creation" on page 118 in Chapter 6, *Working with Models*

"Diagram auto layout" on page 175

"Show elements" on page 179

"Creating a message" on page 232

## Show elements

The **Show Elements** dialog makes it possible to decide what model elements to show as symbols in the current diagram.

**Show Elements** is available:

- From the Tools menu
- In diagram shortcut menus.

When **Show Elements** is invoked, a dialog appears, with a list of model elements that can be shown as symbols in the current diagram. By adding or removing check marks for model elements, you can add or remove symbols for their corresponding presentation from the diagram.

The **Show Elements** dialog has the following features:

- The **All** button allows you to check all model elements in the list with one click.
- The **None** button allows you to remove all existing check marks with one click.

- The **Short list** check box makes it possible to toggle between a short and a long list of model elements.
  - The **short list** contains model elements that are natural to show in the current diagram. (For instance, a class in a class diagram.) Model elements that are already shown as symbols in the current diagram, are also included in the short list, even if they are not natural to show in the current diagram.
  - The **long list** contains all model elements from the selected scope that can be shown as symbols in the current diagram. In addition to the natural model elements for the current diagram, this list also includes more uncommon conversions. (For instance, the alternative to show a class as an actor in a use case diagram is included in the long list.)
- The **Select scope** button brings up a dialog, in which it is possible to select the scope or scopes to pick model elements from for the list in the main dialog. As default, only model elements from the local scope where the diagram resides is included in the list.

## Select symbols

Pressing CTRL while double clicking outside a text field (but within the symbol boundaries) will select the symbol, outgoing lines and all connected symbols.

Click and drag (when not clicking on a symbol or line) will create a selection rectangle. Everything that is placed within the rectangle is selected.

Click and drag while pressing CTRL (when not clicking on a symbol or line) will create a selection rectangle. Everything that is intersecting the rectangle is selected.

In a state machine flow if you do CTRL + double-click on a symbol in a flow this will select the symbol and all symbols after. This also works when the flow is branched.

## Move symbols

To move a symbol, click it and drag it to the desired location within the diagram. It is also possible to drag symbols to other diagrams.

Symbols with text fields should be selected as not to enter the text edit mode. The cursor will change appearance to indicate this.

### Moving text fields

It is possible to move some text fields (labels). This is true for labels that belongs to lines and for labels that belongs to symbols with the label outside of the symbol boundaries (port, pin, etc.).

This is done by first selecting a label, which then can be dragged in one of its handles. The new position will be saved as an offset to the default position. If you move a line or symbol with a label, then the label will also move and preserve the offset.

The offset can be removed by clicking on the shortcut command **Reset all label positions**. That will reset all labels belonging to the currently selected symbol to their default position.

Labels can be dragged to anywhere on the desktop, even outside the frame symbol and diagram area. Labels outside the diagram area will not be printed.

## Resize symbols

### To resize symbols manually

1. Select the symbol in question.
2. Place the mouse over one of the eight gray squares.
3. Drag the mouse until the symbol is the size you want it to be.

### Autosize symbols

All symbols offer the choice of **Autosize**, which adapts the symbol to the size of the text entered within it. Right-click the symbol and choose **Autosize** from the shortcut menu.

### Collapse symbol

A symbol with compartments (for example a class symbol) can be collapsed by checking the “Collapsed” menu item in the shortcut menu for that symbol. Compartments and any labels inside the compartments will not be visible when in a collapsed state.

### Resized symbol indicators

When three dots appear outside a symbol's lower right corner, the size of the symbol is too small to show all text that is available in the symbol's text fields. To resize the symbol to adapt to the text, select the symbol and double-click the dots.

## Connect symbols

### To connect symbols manually

1. Click a symbol and find its line handles.
2. Drag the line to the other symbol.
3. The end of the line will appear as a circle with crossbars when you reach the second symbol. Complete the connection by clicking inside the symbol close to the border position where you would like to attach the line.

Some connections will result in model elements. If you for example drag the generalization handle to another class in a class diagram you will end up with a line between the two classes as well as new icons in the Model View, informing you that a generalization has been added.

Symbols in a State machine diagram can be connected automatically to a flow with Auto placement.

**See also**

"Draw lines" on page 196

## Symbol flow editing

In diagrams which have a concept of flows, such as state chart diagrams and activity diagrams, there are certain operations possible to manage these flows.

### Select a flow or a branch of a flow

If you do CTRL + double-click on a symbol in a flow this will select the symbol and all symbols after the selected symbol. This also works when the flow is branched.

### Append symbols to the flow

When adding symbols in an diagram you can create a connected flow of symbols. Hold down SHIFT and click the toolbar. The symbol clicked will be connected to the currently selected symbol. The symbols in the toolbar will be dimmed if they cannot be connected in a syntactically correct flow to the currently selected symbol.

**See also**

"Auto placement" on page 178

### Insert a symbol in the flow

Insert operation is possible when CTRL is pressed and a button is clicked while the selection corresponds to one of the following cases:

- A single selected symbol (insert operation after this symbol)
- A single selected line (insert operation on this line)
- Two selected symbols with one line between them (insert operation between the symbols)

**Note**

- *Using **CTRL** + **decision symbol** is not possible. There are several possible flows out from a decision symbol and because of this it is not supported to insert a decision symbol this way.*

**See also**

"Auto placement" on page 178

### Remove a symbol from the flow

When a symbol is cut or deleted from a flow an auto-created line replaces the removed symbol and its connected lines if possible.

## Edit text fields in symbols

To be able to edit a text field in a symbol, the symbol must first be selected.

- To edit a text field in a symbol, select the symbol and click in the text field at the position where you would like to add or change. You are now able to enter your text changes. Text within guillemets, «» (for example stereotype information), cannot be edited.
- If a symbol is selected, and a double-click is done in a text field the closest text word will be selected. If the symbol is not selected, the double-click will be done on the symbol (normally navigation).
- If a symbol is selected, click and drag in a text field in the symbol will enter edit mode and select the text. If the symbol is not selected, the symbol will be moved.
- Pressing F2 will enter edit mode for the main text of the selected symbol. For a single line text all text will be selected, while for a multi line text, there will be no selection and the text insertion marker will be placed at the end of the text.

If you enter syntactically incorrect text in the symbol, it will be marked with a red indicator positioned at the first error. The text is checked continuously as you type.

**Note**

*Double clicking outside a text field (but within the symbol boundaries) will always perform the double click operation for the symbol (normally navigation).*

## Diagram element properties

A specific toolbar called Diagram Element Properties is available. This will contain drop-down boxes that control various properties of the selected symbol(s)/line(s):

- font
- font size
- symbol / line background color

The toolbar contains a button that will remove the set properties and revert to default styles.

If no symbol is selected, the toolbar commands will apply to all symbols in the current diagram, except symbols with individually applied properties.

## Handling comments

Comment symbols can be added to all symbols.

1. Click on the comment symbol on the toolbar.
2. Position the symbol in the diagram.
3. Connect the annotation line from the comment symbol to the symbol you want the comment to belong to.

### Comments and constraints

The shortcut command **Show Comments** for Signature symbols (in the sub-menu to **Show/Hide**) will create and attach a comment symbol for each comment model element owned by the signature symbol that does not already have a comment symbol in the current diagram.

The shortcut command **Show Constraints as Symbols** for Signature symbols will create and attach one constraint symbol for each constraint model element owned by the signature symbol that does not already have a symbol in the current diagram.

### Column of Remarks

Two or more comment symbols form a **Column of Remarks** when they are positioned close and aligned or almost aligned in vertical position, see Figure 49 on page 186. The vertical positions will be auto-adjusted to form a left-aligned column when a column of remarks is detected.

The column can be moved in the horizontal direction by pressing SHIFT and moving the top comment symbol a small vertical distance (less than the total width of the column). If another comment symbol in the column is moved a small distance (with SHIFT pressed), it will be repositioned back into its place in the column. Moving any comment symbol a larger distance will remove it from the column.

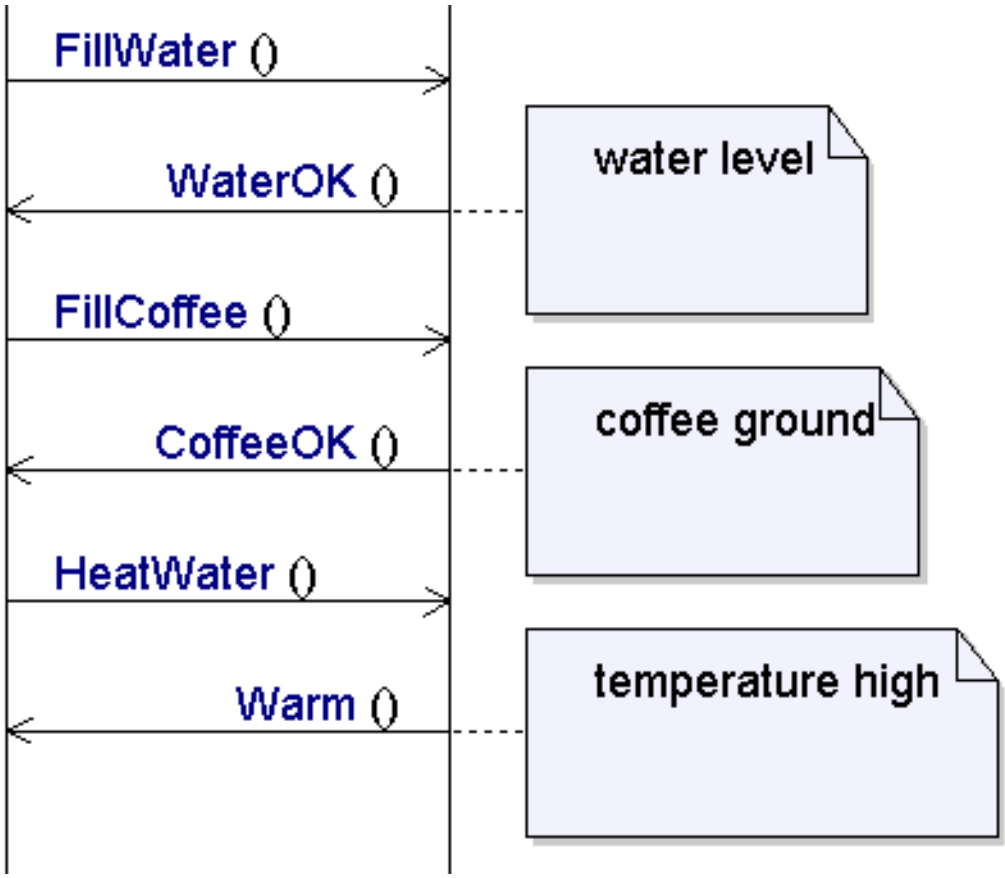



*Figure 49:Column of remarks*

**Note**

*The top comment symbol in the column of remarks must be positioned below the lifeline headers to be included in the column.*

## Copy, cut, delete or paste symbols

All symbols have a shortcut menu that you may access by right-clicking on the symbol. From this menu, choose **Cut**, **Copy** or **Paste** according to your needs.

You can also drag symbols to other tools, for example MS Word.

To delete a symbol, select it and press the **Delete** key.

**Note**

*A Delete operation may or may not affect your model depending on the type of symbol and its relation to the model. When you add symbols to your diagrams this will in most cases add information to your model. When you delete a symbol in a diagram it will only delete information in the model if there is a one-to-one relation with the symbol and the model. This is for example the case with State machine flow symbols. To delete a symbol and its corresponding model element use Delete from Model.*

# Icon

## User-specified icons

It is possible to use an image file and replace selected symbol icons with a user-specified icon. The icon can be specified on the following levels:

- for a specific symbol
- for a specific semantic model element, implying that all symbols that are associated with the model element will use the icon
- for a specific stereotype, implying that all symbols that are associated with model elements stereotyped by the symbol will use the icon
- for a specific type (for example class or datatype), implying that all symbols that are associated with instances of the type will use the icon

## Add stereotype

This feature is controlled by a stereotype. To activate this you right-click the model element that is to have the icon, and from the shortcut menu select Apply Stereotypes. In the dialog apply the stereotype `TTDStereotypeDetails::`Icon. You can also through the Properties editor open this dialog using the **Stereotypes** button.

The entities that this stereotype can be applied to is controlled by the Metamodel properties. To view this information go to the `Library` section in the Model View and open the package for `TTDStereotypeDetails`. In the class diagrams you can view the relations between the supported entities (metaclasses) and the icon stereotype.

### Ordering

If more than one user-specified icon is specified among the alternatives above, then the order is according to the list above. Thus, if an icon is specified for a specific symbol, this will be used, otherwise the icon for the model element is used and so on.

### Icon mode

For symbols identified with a user-specified icon there will be a shortcut menu choice called **Icon mode**. When selected this menu choice will cause the symbol to be visualized instead of the usual symbol.

### Image file

The icon is defined by a property of the symbol, model element or applied stereotype and can be changed using the Properties dialog for the entity. In this dialog select **Icon** in the Filter drop-down menu to display a text field called Icon File. The text in this field is a relative path from the model file (`.u2`) to where the image file can be found.

The formats that are supported for the icon image files are

- bitmap (file extension "`.bmp`")
- JPEG compressed images (file extension "`.jpeg`" or "`.jpg`")
- Enhanced Meta File (file extension "`.emf`")
- GIF (file extension "`.gif`")
- TIFF (file extension "`.tif`", "`.tiff`")
- Targa (file extension "`.tga`", "`.targa`")
- PCX (file extension "`.pcx`")

**Note**

*Using a white and transparent background for an icon image may result in a black background when Printing Diagrams.*

## Image Selector

The symbols in diagrams can also be displayed with a user-defined image using the Image Selector. This function is delivered with the Add-Ins. Activate the add-in `ImageSelector`, then the commands **Load image** and **Remove image** are available in the **Tools** menu.

## Undo

Multiple level of Undo and Redo is available. The whole tool (workspace window and editors) has a common undo stack. When an operation is undone, it is put first in the redo stack to make it possible to redo the undone operation.

**Note**

*You can undo operations in diagrams that are not currently visible.*

Some special considerations have to be taken when using Undo and Redo when in text-editing mode. Undo steps will be available for each update. An update is made according to the following scheme: a series of character additions will not cause an update until you do something else, for example back space, delete, arrow keys or mouse selection. Similarly will a series of deletes not cause an update until you do something different.

When doing an explicit unload (including revert) of a file/resource in a project the Undo stack is emptied.

Undo is not possible for file system operations.

The Undo stack is not emptied when a Save is performed.

## Model references

To find references to model definitions and their usage there is a group of shortcut commands with similar features. These commands have a contextual nature meaning that they will be dependant on the element that they are applied on.

### List references

List References is a shortcut command in the Model View, which applies to all model elements. The command calls up a dialog and returns a reference list in the References tab of the Output window. Possible settings:

- **References made to...**
  The listing contains all references to the model element, for example usage of a specific class as a type for an attribute is a reference to the class.

- **References made from...**
  The listing contains references from a selection, for example references from an attribute to the class used as its type.
- **Include Contained Hierarchy in Report**
  When selected, this alternative will cause the tool to recursively include any references to or from the elements contained under the selected element. An example is to find for all definitions outside a package, used by the package and its contained definitions.
- **Include Internal references in Report**
  When selected, this alternative will cause the tool to report references originating in the object or its contained hierarchy to itself, or to its contained hierarchy. An example of this is to find all uses of a package and of its contents without finding references made within the package.

### List presentations

List Presentations is a shortcut command in the Model View, which applies to all model elements. Returns a list of all presentation elements in the Presentations tab of the Output window.

### Reference existing

This is a shortcut command when placing a new symbol.

### Navigate

Shortcut command in the Model View, opens the Model Navigator or if there is no existing presentation for the element the Create Presentation dialog.

When you have a selection of multiple nodes in the Model view the commands **List References** and **List Presentations** will be applied to all selected elements.

## See also

"Add symbols" on page 177 in Chapter 7, *Working with Diagrams*

## Update model

When the **Active Modeler** add-in is activated the shortcut menu will contain a new choice named **Update model**. This command is used on unbound entities, to invoke them in the current model.

This feature is available in composite structure diagrams, use case diagrams and sequence diagrams.

### Usage

To update the model from a presentation element, select the element, right-click and select Update Model from the shortcut menu. The command is only available when there is a registered update model procedure for an element in the current selection. The Update Model command is also available in the Tools menu.

The Active Modeler tab in the Output window is filled with information about changes made to the model during the execution of the command. To navigate these changes, double-click a row in the tab or use F4 and SHIFT + F4 to traverse the list.

It is possible to undo the update model command just as any other command. The entire command is considered to be one action, even if more than one change is made to the model, and consequently all changes done by the command will be undone in one step.

## Nested symbols

Some symbols can be placed inside other symbols. When a symbol is created inside another symbol, the model element of the parent symbol is used as context for creation.

If the parent symbol is auto-sized the size of the parent will change to fit the created symbol. Otherwise the new symbol will be resized to fit within the boundaries of the parent symbol. A nested symbol can not be dragged outside the parent symbol boundaries.

## Symbols with compartments

Symbols with compartments have some special functionality related to the compartments and the contained text fields.

Compartments can contain text fields which in turn is associated with model elements. Compartment text fields are left-aligned.

Certain symbols, like for example the class symbol, are created with a default set of compartments. The class symbol for example will have an attribute and an operation compartment.

Compartments can be selected, and there are a set of operations possible to perform on them.

When hovering over a compartment for a moment, a tool tip will display the compartment type.

### Resizing

When a symbol with compartments is resized, any compartment that does not fit in the symbol will be hidden. If some content in the compartment can be hidden instead of the entire symbol, this will be done instead.

When a symbol is larger than necessary to display all the compartment contents, then the extra space will be evenly distributed among the compartments.

### Creating compartments

If a symbol can contain compartments there is a **Compartments** sub-menu available on the symbol's shortcut menu. The sub-menu contains a set of operations for creating compartments in the form **Create <Element> compartment**. Executing one of these operations will create a compartment that can be used to create and display elements of the element type. New compartments will be added in the bottom of the symbol.

### Deleting compartments

A compartment can be deleted by selecting it and using the normal **Delete** command. Certain compartments can also be directly associated with the model elements that the compartment is showing, and in that case the **Delete Model** command can also be used.

### Moving compartments

The order of compartments can be changed by using the **Move Up** and **Move Down** commands available on the **Move** toolbar.

### Show/Hide on compartments

When a specific compartment is selected, the shortcut menu will give the possibility to show and hide elements of the type that the compartment is used for. Show and hide operation will only be displayed in the shortcut menu if they are applicable.

When the symbol is selected, the shortcut menu will give the possibility to show and hide elements in any of the existing compartments, or create a new compartment to display a certain type of model element. These operations will only be displayed if applicable. If there are several compartments showing the same type of model element, show and hide operations will only be done on the one first in order. To show and hide elements on a specific compartment use the shortcut menu of the compartment instead.

It should be noted that the owned model elements will not be shown by default if an already created element is dragged into a diagram.

**See also**

"Default Class Symbol Appearance" on page 2463

For the elements to become visible it is also possible to drag-and-drop them into the compartment or symbol or type them in manually.

**Hint**

*Using Name completion is a good way of avoiding to create new features by mistake. Start typing the name, press CTRL + SPACEBAR or SHIFT + SPACEBAR, if there are multiple possibilities a list will be displayed.*

## Compartment text fields

### Delete element

A text field in a compartment is a separate presentational element that is associated with a model element. Due to this it is not possible to delete a text field connected to a model element by deleting all text on the line. To delete the element associated with the label, enter text mode and use the **Delete <Element>** operation from the shortcut menu.

**Note**

*It is not possible to delete a text field connected to a model element by deleting the text on the line, this will only delete the characters on that line. The row will still be connected to the model element.*

A text field **not** connected to a model element can be deleted by deleting all the text on the line and then pressing backspace or delete. It is also possible to delete an empty text field not connected to a model element by pressing backspace when the text cursor is first on the text line below or by pressing delete when the text cursor is last on the text line above.

### Hide element

To hide a specific element displayed in a compartment text field enter text edit mode and use the **Hide <Element>** operation from the shortcut menu.

### Move text fields

It is possible to move feature text fields up and down with the **Move Up** and **Move Down** toolbar buttons (found on the **Move** toolbar).

# Common Line Operations

- Line styles
- Draw lines
- Editing vertices
- Move lines
- Delete lines
- Re-direct and bi-direct lines

## Line styles

There are five different line styles that can be applied to a line. After a line is created these styles are available in the context menu on the line.

### Auto-routed (assign endpoints)

Line is routed automatically so that obstacles are avoided. Line is orthogonal as long as there is a possible route for the line. In other cases the line is drawn as a straight line. Endpoints are automatically reassigned to make the shortest route as possible. When an endpoint is moved the line style of that line will automatically be changed to Auto-routed (keep endpoints).

Note that there is no difference in behavior from the Auto-routed (keep endpoints) line style if the line can only be connected at the center of a symbol.

### Auto-routed (keep endpoints)

Line is routed automatically so that obstacles are avoided. Line is orthogonal as long as there is a possible route for the line. In other cases the line is drawn as a straight line.

If the source endpoint is shared with another line of the same type, a tree structure will be routed as far as it is possible. This is only possible for certain type of lines that are common to draw as tree structures, such as the generalization line.

### Orthogonal

Line is always kept orthogonal and line vertices and segments can be moved. Vertices can be added and removed from the line.

### Non-orthogonal

Line vertices can be moved, added and removed without restrictions. If a non-orthogonal line is rearranged into an orthogonal line, the line style is automatically changed to Orthogonal.

### Bezier

Will give the line a curved layout. When the line is selected two control points are displayed which can be used to shape the curve.

## See also

"UML Editing Line Styles" on page 2466

## Draw lines

A line can be created either by using the toolbar button or the line handle representing the line.

### Creating a line with a line handle

1. Select the source symbol.
2. Click the line handle.
3. Add vertices and/or lock endpoint (optional).
4. Click the target symbol or line.

### Creating a line with a toolbar button

1. Click the toolbar button.
2. Click the source symbol.
3. Add vertices and/or lock endpoint (optional).
4. Click the target symbol or line.

Vertices can be added while creating the line, with the exception of auto-routed lines. When it is allowed to place a vertex the cursor will have the shape of a plus sign.

For all line styles it is possible to lock the starting point position to the symbol edge, if the starting point is selectable. When creating a line with the line style Auto-routed (assign endpoints) or Auto-routed (keep endpoints) a cursor in the shape of a padlock is displayed. When clicking at this state the starting point of the line will be locked to its current position. If the line have the line style Auto-routed (assign endpoints) as default line style and the endpoint is locked in this way, the line style will automatically be changed to Auto-routed (keep endpoints). When creating a line with a line style different from Auto-routed (assign endpoints) and Auto-routed (keep endpoints) the starting point can be locked by holding down SHIFT and clicking.

## Editing vertices

To add a vertex for an existing line hold down CTRL and click on the segment where the vertex should be created.

To remove a vertex hold down CTRL and click on the vertex that should be removed.

This can only be done for lines with the line style Orthogonal or Non-orthogonal applied. The mouse cursor will change to indicate that the operation is possible.

**See also**

"Connect symbols" on page 182

## Move lines

To move a line, click one of the endpoints and drag it to the desired location.

## Delete lines

Lines are in many cases representing a model element which will remain in the model even if they are deleted from a diagram. If you want to completely remove a line, for example an association, then make sure to use Delete from Model.

## Re-direct and bi-direct lines

- To re-direct a line (when applicable), right-click the line and choose **Re-direct** from the shortcut menu.
- To bi-direct a line (when applicable), right-click the line with the cursor close to the side without direction or signal list. Select **Enabled Direction** from the shortcut menu.
- If a line is bi-directed, and you want to allow it only one direction, locate the cursor over the line, close to the side you want to disable. Then deselect **Enabled Direction** from the shortcut menu.

  You can also re-direct the line before or after this operation to make it point in the desired direction.

# 8

# *UML Language Guide*

This chapter describes the UML language as implemented and supported in Tau 4.2.

- For more information on the supported version of UML, see "UML version" on page 200.

**See also**

Chapter 6, Working with Models

Chapter 7, Working with Diagrams

"Description of Workflow" on page 51 in Chapter 4, *Introduction to Tau 4.2*

# Introduction

UML is a modeling language that allows you to specify, visualize, document, and construct software and systems. In subsequent sections you find information about the different diagrams and constructs that can be used to describe the structure and behavior of systems at different levels of abstraction. Some constructs are more useful in early development phases, such as requirements and analysis, while others are more useful in later development phases, such as design, implementation, and test. This ability to tie together the different development phases is one of the primary strengths of UML.

## UML version

The language used in Tau is based on the latest OMG UML 2.1 Superstructure submission. In some cases the implementation of Tau differs from the language specification; this is primarily due either to tool optimizations or the fact that some design decisions were founded on earlier versions of the submission.

Tau also includes some extensions to the language, for example the possibility to use a textual syntax in conjunction with the graphical notation defined for UML.

## Diagrams

UML consists of a set of diagrams that are used to express different viewpoints of a system. Some diagrams focus on the structure of the system, while others are dedicated to describing behavioral aspects of the system, such as how an entity interacts with another entity or the set of actions to be performed under specific conditions. Typically, these diagrams are the primary means through which you specify systems.

The diagrams that are supported in Tau are the following:

| Diagram | Purpose |
|---|---|
| Use case diagram | Describes how a set of actors interacts in terms of use cases, usually in the context of a subject (the described system). |
| Sequence diagram | Describes the event sequence for a use case or an operation. |
| Package diagram | Describes packages and dependencies between them. |
| Class diagram | Declares classes and their relations to each other, typically in the scope of a package or another (container) class. |
| Composite structure diagram | Describes how parts of a (container) class are connected to each other to form an internal structure of the container. |
| Activity Diagram | Used to show parallel and intertwined behavior. This may allow a simplified view of a complex structure where it is possible to focus on a specific flow of control. |
| Interaction overview diagram | Describes some form of parallel behavior. It is often used to describe a use case. |
| Component diagram | Focused on the design of components and shows relations and structure of components |
| Deployment diagram | Used to show how the physical implementation is structured and the relations between software and hardware |
| State machine diagram | Defines the behavior of classes, state machines and operations. |
| Text diagram | Defines an entity textually, rather than graphically. |

## Models and diagrams

A model is a representation of a physical system, and is typically defined by the entities contained in one or more packages.

In Tau, everything that is in a project belongs to the same model.

Since the model is a representation of a system, it should only be as detailed as necessary. If, for example, it should be used as the source for automatic application generation, it needs to contain quite a lot more detail at an algorithmic level than if it is used to visualize requirements.

The model contains all entities that are necessary to describe a system; this includes diagrams and model elements. The model, or rather the model elements it contains, are typically shown in different diagrams using symbols (sometimes called presentation elements, as opposed to model elements).

### Model elements

The primary contents of a model are model elements such as classes, attributes, operations, actions, and constraints. A model element is used to store all characteristics of an entity. It is then possible to show different aspects of a model element in diagrams. For example, one class diagram may show the attributes and operations of the class, while another class diagram may shown the class hierarchy in which it is defined. These diagrams give partial views of the same model element, but many more are possible.

### Symbols

Symbols are used to graphically visualize (parts of) model elements. Each symbol is a two-dimensional object that is shown in a diagram. It has a size that specifies its dimensions and a position that is given in terms of the coordinate system of its diagram.

Most symbols are direct visualizations of a corresponding model element, such as the class symbol but there are a few that have no underlying model element, such as the text symbol. These are then only associated with a specific diagram.

The distinction between model element and symbol is important, but in daily speak the distinction between the two is often blurred. A class model element or a class symbol is commonly referred to simply as class.

### Different views of a model element

In Figure 50 on page 203, there is an example of the model element a, which is shown using three different views. First, it is shown as an attribute of the class C. Second, it is shown as an association end between the classes C and D. Third, it is shown as a part of the internal structure of class C.

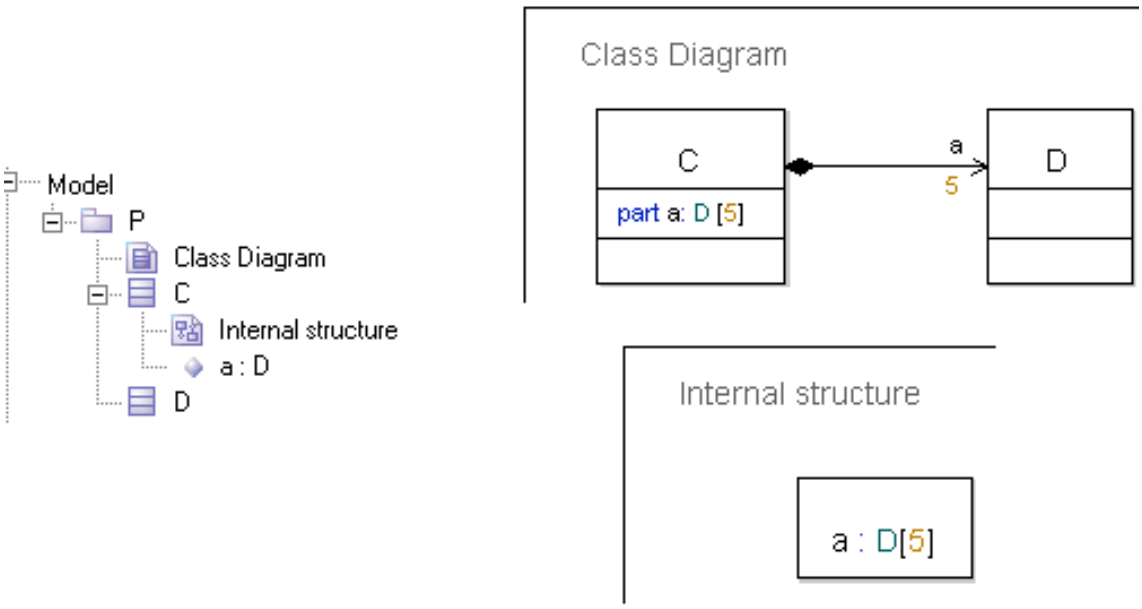



*Figure 50: Example of different views of an attribute*

Here it is also possible to appreciate the difference between a model deletion (Delete from Model) and an ordinary Delete from a diagram. The browser view to the left shows the model, and when deleting elements from the browser view you delete them from the model *and* the diagrams in which they are shown. The two diagram views to the right, however, show the same attribute `a` in three different ways: in the attribute compartment of the class C, as an association end between the classes C and D, and as a part of the internal structure of the class C.

### Deleting symbols and model elements

Elements can be deleted in two different ways from a diagram. An ordinary Delete removes the symbol, but the model element is retained in the model. A Delete from Model operation deletes the element from the model and from *all other* diagrams in which it is shown.

In some diagrams, model elements and symbols are tightly connected with each other. This includes text diagrams, sequence diagrams, and state machine diagrams. Here, there is a one-to-one mapping between symbols and model elements, and if one is deleted then the other is also deleted. (In other words, a delete is the same as a Delete from Model in these diagrams.) In particular, this is applies to for example actions and transitions, but not for states.

**See also**

"Add symbols" on page 177

"Move symbols" on page 180

"Resize symbols" on page 181

"Connect symbols" on page 182

"Edit text fields in symbols" on page 183

"Copy, cut, delete or paste symbols" on page 186

## List of language constructs

The following table lists all the concrete model elements as well as the most significant other language constructs in UML.

| UML model element |
|---|
| Accept Event, Accept Time Event, Access, Action (in operation body, state machine and state machine diagram), Action (in interaction and sequence diagrams), Action Node (in activity diagrams), Active class, Activity, Activity Final, Actors, Aggregation, Arbitrary value (any) expression, Artifact, Assignment, Association, Attribute |
| Behavior port |
| Choice, Class, Classifier, Comment, Component, Composite state, Composition, Compound statement, Conditional expression, Constant, Connector (in composite structure diagrams), Connector (in activity diagrams), Continuation, Co-region, Create |

| UML model element |
| --- |
| Datatype, Decision (in state machine diagrams), Decision (in activity diagrams), Dependency, Deployment, Deployment specification, Diagrams, Destroy |
| Entry connection point, Execution environment, Exit connection point, Expressions, Extension |
| Field expression, Flow Final, Fork |
| Generalization, Guard |
| History nextstate |
| Imperative expressions, Realization, Import, Index expression, Initial Node, Inline Frame, Interaction, Interaction reference, Interface, Internals |
| Join, Junction |
| Lifeline, Literal |
| Manifestation, Merge, Message, Method, Method call |
| New, Nextstate, Node, Now expression |
| Object Node, Offspring, Operation, Operation body, Signal sending action (output) |
| Package, Parent, Part, Activity Partition, Pid expressions, Pin, Port, Predefined, Profile |
| Range check expression, Realized interface, Required interface, Return |
| Save, Self, Send Signal, Sender, Signal, Signallist, Signature, Initial transition, State, State machine, State machine implementation, State expression, Stereotype, Stop, Subjects, Syntype |
| Tag definition, Tagged value, Target code expression, Action (task), This expression, Timer, Timer active expression, Timer reset, Timer reset action, Timer set, Timer set action, Timer timeout, Transition |
| Use cases |

## Scope, model elements, and diagrams

Some model elements, like packages and classes, represent name scopes. This means that they are allowed to contain definitions of other model elements. All definitions within a name scope must be uniquely named, or the semantic checker will complain. You can think of a scope as a container or grouping of model elements that belong together.

Most scopes may not only contain model elements, but also diagrams in which those model elements are shown. The table below shows which diagrams are available for each scope.

| Scope unit | Allowed model elements | Diagrams |
|---|---|---|
| Package | Package, Class, Use cases, Artifact, Stereotype, Association, Datatype, Interface, Syntype, Choice, Operation, Attribute, Signal, Signallist, Timer, State machine | Class diagram<br>Sequence diagram<br>Text diagram<br>Use case diagram |
| Class | Class, Artifact, Stereotype, Datatype, Interface, Syntype, Choice, Signal, Signallist, Timer, Attribute, Operation, Use cases, State machine, | Class diagram<br>Composite structure diagram,<br>Text diagram |
| Use cases | Interaction, State machine implementation, Artifact, Operation body | Sequence diagram<br>State machine diagram<br>Text diagram |
| Interaction | Lifeline | Sequence diagram<br>Use case diagram |
| Stereotype | Attribute | |
| Datatype | Literal, Operation | |
| Choice | Attribute, Operation, | |
| Interface | Signal, Timer, Attribute, Operation | |

| Scope unit | Allowed model elements | Diagrams |
|---|---|---|
| Operation | Operation body, State machine implementation, Interaction | |
| Operation body | State machine, Class, Artifact, Stereotype, Datatype, Interface, Syntype, Signal, Signallist, Timer, Operation, Attribute | State machine diagram<br>Text diagram |
| State machine implementation | Class, Artifact, Stereotype, Datatype, Interface, Syntype, Signal, Signallist, Timer, Operation, State, Action, Attribute | State machine diagram<br>Class diagram<br>Use case diagram<br>Text diagram |
| Activity implementation | Initial Node, Action Node, Object Node, Decision, Merge, Fork, Join, Connector, Accept Event, Send Signal, Accept Time Event, Activity Final, Flow Final, Activity Partition | Activity Diagram |
| Compound statement | Action, Attribute | |

### Overloaded Definitions

For certain kinds of definitions it is allowed to have many definitions with the same name in a scope. This is true for behavioral features, such as Operation, Signal, Timer and State machine. These definitions are identified not only by their names, but also by the types of their parameters. The name and list of parameter types is called the signature of the behavioral feature. All behavioral features in the same scope must have unique signatures. Two behavioral features in the same scope which have the same name, but different signatures, are said to be overloaded.

# General Language Constructs

There are some language constructs in UML that are common to several diagrams.

# Names

All definitions in a UML model should have a name—an identifier. There are certain rules to which these have to adhere.

### Naming rules

The characters that are allowed in a name are letters, digits, and '_' (underscore).

The first character of a name cannot be a digit, but should be either a letter or an underscore. There is furthermore a special case for destructor names, which always start with an initial '~' (tilde).

### Using spaces and special characters in identifiers

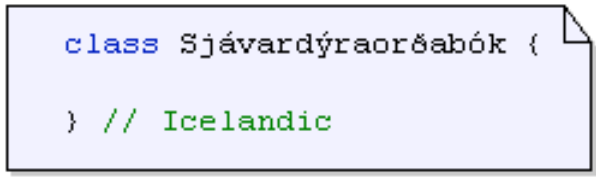



*Figure 51: Using special characters in identifiers*

By enclosing a name in single quotes, it is possible to get rid of the above mentioned restrictions, so that (almost) any character can be part of a name, see Figure 51 on page 208. For example, it is possible to use spaces in a name as long as the name is enclosed by single quotes, see Example 12 on page 209.

There exist a number of escape characters for string handling. They are \n, \t, \b, \r and \f and can be placed inside charstring (inside "") or used as a character (e.g. '\n').

The "\"" is used in charstring, "\'" respectively as character, and "\\" is used in both to represent backslash. Any other escaped character between quotes ('\+', '\s') is interpreted as identifier (+ and s respectively). Inside a quoted string any other character can follow the slash, representing just itself (e.g. "a\qa" = "aqa").

```
\n: new line
\t: tab
\b: backspace
\r: carriage return
\f: form feed
```

```
\": quotation mark, e.g. "my \"quoted\" word"
\': apostrophe character, '\''
\\: backslash
```

**Example 12: Spaces in identifiers**

```
Boolean 'has finished'=false;
```

## Case sensitivity

Identifiers are case sensitive. This means that names that differ only in the way they use lower and upper case characters are distinct.

**Example 13: Case sensitivity**

```
Integer MyInt, myint; // Two distinct attributes
```

## References

Named definitions may be referenced from other places in a model. In simple cases a reference just consists of the name of the definition (enclosed in single quotes if necessary). However, in the general case a reference can be more complex.

- A reference may contain a qualifier.

In some cases it is necessary to qualify a name in order to be able to distinguish a definition in one scope from another definition with the same name but in another scope. This is done by prefixing the identifier with the scope path and using the special scope resolution operator "::". Global names have no path, and are simply preceded by "::". Qualifiers that start with "::" are called absolute qualifiers, while those that start with a name are called relative qualifiers.

- A reference may contain actual template arguments.

If the referenced definition is a template (i.e. has Template parameters) the reference must contain actual values for its template parameters. The actual template arguments are given as a comma separated list after the name within '<' '>' brackets.

- A reference may contain a list of parameter types.

When referring to a behavioral feature you must add the names of the parameter types, since not only the name but also the parameter types are part of the signature of the behavioral feature. The parameter type names are given enclosed in parenthesis after the name.

**Example 14: Different kinds of references**

In this example two attributes refer to their types using a qualified name.

```
::Predefined::Integer i;
UtilityTypes::Sorts::ClientIdx j;
```

If the type is a template class actual template arguments must be specified:

```
MyClass<Integer, 4> k;
```

When referring to a behavioral feature such as an operation, the parameter types must be specified. Note also the keyword ‘operation’ which must be used to syntactically disambiguate such a reference from an ordinary call of the operation.

```
OperationReference r = operation foo(Integer, Boolean);
```

## Reserved words

Some names are reserved words in Tau and cannot be used to name model elements directly. For a complete list of reserved words, see .

Although it is possible to use reserved words as names of definitions by enclosing them with single quotes, risk for confusion is apparent and this should be done only when absolutely necessary.

**Example 15: Using single quotes for names that are otherwise reserved**

```
Integer ‘class’; // confusing attribute name, but valid
```

**See also**

"Reserved words" on page 957, an extended list which is applicable when generating an application or a Model Verifier.

## Alternative syntax

In addition to the graphical notation defined by UML, a complementary textual syntax is defined for describing the model in plain text. This can be used in lieu of the graphical symbols, or in conjunction with them.

The text is shown either in a Text diagram or inside a Text symbol within a diagram.

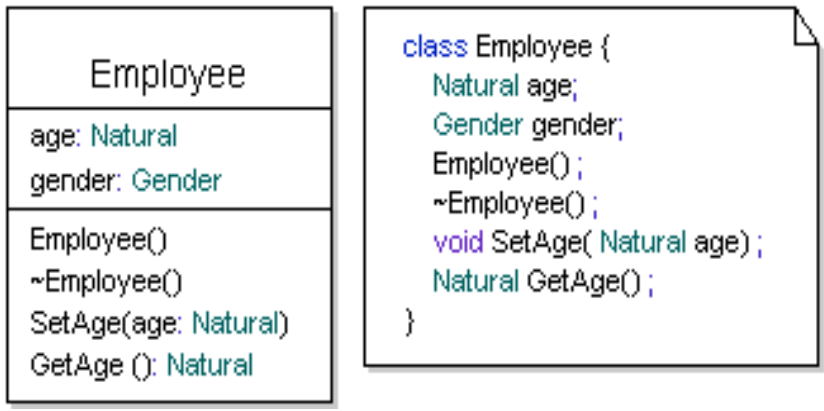



*Figure 52: Example of differences between the syntax in class symbols and the textual syntax*

In Figure 52 on page 211, the same model element is shown twice in a single diagram. To the left, a graphical notation is used, and to the right, a textual syntax within a text symbol is used. Changes in either of these views are automatically propagated to the other.

## Common element properties

The following properties exist for many different kinds of model elements. They can be inspected and controlled through the Properties Editor.

### Visibility

Many model elements have a visibility, which is used to determine access rights for elements outside the scope in which the element is defined. Within a scope, all elements can be accessed regardless of visibility. There are different levels of visibility:

- **Public**
  All elements that can see (access) the container of the element with public visibility can also access the element.
- **Protected**
  All elements in the same scope as the element with protected visibility and the subclasses of its container can access the element
- **Private**
  Only elements within the same scope as the element with private visibility can access the element.
- **Package**
  An element with package visibility is accessible by all elements enclosed within the same package.
- **None**
  If visibility is not specified, the element gets a default visibility according to the table below.

The default visibility of a definition is decided by its scope and type.

| Scope | Visibility |
|---|---|
| `Class, Choice, Stereotype, Collaboration, Artifact` | Private |
| `Package` | Public |
| `Interface` | Public |
| `DataType` | Public |

**Note**

*Literals always have public visibility.*
*All literals and public static members of a datatype are visible outside of a datatype without a qualifier. Qualifiers are only required to resolve ambiguities, for example when two datatypes in the same scope have literals with the same name.*

### Virtuality

Virtuality comes into play when you have generalization between classifiers such as classes, and determines whether contained model elements of a specialized class can be redefined or not.

Virtuality only applies to elements that are contained in types (classifiers that can be specialized). If the container is specialized, the individual virtuality of each contained element controls if that element may be changed.

- **Virtual**
  If a contained element is virtual, it is allowed to redefine (change) this element when its container is specialized.
- **Redefined**
  If an element in a specialized container is redefined, it is changing the definition of the original element from the base container. The original element in the base container must be virtual.

  A redefined element is still virtual, that is if the container is specialized once more, the element may be redefined further.
- **Finalized**
  If an element in a specialized container is finalized, it is changing the definition of the original element from the base container. The original element in the base container must be virtual. Finalizing also implies prohibiting further redefinition of this element if the container is specialized once more. In this sense, finalized means “redefined but not virtual“.
- **None**
  If a contained element has no virtuality, it is not allowed to redefine (change) this element when the container is specialized.

### Derived

If an element is derived, it means that its value can be calculated by means of other elements. Exactly how to specify how to perform the calculation of the value is context dependent.

A common case of derived elements are derived attributes. For these the derivation rules used when accessing the attribute can be specified using accessor operations called ‘get’ and ‘set’.

**Example 16: Specifying derivation rules for a derived attribute**

```
Integer y;
Integer / x
  get { return 5; }
  set { y = value; };
```

### Other properties

- **External**
  If a definition is external, it means that it resides outside this model. The code generators supplied will not generate code for external elements. External elements can thus be seen as model representations of externally available definitions.
- **Abstract**
  If a classifier is abstract, it is not allowed to directly instantiate the classifier. If an abstract classifier is specialized, which it typically is, it is allowed to instantiate the specializing classifier (unless it too is marked as abstract).
- **Static**
  If a definition is static, all instances of the containing classifier shares the implementation for this element, that is uses the same piece of data. Hence a static definition can be used without having an instance of the classifier in which it is defined.

### Parameters

Definitions that are behavioral features, such as Operation, Signal or State machine, may have parameters. The general format (used in classifier symbols and in the Properties Editor) is:

```
name:type, name2: type2
```

A parameter has a direction which specifies the direction in which data “flows” in a call to the behavior:

- **In** (default)
  Data is passed from the caller to the invoked behavior.

- **In/Out**
  Data is passed from the caller to the invoked behavior and also from the invoked behavior back to the caller.
- **Out**
  Data is passed from the invoked behavior back to the caller.
- **Return**
  Data is passed from the invoked behavior back to the caller as the return value of the call. At most one parameter may be a return parameter.

### Template parameters

A template parameter is a concept for allowing flexible, context-free classifiers. Another name for template parameters is context parameters.

Elements that can be specialized or that can be instantiated (called) may have template parameters, for example classes and operations.

Template parameters are bound with actual parameter "values" either at instantiation or when the containing classifier is specialized or redefined. It is allowed to bind a subset of the template parameters at specialization. On instantiation, all template parameters must be bound.

As a general rule, whenever a template definition is referenced actual values for all its template parameters must be specified. There are two exceptions to this rule

1. If a template parameter has a default value, it is not needed to give an actual value for it. The default value will then be used.
2. In calls to a behavioral feature with template parameters it is not necessary to specify the actual template arguments if these can be deduced from the actual call arguments used in the call.

The operators `reinterpret_cast<T>` and `cast<T>` cannot be used as actual template parameters. Any template instantiation containing `reinterpret_cast<T>` or `cast<T>` operator cannot be resolved by name resolution.

**Example 17: Template instantiation containing casting operator**————————

The following example illustrates this restriction:

```
template<const Integer x>
class MyTemplate { }
enum E { L }
```

```
/* These template instantations cannot be resolved */
MyTemplate<cast<Integer>(L)> myVar1;
MyTemplate<reinterpret_cast<Integer>(L)> myVar1;
```

## Predefined names

In the provided utility package Predefined  are found a number of useful datatypes, literal values and operations. The names of these entities are not reserved, but it is recommended to avoid using these names for other entities as that is likely to cause human misinterpretation.

**See also**

"Predefined" on page 390

# Use Case Modeling

Use case modeling focuses on determining the context of a system or parts of it, often in terms of the actors that interact with it, but also on modeling the requirements of the behavior of these elements.

## Use case diagram

A use case diagram illustrates a usage situation by showing the relationships between use cases and actors. A use case diagram gives a static view of dynamic aspects of systems.

### Example

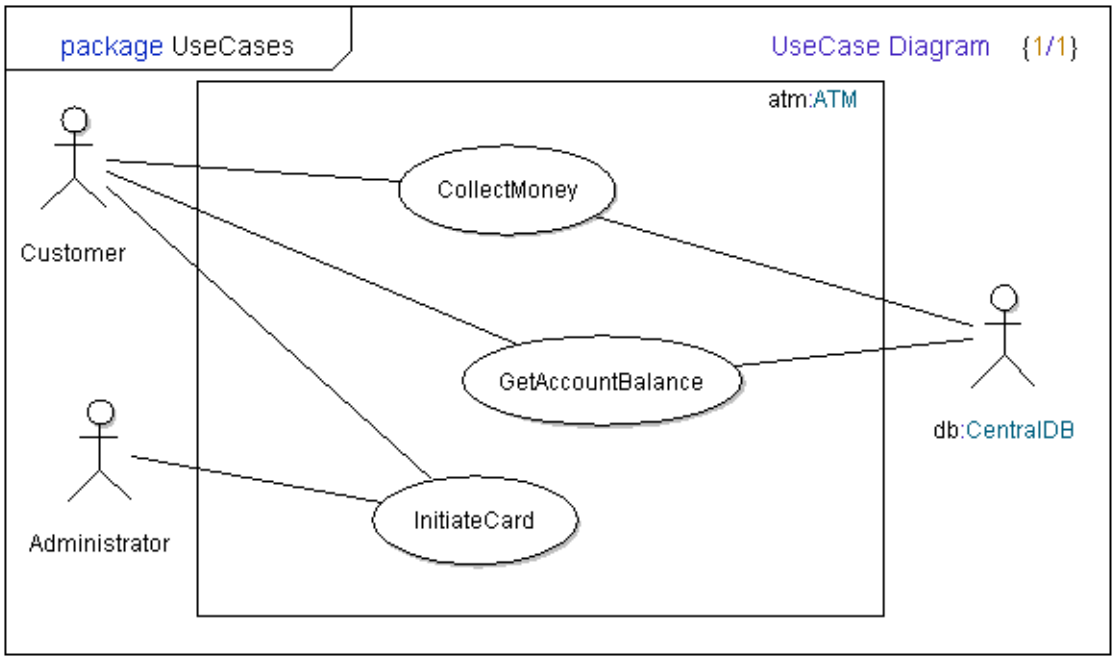



*Figure 53: Use case diagram with Actors, Use Cases, a Subject and Association relationship between the Actors and the Use Cases*

### Model elements in use case diagrams

The following elements are found in use case diagrams

- Use cases
- Actors
- Subjects
- Dependencies
- Includes
- Extends
- Generalizations
- Association

### Create a use case diagram

Use case diagrams can be included in packages, classes and collaborations.

1. Select the package (class, collaboration) in the Model View.
2. From the shortcut menu select **New** and then **Use Case diagram**.

   Use cases can then be drawn using the toolbar or you can drag use cases from your model into a use case diagram.

- To use the toolbar, first click on the use case symbol and then click in your diagram where you want to position the use case symbol.

## Use cases

A use case represents a coherent unit of functionality provided by a system or parts of a system. Usually, the system is represented by a class. The functionality is often manifested in terms of communications between the system and one or more outside actors, including the behavior performed by the system.

A use case is in many ways similar to an operation, and is in fact modeled as an operation with the stereotype «use case».

### Symbol


*Figure 54: Use case symbol*

A use case is visualized through the use case symbol in a use case diagram. It can be specified within the scope of:

- a package
- a class
- a collaboration
- an implementation

### The description of a use case

The behavior of a use case can be defined by:

- an interaction
- a state machine
- an operation body
- an activity

It is also possible to describe the behavior of a use case textually. In this case, there is often some structure to the text, where the name of the use case is given, followed by its goals, preconditions and post conditions, exceptional cases, and the actual functionality in the form of actions that should be performed by the use case.

**Example 18: A textual use case**

```
Use case: CloseAccount
Goal: Close a user account and make sure the balance of
the account is settled
Preconditions: Customer has an open account
Postconditions: Customer has closed the account and has
paid outstanding dues
Description:
1. Check balance of account
2.a If balance is positive, pay customer
2.b If balance is negative, collect payment from
customer
3. Terminate card associated with account
4. Close account
```

### Naming use cases

When naming Use Cases, it is common to use some kind of verbal description, typically a phrase which contains a verb and an object, for example “do something”. It is possible to use this name convention, in spite of the fact that names may not contain spaces, by using a quoted name:

**Example 19: Quoted use case name**

```
<<usecase>> void 'Open Account' ();
```

More often, the verb and the noun are written together without the white space.

# Actors

An actor represents an entity that takes part in a use case, for example to initiate the functionality or as a resource for information needed by the use case.

### Symbol


*Figure 55: Actor symbol*

An actor is visualized using a stick figure symbol in a use case diagram. Actors are connected to use cases using Association.

### The role of an actor

In a use case diagram, the focus is on showing the relationships between actors and use cases. An actor is an entity that is involved in use cases, most often in the context of one or more Subjects. An actor is external to the subject for which the use case is defined, and can be human users, external hardware devices, or other subjects. An actor is not necessarily one single physical entity, but can for example be an entire computer network.

In different use cases, there can be different actors representing the same physical entity, but with a different role. An actor may also represent different physical entities in different use cases.

The actor is either a reference to a part or an instance of a class.

In a use case diagram the focus is on showing the relationships between actors and use cases. However, sometimes it is also beneficial to focus on the type-like aspects of an actor. For example to show how actors relate to each other using inheritance or show some properties of the actor. This is shown in class diagrams where actors are visualized as a class symbol with the «actor» stereotype.

The Actor symbol visualizes stereotypes applied to the actor and the class that the actor references (only if no stereotype is applied to the actor)

# Subjects

A subject defines the system boundary for a set of use cases. A subject can represent a system, subsystem or class. The subject is either a reference to a part or an instance of a class.

Subject corresponds to the System Boundary of use cases in UML 1.X.

### Symbol

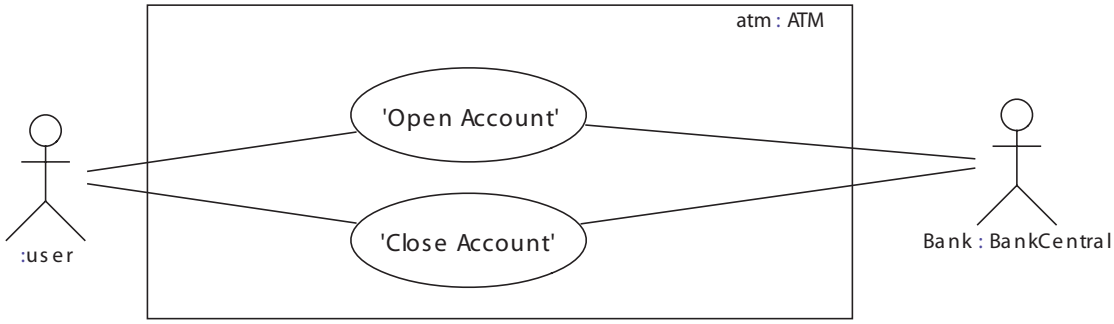



*Figure 56: Subject symbol (ATM)*

Use cases can be enclosed inside a subject symbol. The subject symbol is drawn around a set of use cases that represent the behavior of for example an active class. A name and the class type can be written in the upper right corner of a subject symbol.

A hatched background color can be assigned to the subject symbol.

# Relationships

The following relationships can be used within a collaboration or a use case diagram:

### Association

The association relationship is used between an actor and a use case and indicates that an actor participates in that use case. Reversely, the use case is performed by the actor. One actor may participate in several use cases and one use case may have several participating actors. Association text is informal.

### Includes

The include relationship is used between different use cases to indicate that one use case is part of another use case. This provides a mechanism to split large use cases into smaller ones. The behavior of the including use case is typically not meaningful by itself, but is dependent on the included use cases.

### Extends

The extend relationship is used between different use cases to indicate how and when a use case should be inserted into an extended use case. The extended use case should be complete by itself; the extensions typically describe supplementary functionality to be addressed under certain conditions.

### Dependencies

Dependencies may be specified between use cases or between actors. A dependency does not give any indication about how the entities are related.

When a dependency is created between two use cases it will implicitly become an include relationship.

### Generalizations

A generalization can be specified between use cases; one use case may specialize a more general use case. For actors that are associated with classes a generalization can be specified. Generalization text is informal.

**See also**

"Relationships in UML" on page 375

## Update model

When the **Active Modeler** add-in is activated the shortcut menu will contain a new choice named **Update model**. This command is used on unbound entities, to invoke them in the current model.

For use case diagrams the following is supported:

### Use Case Diagram

Updates the entire diagram by updating every element of the diagram.

### Actor Symbol

If actor symbol is bound, but the type of the actor is unbound, a new class is created. The name of the class is the same as the name of the "`actor`" + "`Class`". If the actor symbol is unbound (grey lines) an attribute corresponding to the actor is created in addition to the class.

### Subject Symbol

If the type of the Subject is unbound, a new class is created. The name of the class is the same as the name of the "`subject`" + "`Class`".

# Scenario Modeling

Scenario modeling focuses on describing scenarios of system or subsystem usage. These scenarios are described as sequences of events that occur on lifelines.

When describing message interactions in increasing detail during this modeling activity, a clearer view emerges of how the responsibilities are divided between components of the system, but also of the borderline between the system and the external actors that interact with it.

The scenario modeling activity often takes place rather early in the analysis activity, but can of course also continue, with greater precision, in the design activity. The scenarios that are produced are specifications of the dynamic interfaces of the system and system components. They often have a twofold purpose:

- as a basis for the behavior modeling of components
- as a basis for test cases.

In UML scenarios are modeled using Interactions and the events are shown in Sequence diagrams as described in this section. Interaction overview diagrams are used to control and coordinate individual interactions.

Scenario modeling is very often done as part of a use case analysis. For each use case an interaction is created describing the behavior associated with the use case and a sequence diagram is used to visualize the interaction.

# Sequence diagram

## Description

A Sequence diagram describes an Interaction, visualizing the message interchange between lifelines, but also other event occurrences.

## Example

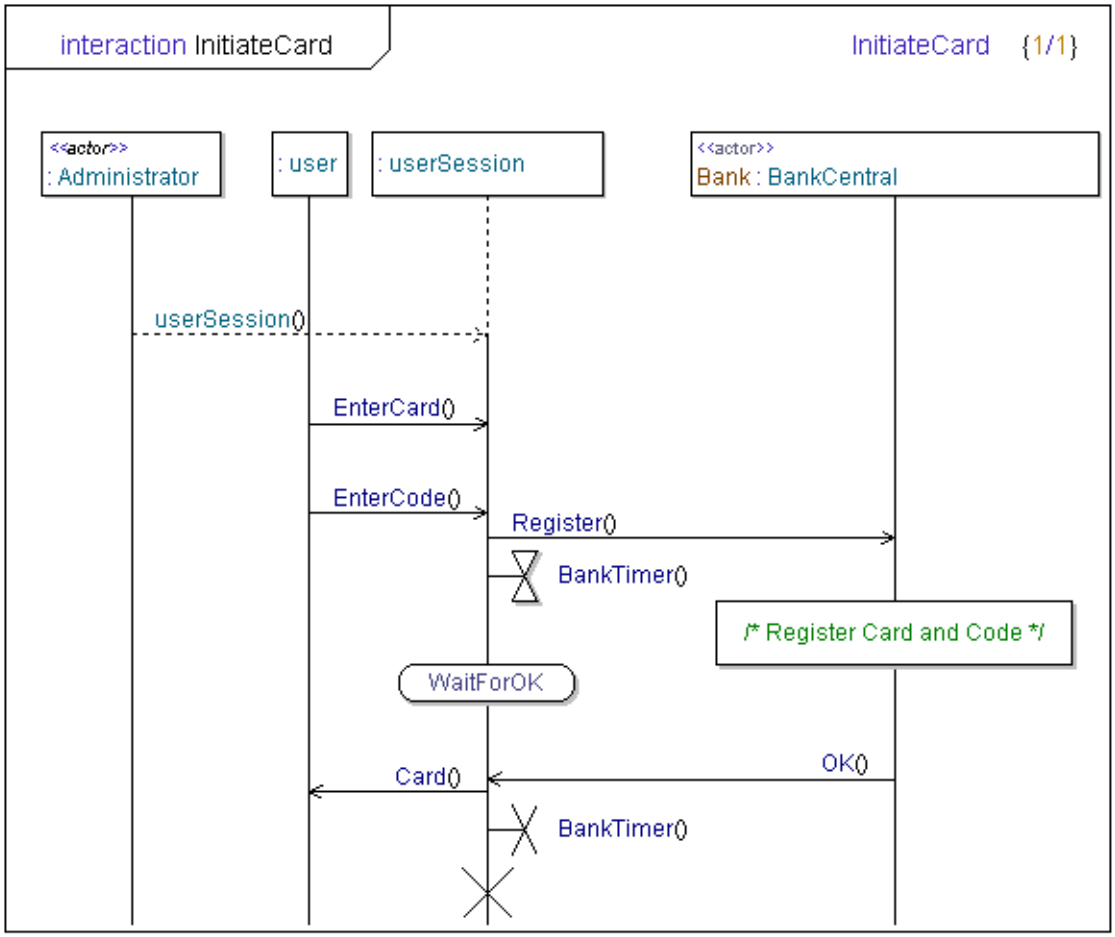



*Figure 57: Sequence diagram*

## Model elements in sequence diagrams

The following model elements can be found in sequence diagrams:

- Lifeline
- Message

- Action
- State
- Interaction reference
- Timer event
- Time specification line
- Create
- Destroy
- Inline Frame
- Co-region
- Continuation
- Method call

### Create a sequence diagram

A sequence diagram is a graphical description of the implementation of an Interaction. When creating a sequence diagram for example in a package it will automatically be encapsulated in an Interaction with its implementation.

It is however also possible to give a sequence diagram as implementation of other behaviors, such as operations and use cases. To accomplish this the sequence diagram can be created directly inside the behavior itself.

There are also options related to sequence diagrams:

- Message separation
- Lifeline separation

### The lifeline ruler section

When the header is not visible on screen at its normal position in the diagram because the header is scrolled out of sight in the vertical direction, then the header is instead visible in the lifeline ruler section.

## Interaction

An interaction is a description of the behavior of a use case, operation or other entity that can have a behavior. In an interaction the focus is on information exchange between parts. It is typically described by a Sequence diagram.

The semantics of an interaction is defined by the set of traces that can be derived from the interaction. A trace is a sequence of event occurrences. This sequence is not necessarily totally ordered. The traces may describe both possible and impossible scenarios.

Interactions can be referenced from within other interactions, thus allowing reuse. This is normally done by the Interaction reference symbol that reference another use case or operation that contains an interaction as its behavior definition. It is also possible to refer to an interaction by a Lifeline decomposition.

Interactions are typically used in two different ways:

- to specify the externally visible behavior of a system and its components
- to describe a trace of an execution of a system

**See also**

"Sequence diagram" on page 224

"Use cases" on page 218

## Interaction reference

An Interaction Reference is used to represent references of interactions in sequence diagrams. The referenced interaction is usually described in a sequence diagram of its own.The name used in the Interaction Reference is the name of the use case or operation that contains the interaction, not the name of the interaction itself.

The interaction reference is useful in two ways:

- It can be used as an encapsulation mechanism to hide detailed interactions while focusing on the important message interchange
- It enables reuse of interaction descriptions.

### Symbol

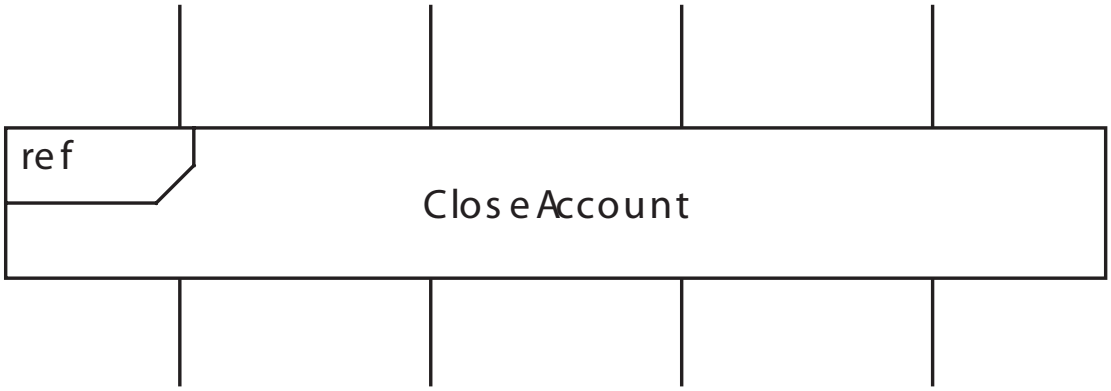



*Figure 58: Interaction reference*

### Syntax

The Interaction Reference symbol contains a name, referring to a use case, operation or other entity that can contain an interaction.

**See also**

"Interaction" on page 225

"Use cases" on page 218

"Sequence diagram" on page 224

"Attach/Detach from lifeline" on page 228

## Lifeline

A Lifeline represents an individual participant in an interaction. While Parts and structural features may have Multiplicity greater than 1, lifelines represent only one interacting entity. If a lifeline represents a part that has greater multiplicity than 1, a specific instance must be chosen through indexing.

### Symbol

The lifeline symbol consists of a head and an axis. If the lifeline has not been created yet, the axis is drawn by a dashed line. When a lifeline is destroyed (the instance is terminated), the axis is again drawn with a dashed line.

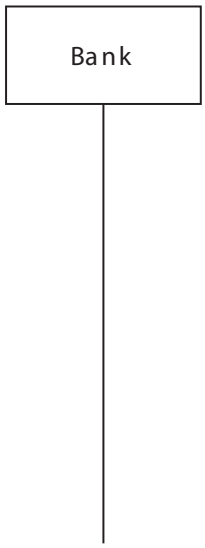



*Figure 59: Lifeline symbol*

### Create a lifeline

To create a lifeline you can either:

- Use the Diagram element toolbar and select a lifeline symbol. Place it in your diagram. Type in the appropriate information in the heading, for example **Partname** or **Classname**.
- Drag a class symbol from your model into the sequence diagram to create a lifeline representing this class. The lifeline will represent any instance of the class, text in heading symbol reading **Classname**.
- Drag a part from your model to create a lifeline representing this part. The lifeline will represent the instance of the part, text in head reading **Partname** (or **Qualifier::Partname** if scope qualifier is necessary).

### Attach/Detach from lifeline

When a symbol that can span over several life lines (such as the inline frame symbol) is selected there is a button next to each lifeline covered by the symbol. This button attaches or detaches the symbol from the lifeline the button is next to, depending on its current state. If the symbol is currently attached, the button display a minus sign (-). Otherwise the button display a plus sign (+).

### Ordering of events

The order of event occurrences along a Lifeline is significant, denoting the order in which these event occurrences will occur. The absolute distances between the event occurrences on the Lifeline are, however, irrelevant for the semantics.

Although the order of events is strictly specified on one lifeline, there is generally no ordering between events on different lifelines. It is possible to describe a distributed system using an interaction or a sequence diagram, so that each asynchronous component is described by its own lifeline.

The only mechanism to order events on different lifelines in the general case is to synchronize them by message sending. The ordering mechanism of sequence diagrams is often called **partial ordering**; they do not describe a total order, nor a complete disorder.

For systems that by nature are not asynchronous or distributed (normal programs, without threading), it is of course possible to have a stricter order interpretation than the general, asynchronous case.

### Lifeline decomposition

A lifeline can refer to a composite, that is to an object with parts. This is a way to reduce complexity of interactions and focus on the most important message interchange.

In some situations, though, you also want to see the internal communication, that is to say the detailed message interactions between the parts of a composite object. The decomposition mechanism offers this duality: it is possible to have two descriptions of the same behavior: one high-level description and one detailed. The detailed interaction is referenced in the lifeline heading and is defined in a separate use case or operation, as the example in Figure 60 on page 230 shows.

## Decomposition example

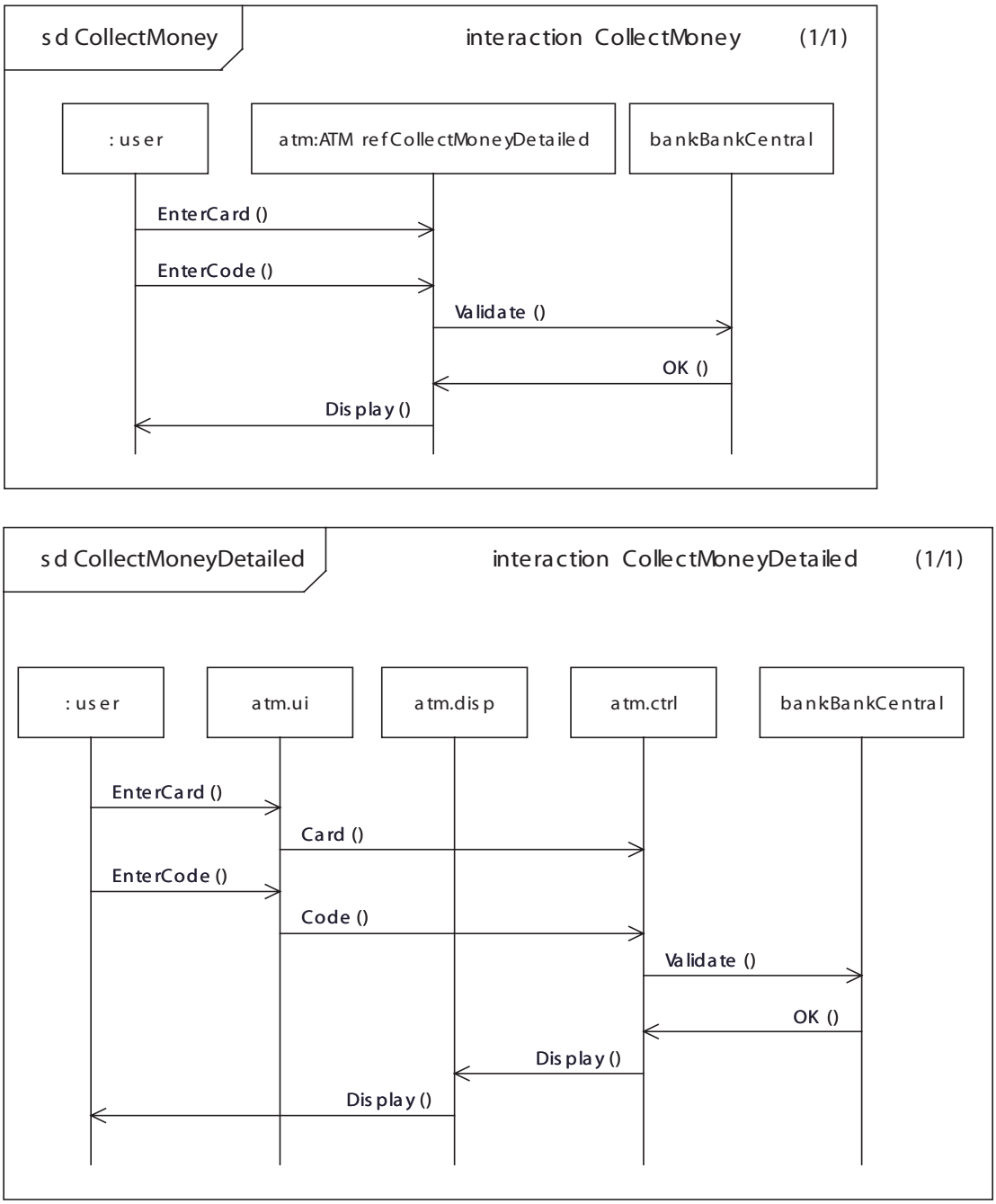



*Figure 60: Example of lifeline decomposition*

## Syntax

The following syntax is accepted in a Lifeline:

`Bank`
An instance name, referring to a Part, Port, Attribute or Subject.

`Bank: BankCentral`
An instance name and a type name, referring to a Class.

`:BankCentral`
A type name, referring to a Class.

`atm[3]`
An instance name with a selector expression to reduce the Multiplicity to 1 instance.

`atm.Display`
An instance name with an attribute referring to a part.

`atm ref OpenAccountDetailed`
An instance name and a lifeline decomposition, referring to a Use Case or an Operation, described in a separate interaction and Sequence diagram.

`atm[2].Display:ATM ref CloseAccountDetailed`
An instance name with selector, part, type and lifeline decomposition.

## Message

A message is an occurrence of a Signal, a method call, or a method reply. It normally has two events; one send event (out) on the sending lifeline and one receive event (in) on the receiving lifeline. A message can be horizontal or it can have a slope, but the receive event should not appear above the send event in the diagram.

### Symbol

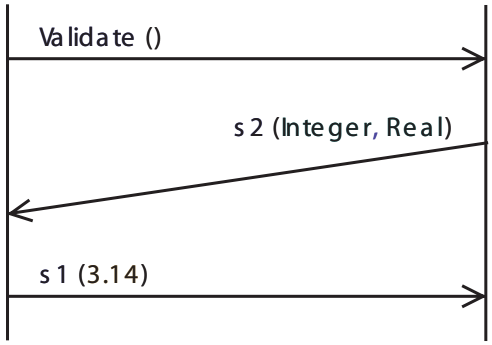



*Figure 61: Messages*

It may take time to send a message and pass it to the receiving side, but a slope does not have that interpretation. Correspondingly, a horizontal message is not necessarily directly delivered at the receiver.

Because of the relationship between signals and messages, the name of a message must always refer to a visible Signal in the model. If the signal has parameters, the message should have actual parameter expressions.

### Creating a message

Messages have three associated text fields, one for signal name and parameters and two for Gate names.

There are two different methods for placing messages allowing you to create messages in a simple and unrestricted way.

**Traditional:** Click on **Message line** in the Diagram element toolbar. Click on the sender lifeline, then on the receiver lifeline.

**Single-click**: Click on **Message line** in the Diagram element toolbar. When you **click and hold** between lifelines the message will attach to the lifeline to the left. When you **release** the lifeline will attach to the lifeline to the right. You can now do several things to create the message you aim for:

- Release to attach the receive point to the lifeline to the right
- Drag to cross any intersecting lifelines, release close to the left of the lifeline you want to receive the message.
- SHIFT + click to send the message from right-to-left.

Summary of how different line types can be created:

- **Normal message**: Select message in the toolbar. Click between lifelines for left-to-right direction. SHIFT + click for right-to-left direction. Click and hold, then drag to cross intersecting lifelines, release to attach next lifeline in message direction.
- **Message to self**: Select **Message line** in the element toolbar. Then click twice on the same lifeline.

### Reference existing signals when you draw a message

1. Click on **Message line** in the Diagram element toolbar.
2. Point and click on the lifeline that the message should go from.

3. Point and right-click close to the lifeline that the message should go to. Point to Reference existing on the shortcut menu and select the signal from the list.

Reference existing will display the signals visible in the scope.The signals that are shown are computed as follows:

- If the target lifeline has a type, then the signals/operations shown in the list are all signals that can be received by this type taking into account signals in realized interfaces, signals defined in the class itself etc.
- If the source lifeline has a type, then the signals/operations shown in the list are all signals that can be sent by this type, taking into account all required interfaces.
- If the source and target lifelines do not have types, but the target lifeline has a selector then the signals/operations shown in the list are all signals that can be received by the type of the selector taking into account signals in realized interfaces, signals defined in the class itself etc.
- If the source and target lifelines do not have types, but the source lifeline has a selector then the signals/operations shown in the list are all signals that can be sent by this type, taking into account all required interfaces that exist.
- If none of the above conditions apply, the signals/operations shown in the list are all signals visible from the lifeline itself.
- If none of the above mentions conditions apply, the signals/operations shown in the list are all signals visible from the lifeline itself.

A message can in some cases be drawn so it is only connected to one lifeline. This is particularly useful when using sequence diagrams for tracing. There are four message types that can be identified:

- **New**, the message is sent but not yet received. The message is connected to its sender.
- **Lost**, the message is sent but will not be received. The message is connected to its sender and a small circle is drawn at the message arrowhead.
- **Old**, the message is received but the sender is so far unspecified. The message is connected to its receiver.
- **Found**, the message is received but the sender is unknown. The message is connected to its receiver and origins from a small circle.

Use the property editor to mark a message as Lost or Found.

A message line can also be auto created in the following ways:

- SHIFT + click on the message in the symbol element toolbar when a lifeline is selected creates a new message. The new message is placed last on the lifeline, but before any destroy lifeline symbol.
- SHIFT + click on the message in the symbol element toolbar when two lifelines are selected creates a normal message between the lifelines. The normal message is horizontal, placed last on the lifelines (before any destroy lifeline symbol), and has a left to right direction.
- SHIFT + click on the message in the symbol element toolbar when a message is selected creates a normal message immediately below the selected message. The normal message is connected to the same lifelines as the selected message and has the same direction.

**Note**

*When you edit a message, you will see all parameters for that message, independently of whether parameters are shown or not. When you leave editing mode, the message text will go back to showing parameters or not in the same way as other messages do*

### Toggle parameters

Hides or shows all message parameters in the diagram. As default, parameters are shown. When you enter edit mode for a message text, the parameters will be shown for all messages.

### Incomplete message

A message may be incomplete in the sense that only one of its events is specified. If the receive event (in) is missing, it is a Lost message. If the send event (out) is missing, it is a Found message.

### Lost message

A lost message is a message where the send event is known, but there is no receive event. This can be used to describe the case when a message never reaches its destination.

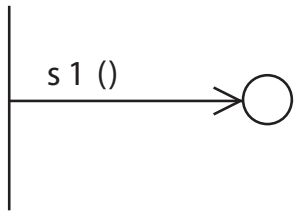



*Figure 62: Lost Message*

### Found message

A found message is a message where the receive event is known, but there is no (known) send event. This can be used to model the case when the origin of the message is outside the scope of the description. It can also be used to avoid over-specification: when several lifelines can be the sender, but which one is not relevant to the scenario.

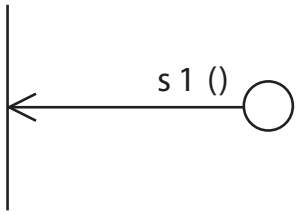



*Figure 63: Found Message*

### Copying a message

There are two different methods of copying messages. The first method always keeps message sender and receiver.

**CTRL + Drag**: Press CTRL key and hold it. Then click and hold a message you want to be copied. Drag the message and drop it to the new position. Release CTRL key.

The other method allows setting different sender and receiver.

**Copy and Paste commands**: Open the shortcut menu for a message you want to be copied by right-clicking on this line. Choose Copy from the menu. Open the shortcut menu by right-clicking in a place in the diagram where the new message should be inserted. Choose Paste from the menu. You can also

use CTRL + C and CTRL + V shortcuts for performing Copy and Paste commands, but note that the position of the new message is defined by the point in the diagram where you clicked last before pasting.

There are the following options.

- If the point you clicked is between two lifelines, then the new message will be inserted between these lifelines.
- If the point you clicked is either before the first lifeline or after the last lifeline, then the sender and receiver will be kept as in source message.

## Timer event

A timer is normally described by two distinct events in an interaction. The first event is the timer set, the second event is either a time-out or a reset.

A timer needs to be declared, before it can be used (just as messages need the corresponding signals or operations to be declared). Timers are declared with the Timer symbol in class diagrams or using textual syntax in a text symbol or text diagram.

The timer event symbols have one text field, for name and parameters.

### Timer set

The set event creates a timer instance, which now is active. The timer set event maps to the Timer set action.

### Timer reset

The reset event cancels an active timer. The timer reset event maps to the Timer reset action.

### Timer timeout

The timeout event occurs when the timer duration has passed and the timer signal has been received and consumed by a state machine. The timeout event maps to a timer signal consumption.

### Symbols

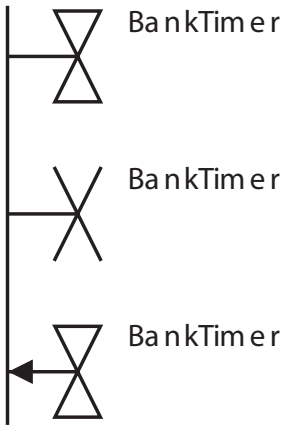



*Figure 64: Timer set, reset and timeout symbols*

**See also**

"Timer" on page 287

"Timer set action" on page 349

"Timer reset action" on page 349

## Time specification line

The Time specification line is used to create an **Absolute time** line, a **Relative time** line and a **General ordering line**.

### Absolute time line

An absolute time line can be added to the left or right of a lifeline, specifying an absolute time or a range, "`{<Time>}`". The line can be moved up or down along the lifeline. An absolute time line is created by clicking in the symbol palette on **Time specification line**, and by drawing a line connected to a lifeline in only one end.

### Relative time line

A relative time line is created by clicking in the symbol palette on **Time specification line**, and by drawing a line connected to the same lifeline in both ends.

A specific time duration observation, "{<Duration>}", or a time duration constraint, "{<Duration>..<Duration>}", can be specified in the text field.

A relative time line has an upper border, a lower border and a duration. The line is always drawn on the right side of a lifeline, but can be moved to the left side. The borders can be moved up or down along the lifeline.

In most cases, the start and stop events of a relative time line are connected to other events of the lifeline. For instance:

- The arrival of a message
- The sending of a message
- The start/top of a reference symbol
- The end/bottom of a reference symbol

It is allowed to place a Relative time line start or stop at a place where the event is not connected to other events.

### General ordering line

The general ordering line is a time specification line going between two lifelines. Create a general ordering line by clicking on **Time specification line** in the symbol palette, and by drawing the line between two lifelines.

The general ordering line is used to specify the order of events on different lifelines without using message lines. It is visualized as a dashed line, with a filled arrow in the middle. No text is normally associated with the line, but it is possible to associate a specific duration, "{<Duration>}", or a range, "{<Duration>..<Duration>}" with the line.

### Symbols

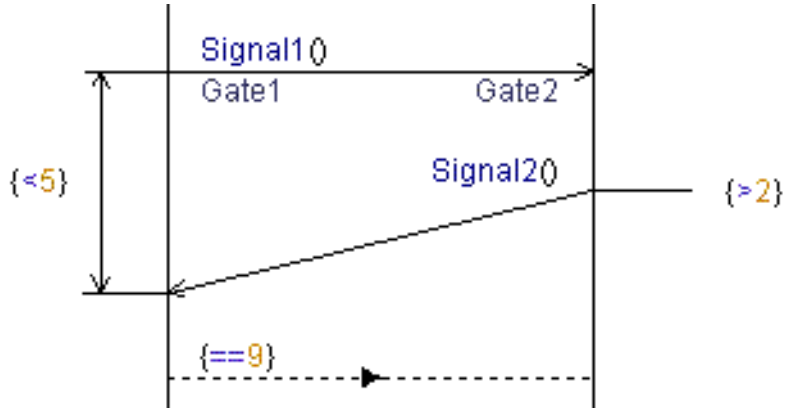



*Figure 65: Absolute time line, Relative time line and General ordering line*

## State

The state symbol is used to indicate that the instance described by the lifeline is in a specific state.

### Symbol

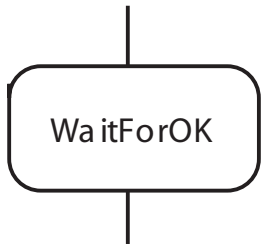



*Figure 66: State*

In scenario specifications, the use of the state is mostly done to highlight a certain state. Normally, you do not indicate all passed states along the life-line.

The State will bind to a model element if the state machine of the active class that the lifeline references has a state with the same name.

For traces, though, each state symbol maps to a specific Nextstate occurrence in a state machine transition. This is true if the lifeline object only has one main state machine; for active objects with parts, that is active objects that have several state machines, a simple mapping is not feasible.

# Action

The action symbol is used to express events that occur in a lifeline. It corresponds to an action symbol in a State machine. Informal statements must be written as comments.

### Symbol

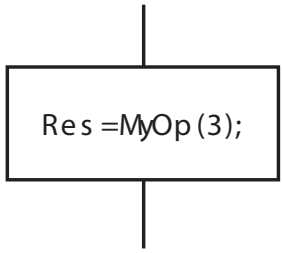



*Figure 67: Action*

The allowed textual syntax in the Action is the same as for the Action (task) symbol in state machine diagrams.

# Create

The Create event corresponds to the New operation applied on active classes.

The lifeline that is created is dashed before the reception of the create event, meaning that it has yet to be created. The name on the create line is the name of the class corresponding to the lifeline.

### Symbol

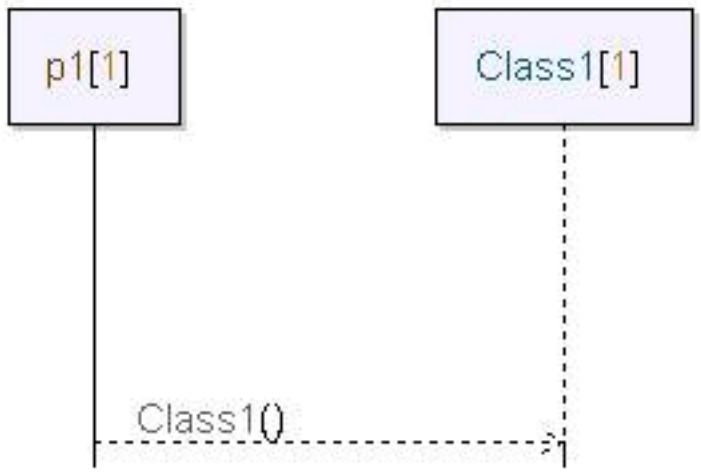



*Figure 68: Create line*

### Creating a Create line

When drawing a lifeline representing a dynamic instance of a class it is possible to draw the create event. This is done with the Create line button in the diagram element toolbar and is handled much like a message. The name of a create line is the name of the class corresponding to the lifeline. It refers to a constructor operation for the class. A create line have three associated text fields, one for the constructor operation name and parameters and two for Gate names. Formal parameters can be added similar to adding of operation parameters to a method call line.

### Binding of a constructor

Binding of a constructor initializer reference to a base class constructor fails if the base class constructor is called initialize. The recommendation is to name it to the same name as the class.

**Example 20: Constructor initialize that does not bind**

```
class AutoDispatchableClass : tor::DispatchableClass {
  initialize(tor::DispatchableClass d) {
    d.addToCurrentDispatcher(this);
    init();
    'start'();
  }
}
```

```
class MyClass : AutoDispatchableClass {
  initialize(tor::DispatchableClass d):
AutoDispatchableClass(d) {  }
}
```

The `AutoDispatchableClass` reference does not bind.

---

## Destroy

The Destroy event represents a termination of the instance. It corresponds to the Stop action in a State machine. Events can not occur on a lifeline after the destroy event.

### Symbol

*Figure 69: Destroy*

## Inline Frame

The inline frame symbol provides a way to group messages that should be treated similarly within an interaction. This means that it is possible to express different kinds of variations in a single diagram rather than having to create a new diagram for each possible variation.

### Symbol

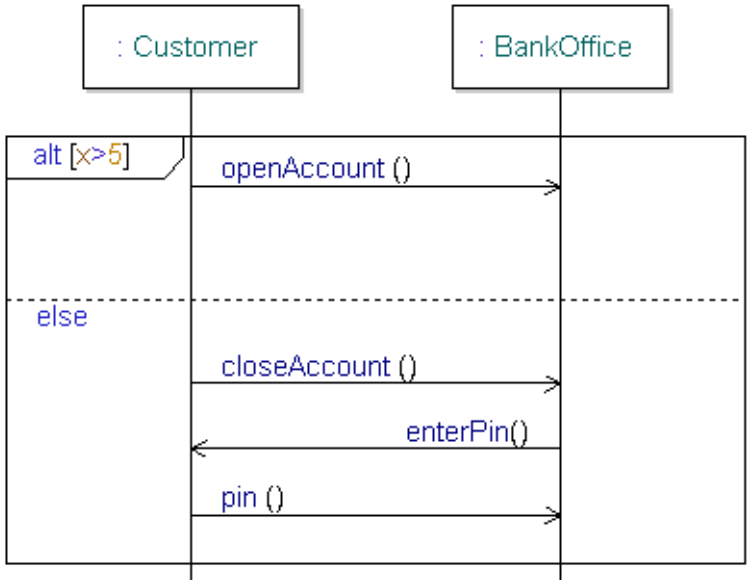



*Figure 70: Inline frame*

It is possible to have inline frame symbols inside other inline frame symbols. When a second inline frame symbol is added at the same height as an existing inline frame symbol, this will place a new inline frame symbol inside the existing inline frame symbol.

An inline frame symbol can have one or several inline frame sections. The default inline frame symbol has one inline frame section. Inline frame separator lines divide the inline frame symbol into several inline frame sections. Each inline frame separator line has a constraint text.

An inline frame separator line is created with a line handle that appears when an inline frame symbol is single-selected.

You can drag the inline frame separator line up or down within the symbol, but it is not possible to cross another separator line connected to the same inline frame symbol. An inline frame separator line can be deleted by selecting the separator line and pressing the delete key.

When a section is removed, objects in that section will also be removed as they are a part of the removed separator.

If the inline frame symbol is deleted, the contained objects are deleted with it.

The inline frame symbol has one text, which is a combination of:

- Operator keywords: Examples: `seq` (default keyword), `alt`, `else`, `loop`, `assert`.
- Constraint text. Examples: "[a<3]", "`else`"

It is possible to assign a background color to the inline frame symbol. The color will be shown as diagonal colored lines in the background of the symbol.

### Variations

There are several different possible variations, where the frame is sometimes split to express alternative groups of messages. The available variations are the following:

- `alt`**:** Expresses one branch of an alternative, or a decision. The frame can be split into multiple operands, and each operand can be associated with a condition. Only the alternative branch whose condition is evaluated to true will be chosen. Exactly one of the branches may be an else branch.
- `opt`**:** Expresses that the grouped messages are optional, meaning they do not have to happen. An optional frame cannot be split. It is possible to associate the optional frame with a condition, in which case it behaves just like an alternative, where the second choice is empty.
- `loop`**:** Expresses that a set of messages should be repeated a number of times. A loop frame cannot be split. The number of iterations is given using a minimum value and a maximum value of the format "`loop(min, max)`". It is possible to give "`max`" the value "*" which then denotes an infinite loop.
- `par`**:** Expresses that the messages of multiple operands can be interleaved with each other, or occur in parallel, but the ordering constraint within each operand must still be preserved. To be meaningful, a parallel frame must be split.
- `seq`**:** This represents the normal semantics of sequence diagrams, where each lifeline is independent of other lifelines. Weak sequencing is primarily used to override strict sequencing.

- `strict`**:** Expresses that the messages enclosed either in the sequence diagram or the combined fragment should have strict sequencing, that is to say that the vertical position in the diagram is equivalent to the order in which things will happen. Compare this with weak sequencing, which is the default for a sequence diagram, where each lifeline has its own timeline. When using strict sequencing, you can think of this as having a common global time for the involved lifelines.
- `neg`**:** Expresses that the set of messages represented are invalid.
- `critical`**:** Expresses that the enclosed messages cannot be interleaved by other inline frames. This can for example be used within a parallel frame to override the implied interleaving for a set of messages.
- `break`**:** Expresses an exceptional occurrence that interrupts the rest of the sequence diagram, and instead performs the set of messages enclosed by the break frame. A break frame cannot be split.
- `assert`**:** Expresses that the sequences expressed by the assert frame are the only valid ones, and that all other sequences are invalid. An assert frame cannot be split.
- `ignore`**:** Expresses that a given set of messages are insignificant and should not be shown within the frame. This gives a way to only show the most important messages of an interaction. The format is ignore {<list_of_messages>}. The converse operation is consider. An ignore frame cannot be split.
- `consider`**:** Expresses that a given set of messages are significant within the frame, and that messages not shown are thus insignificant. The format is consider {<list_of_messages>}. The converse operation is ignore. A consider frame cannot be split.

**See also**

"Attach/Detach from lifeline" on page 228

## Co-region

Symbol and lines can be connected to the lifeline in the normal way also inside the co-region symbol. When symbols are connected inside the co-region symbol, they are always covering the co-region symbol.

A co-region is used to indicate that the order in which elements on a single lifeline is insignificant.

### Symbol


*Figure 71: Co-region*

## Continuation

Continuations are only used in alternative inline frames, and acts as labels that decide how to continue from one part of a sequence to another. An alternative or interaction that ends with a continuation can only be continued in an interaction or alternative that starts with the same continuation.

### Symbol

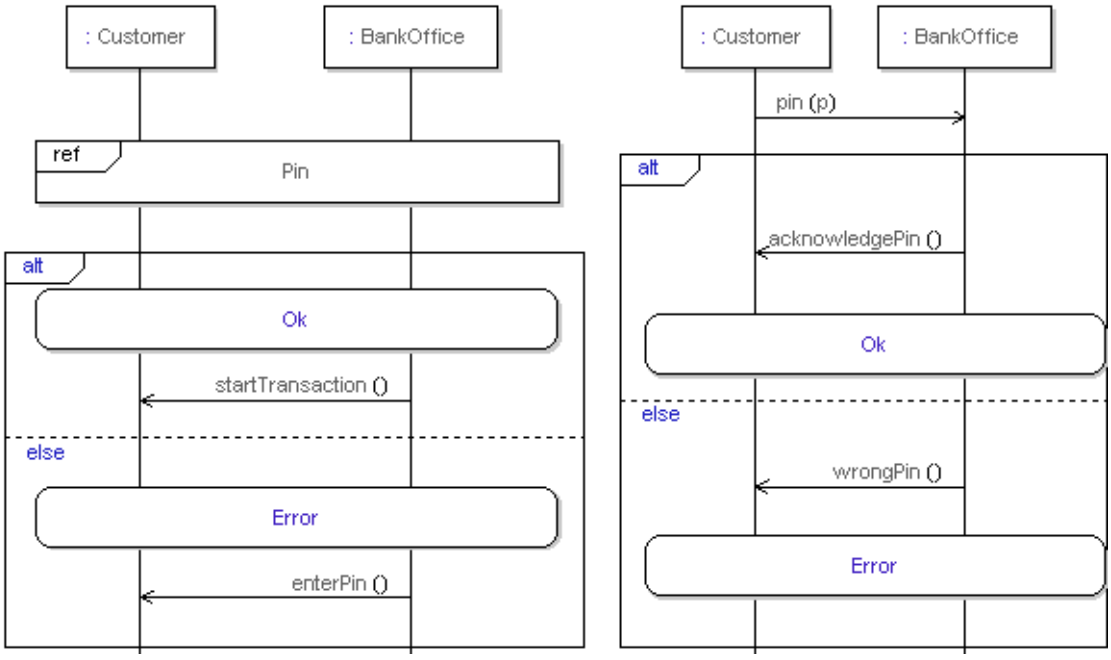



*Figure 72: Continuations*

The continuation symbol looks like the state symbol, but may span multiple lifelines.

The symbol contains a text field, located in the center of the symbol. The entered text is not parsed, just saved in the symbol.

It is not possible to place symbols and lines inside the continuation symbol.

**See also**

"Attach/Detach from lifeline" on page 228

## Method call

A method call is similar to a message, but is always synchronous. This means that it will always be associated with a method reply. Method calls are used to model for example how operations are invoked between different classes.

### Symbol

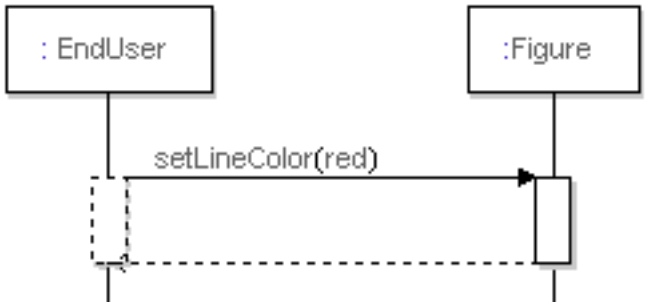



*Figure 73: Method call and reply*

A method call results in four graphical elements: a call, which is a solid arrow, a reply, which is a dashed arrow, an activation area, and a suspension area. The suspension area is a dashed rectangle on the caller lifeline, while the activation is a solid rectangle on the called lifeline.

The call message and reply message have three associated text fields each, one for operation name and parameters and two for Gate names.

To draw a complete method call:

1. Click on **Method call** in the Diagram element toolbar.
2. Place the **call message** start event on the lifeline that the method call should origin from, drag it to the receiver.
3. Type in operation name and parameter information or drag an operation from the model onto the message.
4. Edit the operation parameter type information in the call message and reply message name fields.

The main text of the reply message should normally refer to the same method as in the call message. The parameters may be different, for instance when the method has assigned values to out parameters, and there may also be a return value. It is allowed to only give a return value for the reply line: ": <value>".

Deleting a method call or reply line, connected suspension and activation area symbols are also deleted.

Deleting a suspension or activation area symbol, only the symbol is deleted, not any connected method call or reply lines.

When you drag a call or reply message the Method call symbol will be resized in the corresponding direction.

### Gate names

With the shortcut menu choice **Add/Remove Gate text**, gate names can be added to a message, method call or create event. The two gate name texts are placed below the line. When a gate text is activated, the gate gets a default name, which can be edited.

### Activation and suspension

The lifeline from which the method call originates is suspended while the receiver is busy executing. This means that it is not doing anything but waiting for a reply. The lifeline that receives the method call becomes activated while it is executing the method that is invoked. Once the reply has been sent back to the caller, both the activation and suspension areas are closed

# Update model

When the **Active Modeler** add-in is activated the shortcut menu will contain a new choice named **Update model**. This command is used on unbound entities, to create model elements in the current model to which they can be bound.

For sequence diagrams the following is supported:

### Sequence diagram

Updates the entire diagram by updating every element of the diagram.

### Interaction Occurrence Symbol

Creates a new use case with a sequence diagram in the same context as the use case that owns the reference. The name of the reference symbol is used to name the new use case. If no name is supplied, the default name will be used.

Qualified names are not considered. The qualifier will be part of the name of the new use case.

### Lifeline Symbol

If the type of the lifeline is unbound or if the lifeline is not typed, a class is created and the type of the lifeline is updated. If there is an unbound type it is used to name the class. If the lifeline is named, the name is used to name the class accordingly: "`lifeline name`" + "`Class`". If the lifeline is not named, a default name will be used.

If the lifeline has a name it is used to see if the lifeline has a corresponding attribute or not. If there is no corresponding attribute, a new one is created.

Update model is executed for each Message Line with this lifeline as source or destination. Update model is executed for all symbols owned by this lifeline, for example states and timers.

Qualified names are not considered.

### Message Line

Creates a signal matching the text of the message line. The types of the parameters are derived from the values on the message line. If a parameter is represented by an unbound identifier, a new attribute with the type Any and the same name as the identifier is created in the sequence diagram.

The signal is added to an interface if possible. The types of the lifelines are used to search for a suitable interface.

- If the types have ports with an interface that is realized and required in the correct direction, this interface is reused.
- If no matching interface is found, or if the lifelines are not typed, other message lines are searched for an interface.
- If there are other message lines between the same lifelines with a bound signal that belongs to an interface, this interface is reused.
- If no interface can be found in any of these ways, a new one is created.

In addition, if the source and destination lifelines are typed, their types are updated to be able to send and receive the signal. If they are passive, they are made active. If they do not have a port a new port is created. The port is updated to require or realize the interface containing the signal.

### Method call line

Creates an operation in the class typing the destination lifeline. If the lifeline is not typed, or the type is unbound, an error is generated. The parameters and return value of the operation is calculated by the values of the method call line. The reply line is used to calculate the return type.

If the operation is already bound and the owner is an interface, the operation is copied from the interface to the class.

### Method reply line

Checks if the operation has a return value and creates one if this is not the case. The type is derived from the value of the reply line. If there already is a return parameter and its type is different from the type of the supplied value, an error is generated.

The update will only work if the operation is bound.

### NextState Action Occurrence symbol

Creates a state in the state machine of the class typing the lifeline. The name of the state is the same as the text in the symbol. If the lifeline is not typed or the type is not bound, an error is reported. If there's no state machine in the class, a new one is created.

### Timer Set symbol

Creates a timer in the type typing the lifeline that owns the Timer Set Symbol. The name of the timer is the same as the text on the timer symbol. If no name is given the default name is used. If the lifeline is not typed, an error is reported and the model is left untouched.

Timer parameters and default duration are not considered.

### Timer Timeout symbol

See Timer Set symbol.

### Timer Reset symbol

See Timer Set symbol.

## Appearance and filtered delete

### Compress Layout

The Compress Layout button will compress the distance between messages and lifelines to be as defined in the tool **Options** for sequence diagrams.

When the Compress Layout button is pressed the lifelines are compressed and lined up by moving lifelines in the horizontal direction.

When the Compress Layout button is pressed together with the SHIFT button the lifelines will be compressed as described above, and objects on lifelines are also compressed, by moving these objects up or down along the lifelines.

When you press and hold CTRL and press **Compress layout** the lifelines are reordered to have the lifeline with the first event (for example a signal sending) to the left in the diagram.

When you press and hold SHIFT + CTRL and press **Compress layout** the lifelines and objects on lifelines are compressed as described above, and lifelines are reordered to have the lifeline with the first event (for example a signal sending) to the left in the diagram.

### Delete selected signals

Deletes the selected messages. This command will also delete messages using the same signals as the selected messages. Can also be used to delete other objects:

This command will delete all <X>, when <X> is selected.

<X> is one of:

- create line
- state symbol
- timer symbol (set, reset and time-out)
- time specification line (absolute time, relative time, general ordering line)
- method call (call line, activation symbol, reply line, suspension symbol)
- action symbol
- destroy symbol
- reference symbol
- inline frame symbol
- continuation symbol
- text symbol
- comment symbol

### Keep selected signals

If you press SHIFT and at point to Delete selected signals on the toolbar, the command will reverse the filtering effect: Only those messages that are selected, and those messages using the same signals as those messages that are selected, will remain in the sequence diagram. For other objects, there are the following rules:

This command will delete all <X>, if there is no selected <X> (<X> is defined in Delete selected signals).

### Make space

This command will make space below the selected symbol or line. Press SHIFT and point to **Make space** in the toolbar to remove space below the selected symbol or line.

## Interaction overview diagram

Interaction overview diagram is a form of Activity Diagram that focuses on the control flow between Interactions.

Interaction references in interaction overview diagrams can both define and reference operations/activities. Interaction reference is used instead of Action Node node and Object Node. An Activity edge and control constructs such as Decision, Fork and Activity Final nodes are the same as in activity diagrams.

The table below lists how you can represent the most common interaction operands listed in Variations in an interaction overview diagram.

| Operand | Interaction overview construct |
|---|---|
| `alt` | A Decision node matched with a corresponding Merge node. |
| `par` | A Fork node matched with a corresponding Join node. |
| `loop` | Decisions and graph cycles in the diagram. |

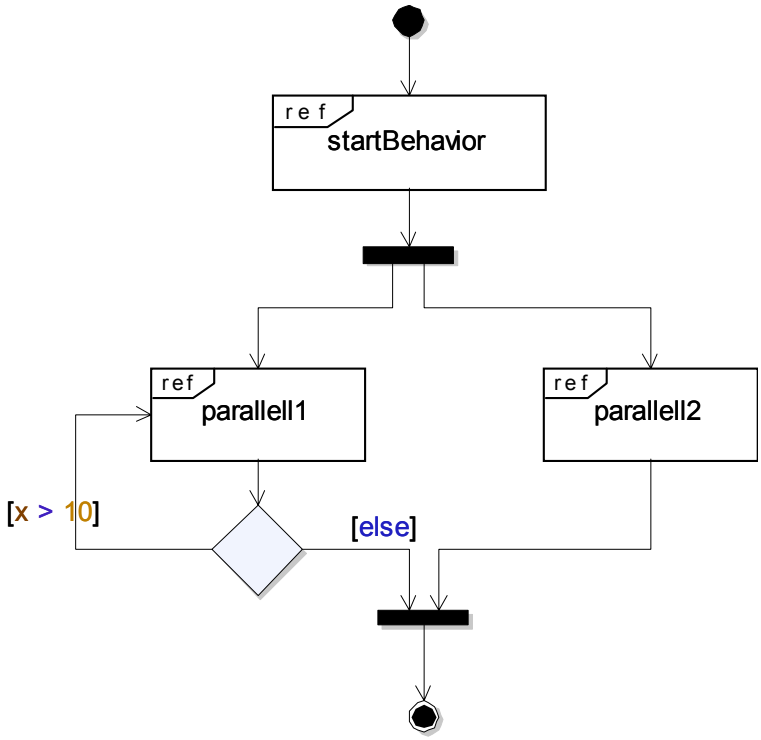



*Figure 74: Interaction overview diagram*

### Create an interaction overview diagram

Interaction overview diagrams can be included in classes and use cases.

1. Select the class (use case) in the Model View.
2. From the shortcut menu select **New** and then **Interaction overview diagram**.

### Model elements in interaction overview diagrams

The following model elements can be found in interaction overview diagrams:

- Decision
- Flow Final
- Fork
- Initial Node
- Join
- Merge

- Interaction reference, see Action Node
- Relationships

**See also**

"Sequence diagram" on page 224

"Activity Diagram" on page 311

# Package Modeling

When larger systems are to be modeled, the Package construct is vital for organizing all the different definitions into logical and manageable groups. A good principle for the organization is to group semantically close elements that are likely to change together.

## Package diagram

Package diagrams are used to visualize collections of Packages and the Relationships between them. It is used to model the breakdown of a system into logical packages and dependencies between these packages.

The package diagram contains packages and dependencies between these packages (for example Import and Access dependencies).

A Class diagram can be used for the same purpose.

### Example

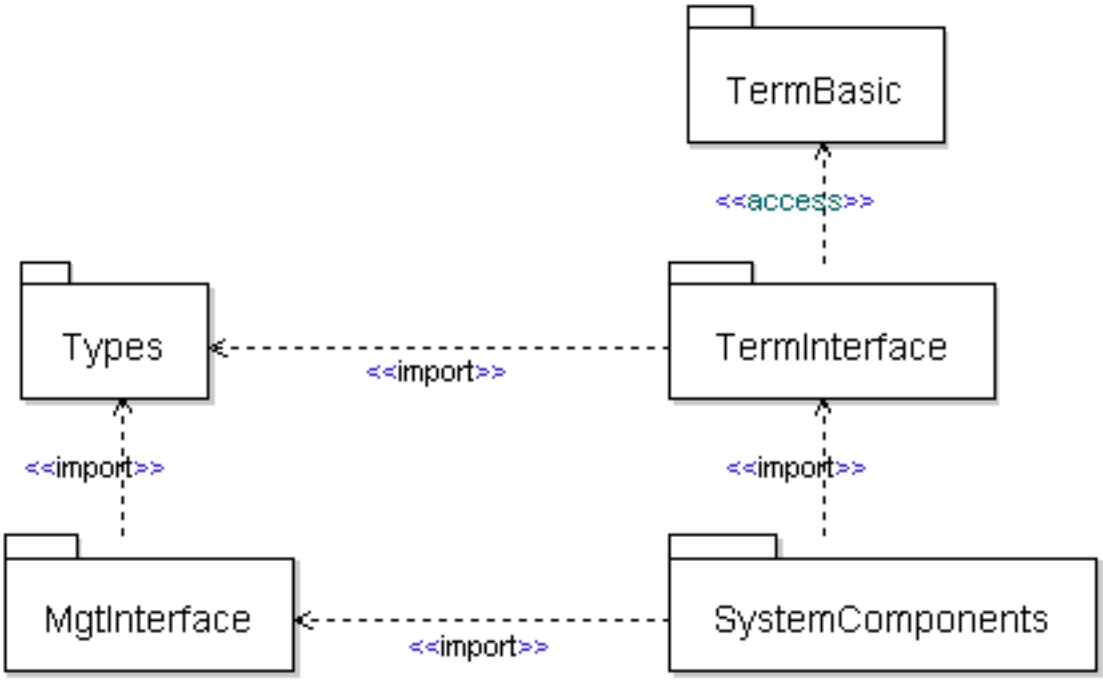



*Figure 75: Packages and their relationships*

### Model elements in package diagrams

The following model elements can be found in package diagrams:

- Package
- Relationships

**See also**

"Class diagram" on page 263

## Package

A Package is a mechanism for organizing elements into groups. A package provides a namespace for the grouped elements. Within the package, those elements can be referred to directly using their names, but from outside the package it is often necessary to qualify the names of the model elements.

A model normally consists of several packages that depend on each other. Understanding how packages relate to each other is critical when modeling systems of any complexity, but the larger the system becomes, the more important this activity becomes since it is often a reflection of the system architecture.

### Symbol

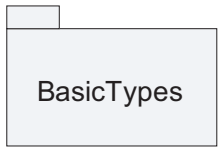



*Figure 76: Package*

Packages also let you control the visibility and access rights to the individual elements defined in the packages.

- Definitions (such as classes and other packages) may be collected in a Package.
- A Package may be imported or accessed by another Package.

It is possible to nest other symbols hierarchically inside a package symbol. An element created inside a package symbol will have the package as owner.

### Syntax

The package symbol contains a text field with the name of the package. When the referenced package is defined in another namespace the package name is preceded by a qualifier, like in “OuterPackage::MyPackage”.

**See also**

“Relationships” on page 257

## Relationships

The following Relationships can be used in package diagrams. These are described further in the section Relationships in UML.

- Dependency
- Containment

A dependency is often stereotyped to give a more precise meaning to the dependency. Two common stereotypes used for that purpose are the <<import>> and <<access>> stereotypes described below.

### Import

Import is a special kind of Dependency that is valid in particular between Packages, but also from for example Classes or State machines to packages. Its role is to import the names of definitions from a package into the current namespace, which is usually also a package. This provides a means to avoid having to use qualifiers. Names of definitions in a package P that has been imported by another package Q are automatically included in packages that in turn import or access package Q.

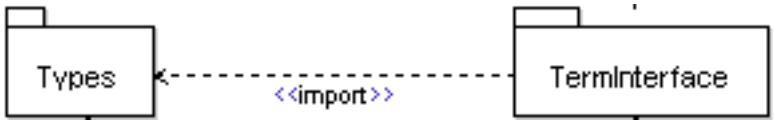



*Figure 77: Import*

**Note**

*Be restrictive with using import dependencies, as the set of names that become accessible without qualifier in the importing scope can become very large. It is often better to use access dependencies. If only a small subset of definitions shall be used the use of qualifiers should also be considered. Although qualified names mean more typing, it becomes very clear for all readers of a model which definition that is used.*

### Access

Access is a special kind of Dependency that is valid in particular between Packages, but also from for example Classes or State machines to packages. Its role is to import the names of definitions from a package into the current namespace, which is usually also a package. This provides a means to avoid having to use qualifiers. Names of definitions in a package P that has been accessed by another package Q are not included in packages that in turn import or access package Q.

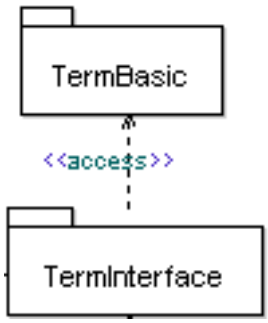



*Figure 78: Access*

An import is very closely related to an access; the distinction is primarily that an import is transitive, meaning that if a package is accessed or imported, you automatically also get the names of the definitions that are in turn imported by that package, but not the names of the definitions that are in turn accessed. Looking at Figure 75 on page 256, the package **TermBasic** is accessed by the package **TermInterface**, meaning that it is possible to refer to the names of definitions in **TermBasic** directly in **TermInterface**. However, these names are not directly available in package **SystemComponents**, which imports package **TermInterface**. In **SystemComponents**, it is therefore necessary to either explicitly access or import package **TermBasic** to refer to those names or explicitly qualify the names.

From an architectural point of view, accesses are preferred over imports since they force you to consider all the packages that you need, and will not bring in excess baggage by accident.

**Note**

*It is not necessary to import or access a package to be able to reference definitions within it. As long as the definitions are public, they can be referenced using qualification, for example "*`TermBasic::Xterm`*" can be used to reference the element* `Xterm` *in package* `TermBasic`*. For understandability, however, it is usually a good idea to produce a description of how packages depend on each other.*

**See also**

"Relationships in UML" on page 375

## <<noScope>> Packages

A «noScope» package is typically used when there is a need to divide the elements of a package into more than one file. However, it can also be used as soon as there is a need to structure the contents of a package into different parts but when the package from a UML name scope point still should be viewed as one entity.

Semantically a package stereotyped by the «noScope» stereotype will be as visible as any other package in the model view. It will also work as other packages with respect to storing it in a separate file. From a semantic point of view all of the elements in the «noScope» package are considered to be part of the containing package. When referring to an element in a «noScope» package using a qualifier, the name of the «noScope» package should normally not be used as part of the qualifier. The «noScope» stereotype makes all definitions visible outside of the package without a qualifier. It is possible to use an explicit qualifier to resolve ambiguous cases.

**Example 21: «noScope» package**

```
package A {
    <<noScope>> package B {
        class C {

        }
    }
    C c; // <<noScope>> makes C visible
}

package A {
    <<noScope>> package B {
        class C {

        }
    }
    class C { }
    C c; // class A::C hides class B::C
}

package A {
    <<noScope>> package B1 {
        class C {

        }
    }
    <<noScope>> package B2 {
        class C {

        }
    }
```

```
    B1::C c;
/* 'C' is an ambiguous name. B1::C or B2::C must be
used. If C is used without qualifier there will be name
resolution errors. */
}
```

## <<openNamespace>> Packages

In some situations it is useful to be able to incrementally define a package as the sum of a set of packages. Depending on what packages are loaded in a specific session the package will from a logical point of view have different contents.

This can in Tau be accomplished using «openNamespace» packages. In practise it works as follows: Define two packages in the same scope (for example as model roots). Give the package the same name and stereotype both of them with the «openNamespace» stereotype. From a semantics point of view the contents of the packages will now be merged. This implies that elements from one of the packages can directly be used in the other package without qualifier and also that the used names must be unique within all of the merged packages.

It is possible to have a hierarchy of nested «openNamespace» packages. So for example if you have an «openNamespace» Top containing an «openNamespace» Sub stored in one file you can have another file that also contains an «openNamespace» Top with an «openNamespace» Sub. If you load both of these files into the same project both the contents of Top and the contents of Sub will be merged.

The most important scenario when «openNamespace» packages are used is when you have a base version of a package hierarchy that is maintained separately but want to extend this, for example with a sub-package, when using it in a specific application.

# Class Modeling

Class modeling is the process of identifying the kind of objects that are part of the system being designed. This activity often takes place early in the design phase, or even in the analysis phase, typically after the objects that are part of the designed system have been identified (through use case and/or

scenario modeling). Objects that appear to share the same properties, behavior, and relationships with other objects are then grouped together and modeled as a class of objects.

Apart from identifying classes, the class modeling activity also involves the definition of these classes. This is typically done in a Class diagram. For each identified class, the following typical questions are answered:

**Does the class have structure?**
**What parts does an instance of the class contain?**

The structure of a class is described in a class diagram by means of attributes, and relationships such as generalizations and associations. A composite structure diagram can also be used to show how a class is composed.

**Does the class have behavior?**
**Which operations are available?**

The behavior of a class is perceived as operations on the class, and the signature of these operations are described in a class diagram. The same goes for other behavioral features of the class such as signals, timers or state machines.

**Which relationships exist between the class and other elements?**

A class may have relationships not only to other classes, but also to interfaces, datatypes, choices, etc. In the section "Relationships in UML" on page 375 you will find information on how to use them in class modeling.

**Is the class active or passive?**

Simply put you may say that an Active class defines dynamic event-triggered behavior and a passive class handle information. An instance of an active class has the ability to dispatch events.

**Which communication ports does the class expose to its environment?**

The ports of a class may be visualized in a class diagram.

# Class diagram

A Class diagram gives a static view of the model and is used to describe the types of the objects in a model. These types are typically Classes, but could also be other classifiers such as primitive, enumeration, interface, choice or syntype. A class diagram may also show relationships between the types, and their structural and behavioral features.

The definitions that are shown in a class diagram will by default be contained in the scope (for example a class or package) that owns the diagram, but it is also possible to show definitions from another scope.

In "Package Modeling" on page 255, information is provided on how to use package diagrams as a means for describing the packages of a system and how they depend on each other, but the same information can alternatively be described in a class diagram.

## Example of class diagram

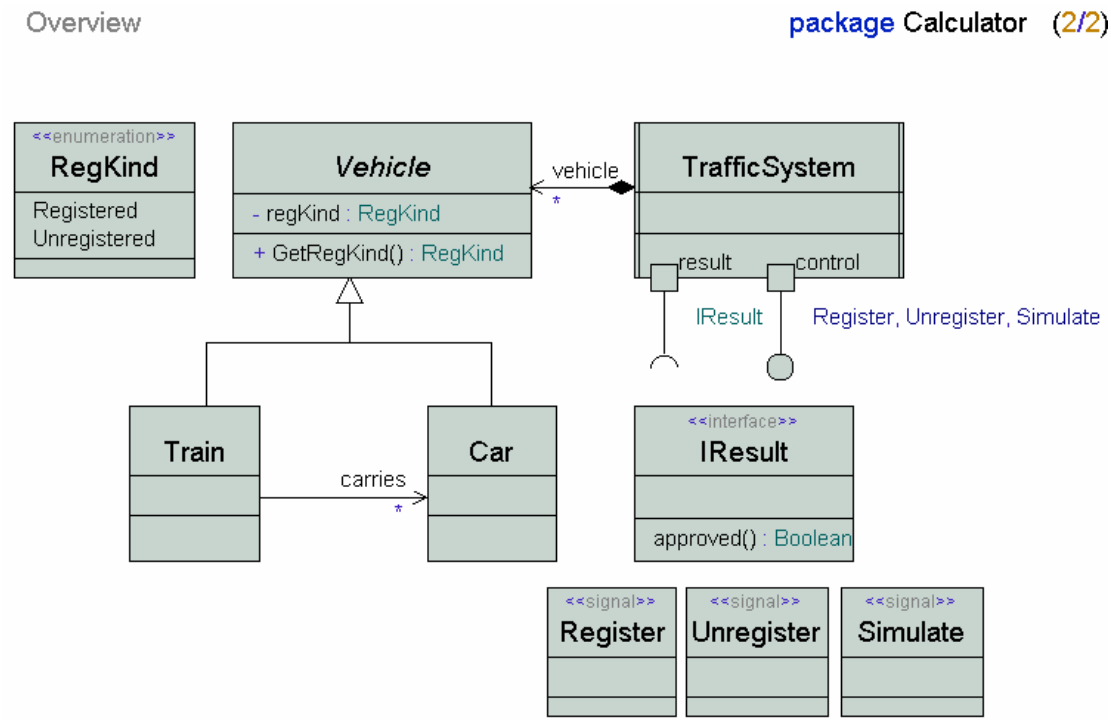



*Figure 79: Class diagram*

## Model elements in class diagrams

The following model elements can be visualized in class diagrams:

- Artifact
- Collaboration
- Class
  - Active class
- Attribute
- Operation
- Port
- Interface
  - Realized interface
  - Required interface
- Signal
- Signallist
- Timer
- Datatype
- Choice
- Syntype
- State machine
- Relationships

**See also**

"Package diagram" on page 255.

## Class

A Class is an abstraction of a group of objects that share the same properties (attributes), behavior (operations), structure, and relationships. A class may be instantiated (as long as it is not defined to be abstract) into a number of instances, all of which share the same properties.

### Symbol

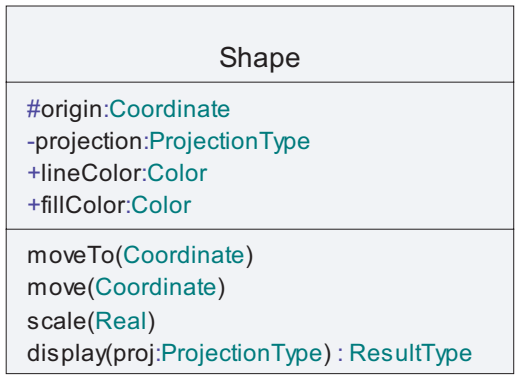



*Figure 80: Class with Attributes and Operations*

If instances of a class will maintain their own thread of execution (run concurrently with other instances), the class is said to be an Active class. If not, the instances will execute in the thread of another active instance, and the class is then said to be Passive.

To make a class active either:

- In the diagram (or the Model View), right-click the class you want to set as active, then on the shortcut menu click **Active**.
- Open the Properties Editor for the class and select **Active**.

The active class is displayed in the diagram with double vertical border lines.

A class may also have an internal structure, visualized in a composite structure (former architecture) diagram with parts and connectors, that describe how it is structured from an internal communication point-of-view. It may also have a state machine (called Initialize or the same name as the class) that describes it from a run-time execution point of view. This state machine is the “main” behavior that will be scheduled for execution when the instance of an active class is created.

Furthermore, a class may have a set of Ports, which specify how instances of the class may be connected to other instances in the architectural description of the class. The ports may also be used to group sets of interfaces that are exposed to different stakeholders.

There are several ways to add classes to a model.

- A class can be added directly in the Model View of the workspace window. Select the scope where the class should reside and from the shortcut menu select New/Class.
- Draw a class in a class diagram. Create and open a class diagram, select a class symbol from the toolbar and place it in the diagram.
- A textual definition of a class can be inserted in a text symbol or in a text diagram.
- From a composite structure diagram: When double-clicking on an unbound part with no type name. This will allow you to create a new diagram describing the part in question, and this diagram will belong to an inline class created to the part. Possible diagrams are: class diagrams, composite structure diagrams, state machine diagrams, use case diagrams.

### Multiple state machines in an active class

You can insert any number of state machines in an active class. However the following applies:

- If one of the state machines is named **initialize** or has the same name as the class, this state machine is considered to be the main state machine of the class. This state machine is executed when an instance of the class is created. If you omit this state machine, the Start and Stop symbols will automatically be inserted in the state machine diagram during the build process.
- Other state machines in the active class must explicitly be called in order to be executed. For example, it is possible to use such statemachines as the behavior when defining composite states.

### Syntax

The class symbol contains compartments with editable text fields:

- Class Heading (required)
- Attribute (optional)
- Operation (optional).
- Constraint compartment (optional).
- Stereotype instance compartment (optional).

### Class heading

The following example shows different class headings.

**Example 22: Class heading**

A simple class:

```
myClass
```

A class including virtuality:

```
redefined myC
```

A class using template parameters:

```
MyParamClass < type T, Integer c >
```

### Attribute

**Example 23: Classes and attributes**

A class with an Attribute:

```
public A : Integer = 4
```

Attributes with multiplicity:

```
A: Integer [10]
B: Integer [3,>15]
C: Integer [*]
```

### Operation

**Example 24: Signal**

```
signal s (Integer, Real)
```

**Example 25: A method example**

```
private m( x: Integer) : Integer
```

### Abstract class

A class can be *abstract*. This means that it is not possible to create instances of this class. The class thus needs to be specialized before it can be instantiated.

If a class is abstract then the name of the class is shown using *italics* in the class symbol.

To make a class abstract either:

- In the diagram (or the Model View), right-click the class you want to set as abstract, then on the shortcut menu click **Abstract**.
- Open the Properties Editor for the class and select **Abstract**.

### Virtuality

Virtuality defines whether a class can be redefined or not. This is only applicable if the class is contained in another class.

### Visibility

The visibility of a feature of a class, typically an attribute or operation, defines if it can be accessed from outside the class where it is defined.

- **None**

  When no visibility is defined for a feature.
- **Public**

  This feature can be referenced from any place where its contained class is visible.
- **Protected**

  This feature can be referenced from any descendant (by specialization) of the class that defines the feature.
- **Private**

  Only the class that defines a private feature can use the feature.
- **Package**

  This feature can be referenced from any place within the nearest enclosing package from which its contained class is visible.

For more information about visibility, see Visibility.

### External class

To define a class as external:

Open the Properties Editor for the class and select **External**. The external property is only shown in the Properties editor.

### Classes and components

There is no specific concept for components representing abstractions, but it can be modeled in other ways.

Classes and components are very similar in UML. A Component is a subclass of Class in the Metamodel. They can both have attributes, operations, composite structure (what is drawn in composite structure diagrams), ports, interfaces, etc. The primary purpose of the component is to provide terminology, and to highlight those features that are most important in component-based modeling. This includes the ability to represent how the component is realized, and also to specify the required and provided interfaces of the component. Typically, the provided interfaces are realized by one of the realizing classifiers.

### Constraint compartment

It is possible to attach one or several constraint compartments to a class symbol, with the **Add Constraint Compartment** shortcut menu choice. A constraint compartment can also be attached to other class-like symbols, such as interface or stereotype symbols.

A constraint compartment is placed below the last visible ordinary compartment of the class symbol.

A constraint compartment is similar to a Constraint symbol, with one read-only "{}" text label and a main text label that is editable.

The shortcut command **Show Constraints as Compartments** will create and attach one Constraint compartment for each constraint owned by the model element corresponding to the class symbol that does not already have a constraint compartment below the class symbol. The shortcut command **Show Constraints as Symbols** will create and attach one Constraint symbol for each constraint owned by the model element corresponding to the class symbol.

### Stereotype instance compartment

It is possible to attach one or several stereotype instance compartments to a class symbol, with the **Add Stereotype Instance Compartment** shortcut menu choice. A stereotype instance compartment can also be attached to other class-like symbols, such as interface or stereotype symbols.

A stereotype instance compartment is placed below the last visible ordinary compartment of the class symbol.

A stereotype instance compartment is similar to a Stereotype instance symbol, with one read-only "«»" text label and a main text label that is editable.

The shortcut command **Show Stereotypes as Compartments** for class symbols will create and attach one Stereotype instance compartment for each stereotype instance applied to the model element associated with the class symbol that does not already have a stereotype instance compartment below the class symbol.The shortcut command **Show Stereotypes as Symbols** will create and attach one Stereotype instance symbol for each stereotype instance owned by the model element corresponding to the class symbol.

**See also**

"Datatype" on page 288

"Choice" on page 291

## Collaboration

The collaboration symbol behaves like a class symbol, including support for Icon Mode, but the collaboration symbol is not showing attributes and operations in the symbol.

## Attribute

An attribute is a structural feature that may hold one or several values at runtime.

Attributes are used for modeling several different, but related, constructs of the UML language:

- **Attributes**

  An attribute of a Structured Classifier is modeled as an Attribute. The instance of such an attribute is often called a *field*, and it may be referenced by using a Field Expression. There are also so called class-scoped attributes (also called static attributes). All instances of a class share the same value for a class-scoped attribute.

  In composite structure diagrams, attributes with composition aggregation are often referred to as parts, which is due to the particular nature of that view to show the hierarchical structure of a class.

  Attributes are also used to represent the ends of an association.

- **Local variables**

  A local variable of a state machine, operation or compound statement is modeled as an Attribute. Such an attribute may be referenced directly by its name, with a scope qualifier if necessary.

- **Constants**

  A constant is modeled as a read-only Attribute. The value of the constant is the Default value of the attribute. Typically constants are defined on package level, but it is possible to define a constant wherever an attribute can be defined. Constants may be referenced directly by its name, with a scope qualifier if necessary. As the name indicates, the value of the constant may not be changed once it has been set.

An attribute always has exactly one static type. This type is determined at the point of defining the attribute, and can be either:

- a class,
- an interface,
- a primitive or enumeration,
- a syntype,
- a delegate,
- or a choice.

Attributes are closely related to associations. A navigable association end and an attribute is in practice the same thing. This implies that it is possible to first define an attribute and then visualize this attribute in a class diagram as the role name of a navigable end of an association. The opposite is of course also possible: Start by defining an association with one navigable association end. Then visualize the association end in the attribute compartment of a class symbol as an attribute.

The navigability is necessary if you want to use a specific association end/attribute that it is associated with to make a call.

**Example 26: Navigability**

Given the classes A and B. You want to invoke an operation B.op() from the class A.

With an association with an association end name ("role name") 'b' in the direction from A to B you can make a call 'b.op();' only if the association is navigable.

---

Attributes can also be visualized as symbols in composite structure diagrams. Although this is allowed for all attributes of the containing class, this possibility is often only used for parts.

### Aggregation kind

If an attribute is typed by a class this implies that the values for this attribute will be objects, that is instances of the class. In this case the attribute can have different aggregation kinds that determine the lifetime relationship between instances of the class containing the attribute and the value instances:

- **None**

  There is no lifetime dependency between the instances of the two classes. This implies that the attribute will contain one or more references to instances of the value class.

- **Shared aggregation**

  There is no lifetime dependency between the instances of the two classes. However, informally one is considered to be "owned" by the other. In the attribute compartment a shared aggregation is indicated by the keyword "shared" before the attribute name as in "shared a:myclass". Some code generators may attach a specific semantics to shared but in practice it is rarely used due to its weak semantics, and it is normally better to use an association with no aggregation instead.

- **Composition**

  There is a strong part/whole relationship between instances of the containing class and instances of the value class. In practice this implies that there is a lifetime dependency between the two instances. If the containing instance is terminated then the contained instance will also terminate. Composition is indicated by the keyword “part” before the attribute name as in “part a:myclass”

**Note**

*A non-static attribute may hold a value only when its defining context has been instantiated. The possible defining contexts listed above for an attribute are instantiated differently. For example, a package is instantiated when it is used, and an event class is instantiated when it is invoked.*

### Default value

An attribute may have a default value specified as an Expression. If it does not have a default value, its value remains undefined when the defining context is instantiated until it is explicitly assigned.

### Port

An attribute that is typed by a Class may have communication ports to which connectors can be connected. These connectors describe communication paths in a system that convey signals to and from the attribute. This is mainly used when the attribute represents a part.

### Multiplicity

An attribute may have a multiplicity, modeled as a collection of ranges. It specifies a restriction on how many instances the attribute may hold at runtime.

Depending on whether the multiplicity of the attribute is >1 or not the actual type of the attribute is different. If the multiplicity is >1 then the attribute will have a container type that can hold a list of values. If the multiplicity is exactly 1 (or `0..1`) then this is not the case.

Depending on what datatype libraries are available the container type may be different. Typically different code generators will supply different container types to provide a suitable integration with the target language. If no specific datatype library is loaded the String type will be used in the built-in pre-

defined package as the type of attributes with multiplicity > 1. (The String type is a predefined collection type that represent an ordered list, or a sequence. The values in the list must adhere to the type of the attribute.)

In the attribute compartment of a class symbol the multiplicity is shown within brackets after the type of the attribute as in:

```
a : myClass [*]
```

In the above example, the multiplicity is unbound (represented by the asterisk), meaning that it can have any number of values.

If no multiplicity is given it is considered to be 1 by default.

### Initial cardinality

For a composite attribute with a multiplicity > 1 there is a shorthand that allows specifying the initial number of instances using an Expression. That number specifies how many instances that will be automatically created when the owning Class is instantiated. If an initial number of instances is omitted, exactly one instance will be created.

**Note**

*If and how the cardinality is interpreted is code generator dependent. Some code generators may ignore the cardinality of an attribute.*

The initial cardinality can be given if an attribute is shown using a part symbol in a composite structure diagram. However, in this kind of symbol the syntax is as in:

```
a : myClass [*] / 2
```

where the initial number of instances for ‘a’ would be 2.

### Visibility

It is possible to specify Visibility for attributes. This can be one of **public**, **private**, **protected** or **package**.

### Derived

An attribute can be declared to be `derived`. This indicates that the value of the attribute is not stored in the corresponding object, but instead is computed from for example the values of other attributes. The syntax for a `derived` attribute is a ‘/’ preceding the attribute name as in

```
/a:myClass
```

For more information on how to specify the derivation rule for a derived attribute, see Derived.

### Static

A static attribute is an attribute that is owned by the class scope rather than the instance scope. This means that there is only one Attribute instance that is shared by all the instances of a particular class.

### Constant

A Constant attribute is an attribute which value cannot be modified dynamically. The value of the constant is the default value of the attribute.

An external Constant attribute means that the value is defined outside of the model or at a later time (build time for example).

**Example 27: Textual constant declaration**

```
const Integer a = 10;
const Integer extern ext_const;
```

## Operation

An operation is a declaration that instances of a class will be able to handle calls that match the signature of the operation. An operation can be implemented either by an operation body or a state machine. This implementation (often called a method) will be executed when the operation has been invoked. This means that if the receiver is a passive instance, the implementation will be executed immediately after the operation has been invoked, while if the receiver is an active instance the execution of the implementation may be delayed and executed some time in the future when the instance is in a state where the operation call is accepted.

Operations can be declared textually in text symbols or text diagrams, in the operations compartment of a class symbol, and using a special operation symbol.

Tau supports a `derived` property for operations as an extension to standard UML. This can be used to indicate that the operation has no implementation but is implicitly computed. This property is for analysis only and will not affect any generated code.

### Symbol

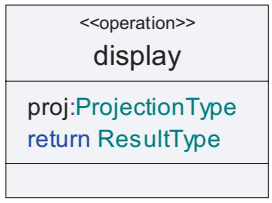



*Figure 81: Operation*

### Syntax

The symbol contains two editable text fields: Operation Heading and Parameters. The bottom field is always empty.

## Active class

An Active Class is a class with its own thread of control. It is distinguished from the normal (passive) class by the property Active. Graphically, this is indicated by the special Active Class symbol, as in Figure 82 on page 276.

### Symbol

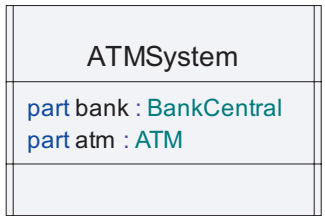



*Figure 82: Active Class*

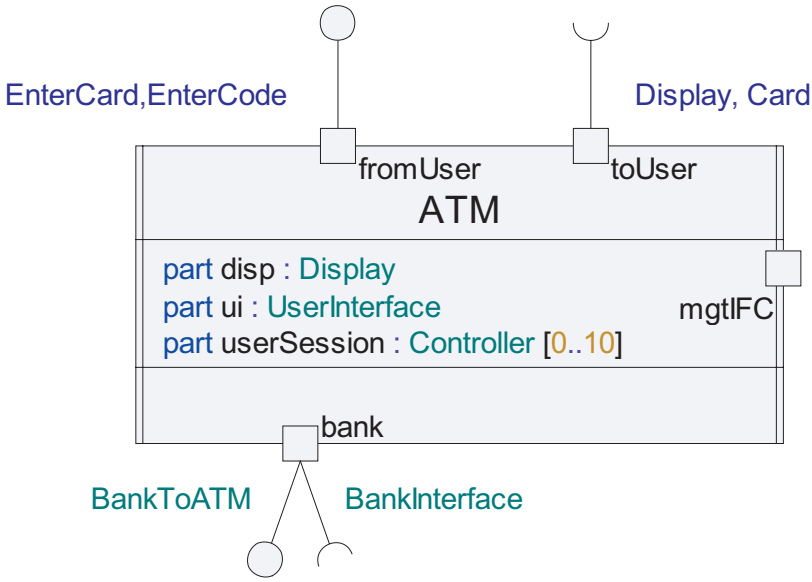



*Figure 83: Active Class with Ports and Realized and Required Interfaces*

You can make a class active by:

- Selecting it and in the shortcut menu choose **Active**.
- Selecting it and in the Properties Editor select **Active**.

The active class is the fundamental building block for modeling real-time behavior in UML. Active classes define both the structure (architecture) and behavior of a model. This duality of the active class concept in UML offers strong and flexible design capabilities.

### Structure

The structure of an active class is defined in one or several Composite structure diagram, which defines the active class as a set of instances of other active classes. These active classes can also have structure, thus enabling descriptions of complex architectures.

### Behavior

The behavior of an active class is defined by a State machine in one or several State machine diagram. This state machine should be named `initialize()`, or given the same name as the class.

In order for an active class to be completely specified, it must have either a structure definition, a state machine definition, or both.

An active class has its own flow of control and can both initiate behavior and passively react to behavior as observed on its interfaces. Traditionalists prefer the name *reactive* class instead of active class, since such classes are typically event-driven. The initiation of behavior is often done through the use of timers; at the expiration of a timer some behavior is kicked into gear.

When an active class has several contained parts defined in its composite structure diagram(s), each part executes asynchronously and concurrently with other parts in the system. This semantic ensures that the model can be deployed in a distributed physical environment and is not dependent on being run on a single processor with shared memory access.

An active class can realize and require interfaces via a Port. Ports together with their required and realized interfaces define the static contract between the active class and its environment.

### Attributes and operations

In the active Class symbol, it is possible to specify or show attributes of the class in the second compartment of the symbol and operations in the third compartment.

**See also**

"Attribute" on page 270

"Operation" on page 267.

## Port

Ports are named interaction points of an active class. They specify the implemented interface (realized) and the needed interfaces from other classes (required).

Ports are typically used only on active classes. To visualize an already created port on an active class symbol or a part symbol, use the **Show/Hide** command on the shortcut menu and point to **Show Ports**.

### Symbol

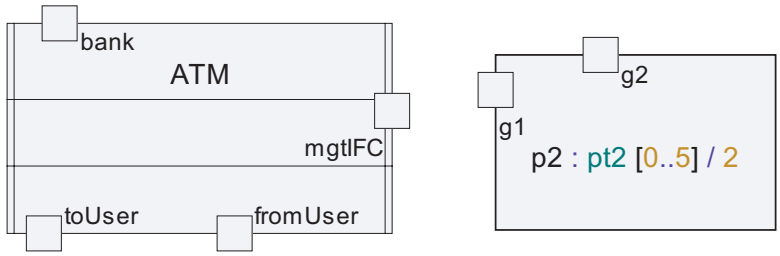



*Figure 84: Ports on a Class and Ports on a Part*

**Hint**

*The easiest way to attach a port symbol is to first select the frame of the symbol where the port symbol should be placed and then click the port symbol in the toolbar.*

### Port type

The symbol has one text field that should contain a name and that optionally may contain a **type**. The type of the port is mainly intended to be used in an analysis phase.

**Note**

*The code generators in Tau do not take the port type into account. Instead code is generated based on the information given for realized and required interfaces of the port.*

### Behavior ports

There are two different kinds of ports: behavior ports and non-behavior ports. The difference between these two different kinds of ports is that a behavior port is directly associated with the state machine of the class, whereas a non-behavior port needs to be connected using connectors and are typically only relaying the communication from outside the class to some of the internal parts of the class.

A Behavior port is a port that is directly connected to the state machine of the class. All signals sent to this port are consumed by the behavior of the class itself.

### Ports and interfaces

For each port, the realized and required interfaces may be specified. The realized interface of a port defines the incoming requests that can be handled via the port. The required interface defines the outgoing requests that must be handled by a class connected to the port from the outside via one or more connectors. In Figure 83 on page 277 you will find an example of ports with realized and required interfaces.

When defining the structure or behavior of an active class, ports can be declared on the border of a diagram used for this purpose (a composite structure diagram or a State machine diagram). Ports can also be referenced from parts, where they are shown on the border of the part symbol.

It is also possible to send messages through a port (without knowledge about possible receivers at the other end of the attached connector) from a state machine as an addressing mechanism.

The realized (or required) interface of a port may typically contain references to interfaces, but also to a signal list, signal or attribute.

The realized and required interfaces of a port are visualized by attaching the Realized interface symbol and the Required interface symbol to the port. On these symbols, the supported or needed Interface names (signal list, signal or attribute) can be specified.

Another way to specify the Realized and Required Interface of a Port is by the Properties Dialog.

Ports represent:

- Connection points for Interfaces to classes
- Connection points for Connector lines in Composite structure diagram, connecting instances of these classes with other instances or with the enclosing frame symbol.

The port symbol can be placed

- On Class symbols
- On Part symbols
- On Behavior symbols
- On the frame of a State machine that is owned by an active class
- On the frame of a composite structure diagrams

- Within Architecture and State machine diagrams (which has the same semantics as when the port is placed on the frame of these diagrams).

A port can have both explicit and implicit connectors. Each Port symbol can have zero, one or two interface symbols attached to it.

When you have two interface symbols, one of them should be defined as a Realized Interface symbol specifying the incoming interfaces (or signals) to the port and the other should be defined as a Required Interface symbol specifying the outgoing interfaces from the port.

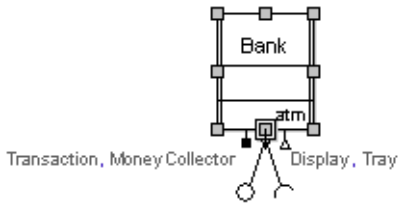



*Figure 85: A port with realized and required interfaces*

Ports with or without interfaces can be drawn directly on a class. Either:

- Select the class and hold down SHIFT while you click on the toolbar port symbol. Type the port name. The port will be positioned on the class' left border segment as close as possible to the upper left corner.
- Click on the toolbar port symbol, click on the class where you want to position the port. Edit the name text field.

### Inheritance

In case of a generalization between classes where there are ports belonging to the supertype these ports will also be inherited.

Ports can be declared public and private to distinguish if a port is externally exposed or if it is only used internally. It is possible to add more signals to ports in subclasses.

## Interface

An interface is a structured classifier that may not be instantiated. Instead, it is used for grouping a set of attributes, operations, and signals that must be implemented by the class that implements the interface. A class that imple-

ments an interface is said to **realize** the interface, thus supporting the operations declared in the interfaces. A class can also **require** interfaces, it is then dependent on other active class(es) in order to perform its operations.

### Symbol

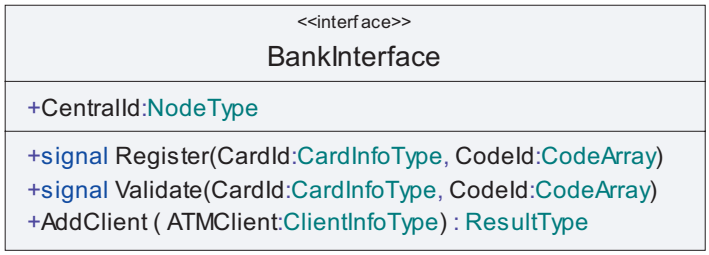



*Figure 86: Interface symbol*

The operations of an interface typically describe services that are offered by the class(es) that realizes the interface. Naturally, a class may realize more than one interface.

Apart from operations, an interface may contain signals and attributes. It may also contain other definitions, such as types.

An interface can be specialized and may have Template parameters. Multiple inheritance of interfaces is a useful mechanism to define the communication interfaces of active classes.

Interfaces can also be associated to each other to provide a definition of protocols or contracts between classes that realize the involved interfaces. An example is given in Figure 87 on page 283 that defines the `MgmI` and `MgmReplyI` interfaces. The association between the two interfaces establishes a relationship between them. This means that wherever one of the interfaces is referenced, for example on a port or associated with a connector, the other interface will automatically be inserted in the other direction. So, for example if a class realizes the `MgmI` interface via a port then the `MgmReplyI` interface will automatically be a required interface of the same port

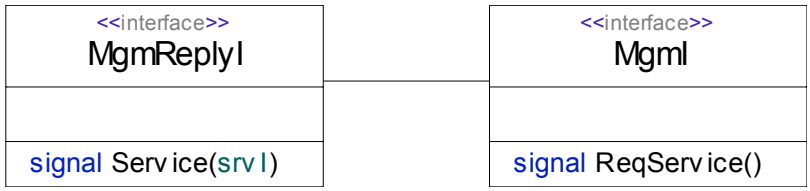



*Figure 87A contract defined using two associated interfaces*

**Syntax**

The symbol contains three editable text fields:

- Heading,
- Attribute, and
- Operation.

The heading field is used to define the name of the interface.

The attribute field contains definitions of attributes that must be implemented by classes realizing the interface. Typically, this is a shorthand for a getter operation and a setter operation to a protected attribute of the realizing class.

The operations field contains definitions of operations and signals that must be handled by classes realizing the interface.

See also

"Realized interface" on page 283

"Required interface" on page 284

"Pid" on page 363

## Realized interface

A realized interface attached to a port on a Class visualizes what interfaces the Class realizes through that port. Interfaces, signals, signal lists and attributes may be specified in the text field.

### Symbol

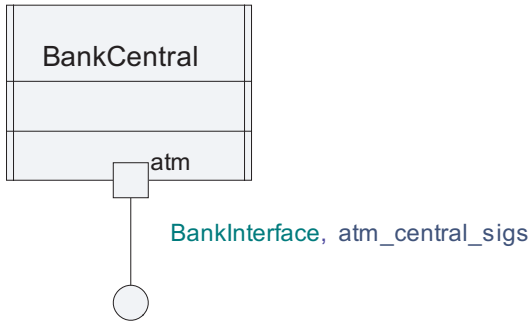



*Figure 88: Realized Interface*

### Syntax

The symbol contains a text field.

**Example 28: Realized interface**

```
S, p, SigList
```

See also

"Interface" on page 281

"Required interface" on page 284

"Pid" on page 363

## Required interface

A required interface attached to a port on a class visualizes what requests the class expects to be handled through the port. Interfaces, signals, signal lists, and attributes may be specified in the text field.

### Symbol

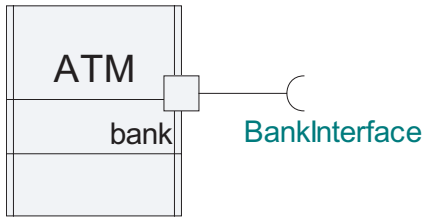



*Figure 89: Required Interface*

### Syntax

The symbol contains a text field.

**Example 29: Required interface**

```
S, p, SigList
```

See also

"Interface" on page 281

"Realized interface" on page 283

"Pid" on page 363

## Signal

A Signal is one of the primary means for communication in UML. A signal represents an asynchronous message that is sent between active classes. The signal can carry data, which must conform to the declared parameter types of the signal.

A signal is most conveniently declared together with other signals, operations and attributes in an Interface that represents the capabilities of the classes that realize the interface.

However, a standalone signal declaration can also be made using a special signal symbol similar to a class symbol as shown in Figure 90 on page 286.

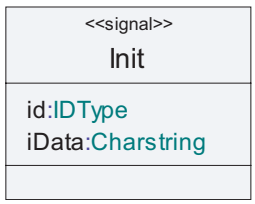



*Figure 90Signal*

If numerous distinct signals will be used, it is often more practical to declare the signals textually in a text symbol:

**Example 30: Textual signal declaration**

```
signal Init (IDType id, Charstring iData);
signal SetupReq, SetupInd, AbortReq, AbortInd;
signal ForwardedMsg (IDType, MsgData);
```

### Syntax

The signal symbol contains two editable text fields:

- Heading
- Parameters

The heading field declares the name of the signal and the parameters field declares the parameters of the signals. The name of the parameters may be omitted, but the parameter types are required.

The third compartment that exists for many class like symbols is always empty for signal symbols.

## See also

"Message" on page 231

"Signallist" on page 287

"Interface" on page 281

"Timer" on page 287

"None" on page 391

## Signallist

The keyword `signallist` is used to denote a group of related signals in order to make the description easier to comprehend. It is typically used in ports and connectors.

**Example 31: signallist declaration**

```
signallist MgtSignals = MOGetStatus, MOSet, MOReset;
```

**Note**

*Using an Interface to group signals together is a more structured approach, compared to signal lists, since the Interface also encapsulates the signal declarations.*

**See also**

"Signal" on page 285

"Interface" on page 281

## Timer

A Timer is an event that, in the same fashion as a signal, can trigger transitions. A timer is set by an implementation executed by an active class and at timeout, a timer event can be received by the state machine of that same active class instance. A time value is associated with an active timer, which is the time of the timeout.

### Symbol

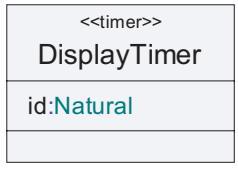



*Figure 91: Timer*

Timers can, like signals, have parameters. This can be used to allow to set more than one timer of the same kind without resetting the already active timer; that is several timers with different parameter values may be active at the same time.

### Syntax

Timers can also be declared textually in a text symbol:

**Example 32: Textual timer declaration**

```
timer DisplayTimer (Natural id) = 2;
timer BankTimer () = BankTimeout;
timer UserTimer ();
```

When declaring a timer textually, it is also possible to give the timer a default duration, that is a duration before timeout that allows to set the timer without specifying the duration.

### See also

"Timer handling and time" on page 365

"Timer set action" on page 349

"Timer reset action" on page 349

"Timer set" on page 236

"Timer reset" on page 236

"Timer timeout" on page 236

"Timer active expression" on page 362

## Datatype

Datatypes are used for two different purposes:

- To describe primitive types that are available
- To describe user-defined enumeration types

Primitive types are most often defined in model libraries that accompany specific UML profiles, either standalone profiles or profiles defined to be used together with specific code generators. In the latter case the datatypes typically define the target language primitive types and makes them available in UML models.

It is however also possible to define primitive datatypes in user models, but this may cause code generation problems.

An enumeration defines a set of values simply by enumerating them as a list of enumeration literals.

In any case the datatype may also optionally contain behavior that is defined by **operations**.

### Symbol


*Figure 92: Enumeration Datatype*

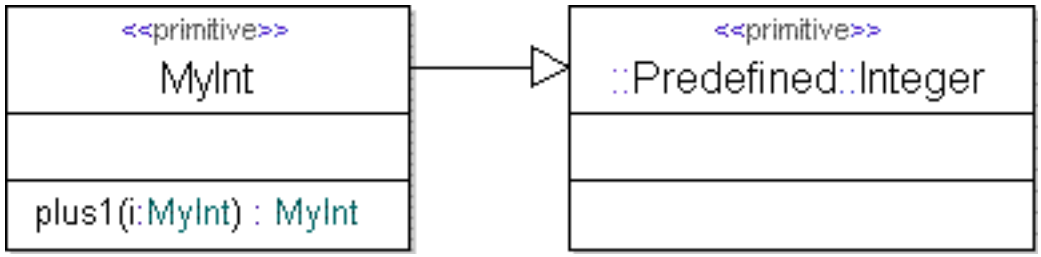



*Figure 93: Datatype with operator*

### Enumerated datatype

An enumerated datatype is a datatype where the literal values are logical names. The logical names can optionally be attached to an integral value specified by a simple expression.

The available default operations are:

- Equality (==, !=)
- Relational operations (<, >, <=, >=)
- Assignment (=)

**Example 33: Enumerated datatype**

```
enum UKColors { blue, red, white }
```

```
enum LinePrinterState {
  outOfService = 1,
  inServiceFree =2,
  inServiceBusy = 6
}

void op() {
  LinePrinterState e;
  Integer i;
  e = cast<LinePrinterState>(1);
  e = inServiceFree;
        i = cast<Integer>(e);
}
```

**Note**

*It is possible to convert between Integer and enumeration types using the cast operation as in the operation* `op` *in* *Example 33 on page 289*.

### Primitive datatypes

Primitive datatypes are usually defined in model libraries in profiles but can also be user defined. However, a user-defined primitive type will not have a literal syntax, which makes them less useful in practice.

There are however two ways to relate the datatype to another already existing datatype:

- use copy constructors
- use inheritance

In both of these cases the literal syntax of the existing datatype will be used. The copy constructor mechanism is the recommended mechanism to introduce new primitive datatypes in UML and this is what is used in most model libraries.

**Note**

*Primitive datatypes usually need special treatment in code generators. A user-defined primitive datatype is not likely to work in a code generator unless specifically stated in the code generator documentation.*

**Example 34: Datatype with operators**

```
datatype simpleInt {
  simpleInt(Integer) {}
}
datatype myInt : Integer
{
```

```
  myInt plus1 ( myInt i) { return i+1;}
}
```

### Literal

A Literal is a definition of an element of the type defined by an enumerated datatype. The literal is owned by that datatype. The visibility of a literal is always public.

Aside from having a name (which all definitions have), a literal may also have an integral value which allows it to be used in arithmetic expressions.

## Choice

A Choice is a datatype that can hold one value. This value can be of different datatypes during the execution. A choice of which type is made when assigning a value to a variable. For each potential type field, there is a boolean operator `IsPresent()` that can check if the field is present or not.

**Example 35: choice**

```
choice IntOrBool {
    Integer a;
    Boolean b;
}

IntorBool ib;
Integer i;
Boolean b=true;

ib.a=5;
i=ib.IsPresent("a")?ib.a:0; /* check if ib is Integer;
       if Integer, return ib,
       if not, return 0 */
ib.b=b;
```

**Example 36: choice**

```
choice IntOrBool {
  Integer a;
  Real r;
  Integer GetInt() {
    if (IsPresent("r")) {
      return 0;
    } else {
      return a;
    }
  }
}
```

Using the `IsPresent()` operator:

```
IntOrBool MyVar;
Real num_real;
Integer num_int;
MyVar.a=1;
if(IsPresent(MyVar,"a"))
  {
    num_int =MyVar.a;
    MyVar.r=3.14;
  }
  else
  {
    num_real=MyVar.r;
  }
if (MyVar.IsPresent("r")) {
switch (MyVar.r) {
  case 3.14 :
  {
    nextstate idle;
  }
  default :
  {
    nextstate idle;
  }
}
}
```

A choice instance value can be specified by an instance expression having only one assignment where `choice_field = value.`

**Example 37: Choice instance value**

```
choice choice_type
{
  public Integer ifield;
  public Boolean bfield;
}
choice_type an_int = choice_type (. ifield = 1.);
```

# Syntype

A syntype is a datatype that is based on another datatype, the parent type. The two types are not distinct in terms of type compatibility and literals. The literals of a syntype are either identical with or a subset of the literals of the parent. A syntype can be regarded as an alias of another type; an alias that may be constrained.

**Example 38: Syntype**

```
syntype myInt = Integer constants (> -10, != 0, <10);
syntype smallPrime = Natural constants (1,2,3,5,7);
```

```
Integer [1..10] myvar; /* inline syntype definition */
```

**Note**

*Constraints attached to a syntype are treated informally, that is they are not checked by the Semantic Checker, or considered by the code generators.*

### State machine

The State machine concept is explained in detail in the Behavior Modeling section.

### Stereotype

The Stereotype concept is explained in detail in the Extensibility section.

### Relationships

The following Relationships can be used in class diagrams. These are described further in the section Relationships in UML.

- Association
- Aggregation
- Composition
- Dependency
- Extension
- Generalization
- Realization
- Manifestation.
- Containment

# Object Modeling

While class modeling focuses on finding the kinds of objects in the designed application, object modeling is concerned with describing how these objects may appear at run-time. Typical questions for this analysis activity may be:

- **Which objects exist in the application at different points in time?**
- **What does the objects look like in terms of attribute values etc.?**

- **How are the objects linked to each other? Which objects have knowledge of which other objects?**

Objects are also knows as instances, and instance modeling is thus also used as a term for describing this analysis activity.

It is common to perform object modeling in parallel with class modeling. As objects of the application are identified they can be defined in the model. This can be done even before the type of the object is known.

In most real-world applications the number of objects at run-time is very large. It is therefore common to only describe those objects that are of special interest for the design. For example, it may be particularly interesting to identify objects that get created at application start-up time to get an understanding of the initialization phase of the application.

Object modeling uses mainly Object Diagrams for defining objects and their relationships, although Class diagrams are sometimes also used.

# Object Diagram

An object diagram gives a static view of objects that exist in an application at a specific point in time (a "snapshot" view). The objects shown in an object diagram can be named, and it is possible to specify the type of objects. The objects' attribute values, called **slots**, can also be specified. Links between objects can be visualized using link lines.

A named object in Tau is called a **named instance** to distinguish it from unnamed instances, such as applied stereotype instances. A Named Instance is a definition which by default is placed in the scope containing the object diagram. It is, however, also possible to show named instances from other scopes by dragging them from the Model View onto an object diagram.

An object diagram may contain multiple instance symbols showing the same named instance.

### Example of object diagram

The object diagram below shows a snapshot view of objects available in the application described by the class diagram shown in Figure 79 on page 263.

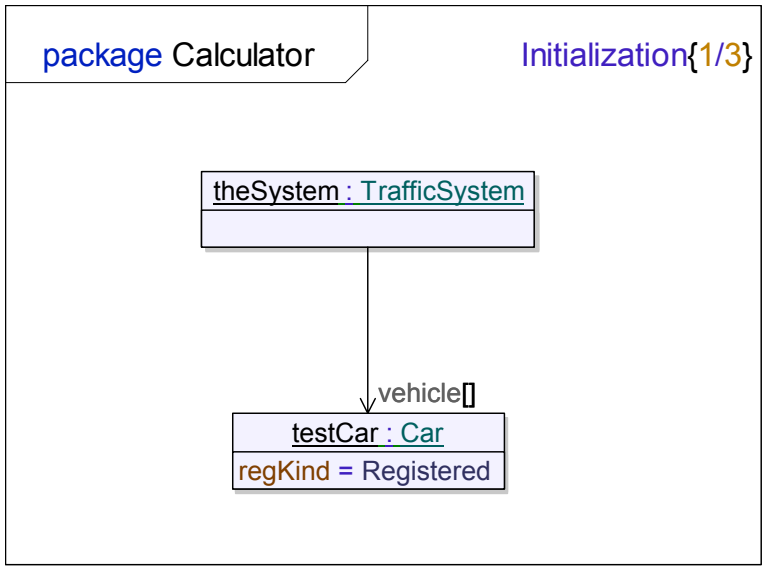



*Figure 94: Object diagram*

This object diagram tells us that at some point in time (supposedly at initialization time judging from the name of the diagram) this application contains one instance of the `TrafficSystem` class. It has one instance of `Car`, called `testCar`, in its `vehicle` list attribute. The `testCar` object has the value `Registered` for its `regKind` attribute.

### Model elements in object diagrams

The following model elements can be visualized in object diagrams:

- Named Instance
- Slot
- Dependency

## See also

"Class diagram" on page 263.

## Named Instance

A named instance represents an object (instance) in a modeled system and describes this object completely or partially. Since objects may change over time, a named instance only provides information about the object at a specific point in time, or for a specific time period. Note that UML object diagrams do not provide means for formally specifying

- the point in time, or time period, where the object complies with the named instance specification
- whether or not the named instance is a complete or partial specification of the object

A named instance may have a name. Often this name is to be interpreted informally, and does not correspond to any property at the run-time object described by the named instance. However, the usual rules for definitions apply to named instances. For example, the names of named instances in the same scope must be unique (see Scope, model elements, and diagrams).

A named instance may have a type. If the specified type is a class, the named instance describes an object of that class. If it is a datatype, the named instance describes a value of that datatype. It is also possible to specify a behavioral feature, such as an operation or a signal, as the type. In that case the named instance describes an event in the system. For example, if the type is an operation the named instance describes an operation call, and if the type is a signal it describes an event of that signal.

The type of a named instance can also be an association. In that case the named instance represents a Link.

It is allowed to specify an abstract type for a named instance. This does not mean that the described object is of abstract type, but merely that all shown properties for the object belong to the abstract type only. The described run-time object would have a type that is a concrete subtype of the abstract type.

If the named instance type contains structural features, such as class attributes or signal parameters, the named instance may specify values for those structural features. Such a value specification is called a Slot.

A named instance is shown in an object diagram using an **InstanceSymbol**.

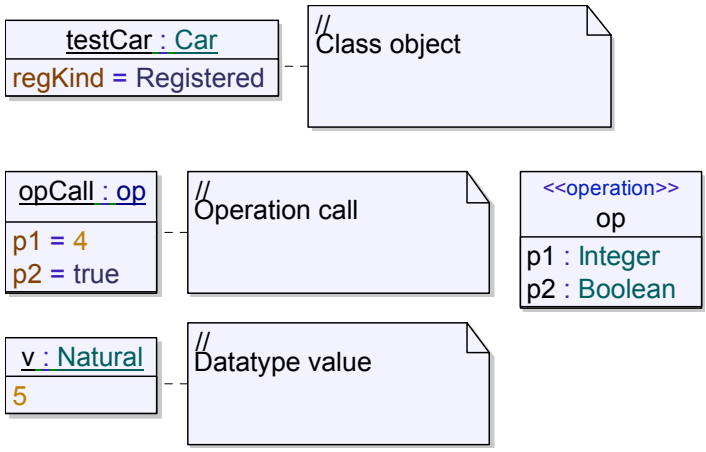



*Figure 95: Instance symbols defining named instances*

As can be seen an instance symbol contains two basic compartments. The upper compartment holds the name and type of the named instance, while the lower compartment contains the slots. Note that the syntax for defining a slot is the usual assignment syntax (the structural feature is assigned a value). For data type values a plain value can also be used.

**Note**

*Currently the semantic checker will not check type compatibility between a datatype value and the datatype. Hence, datatype values in object diagrams are for informal modeling only.*

### Link

A link is a named instance whose type is an association. It describes a run-time relationship between two objects. In programming language terms, a link could correspond to a pointer or a reference.

Links can be visualized in object diagrams in two ways:

1. As a link line, connecting two instance symbols.
2. As an ordinary slot in an instance symbol, where the right hand side of the slot refers to the target named instance.


*Figure 96: Two ways to specify a link*

The text that is entered on the target end of a link line is an expression (see Expressions). It becomes the left hand side of a Slot expression.

The name of a link can be specified by typing it in a label in the center of the link line.

## Slot

A slot is a value specification for a structural feature belonging to the type of a named instance.

Slots are used for showing those values of an object that are of interest. The fact that a named instance has no slots defined does not mean that the corresponding object has no structural feature values, but merely that those values are not of interest in the model.

Slots may reference all kinds of structural features of a type, including inherited features, and features with non-public visibility.

A slot is an assignment of a value (the right hand side) to a structural feature (the left hand side). The right hand side is often just a plain identifier, but more advanced expressions can also be used. Refer to the following model:

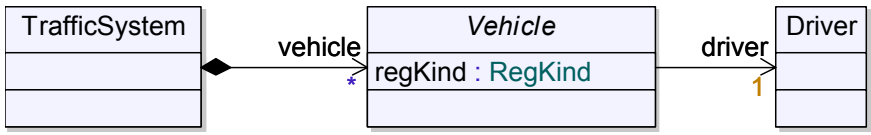



*Figure 97: Three classes with relationships*

Slots defined for an instance of TrafficSystem can for example have the left hand sides listed in the table below:

| Slot left hand side | Meaning |
|---|---|
| `vehicle[]` | One instance in the `vehicle` collection. The index of the instance in the collection is not specified. |
| `vehicle[4]` | One instance in the `vehicle` collection, located at index 4. |
| `vehicle[].driver` | The `driver` instance of an instance in the `vehicle` collection. |

Note that if the latter example is visualized with a link line we can show these kinds of indirect links between objects in a compact way:

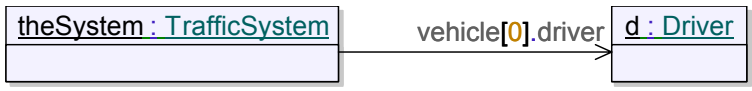



*Figure 98: Visualizing the link from the traffic system to the driver of its first vehicle*

### Self reference

There are two equivalent ways to specify self references for objects. The right hand side of such a slot can either be a reference to the containing named instance, or the keyword `this` can be used.

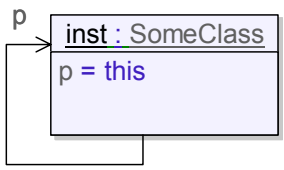



*Figure 99: Self reference shown in slot label and with link line*

# Architecture Modeling

During architecture modeling, the internal structure of active classes is described from a communication point of view. This is done by connecting the attributes of the class (in this context referred to as parts) with connectors, and to specify which signals that may be sent along these connectors. This structure of parts and connectors is called the architecture, or composite structure, of the class.

Architecture modeling typically takes place after, or in parallel with, class modeling during the design phase.

## Composite structure diagram

A composite structure (former architecture) diagram defines the internal run-time structure of an active class, in terms of other active classes. These building blocks are referred to as parts when they are composite parts of the containing class. Furthermore the parts are also restricted to be instantiations

of active classes. Composite structure diagrams may also express the communication within the active class by visualizing connectors between the communication ports of the parts.

### Example

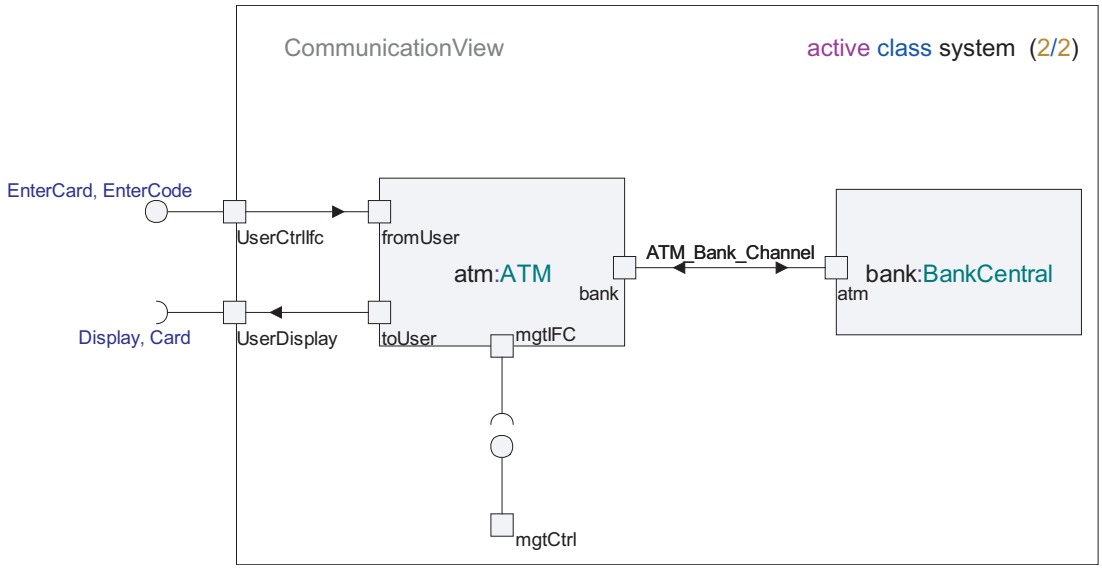



*Figure 100: Composite structure diagram with parts, ports and connectors*

## Part

A part represents one or more instances that is owned by a containing class instance.

As for all attributes a part can have a Multiplicity that constrains the number of run-time instances that may exist. If the part has a multiplicity > 1, then a container type is assumed for the parts. The specific container type can differ depending on the loaded profiles and Add-Ins, but by default the String type is used.

When an instance of the containing class is created, a set of instances corresponding to these parts may be created either immediately or at some later time as described by the initial cardinality and the multiplicity for the part.

### Symbol

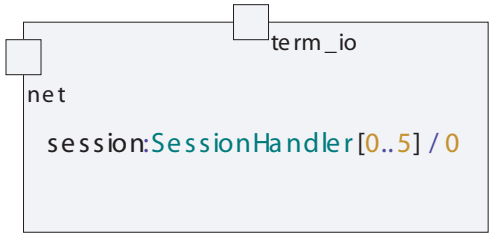



*Figure 101: Part*

- If the part symbol has only a name, the implicit class is constructed automatically when the part symbol is created.
- More than one part symbol with the same name can be present in the same composite structure diagram.

If the referenced class is omitted, this corresponds to a part definition with an inline class definition. Specifying a part in this way means that the class definition is not separated from the usage of the class which makes the description more compact, but on the other hand less suitable for reuse.

A part of an active class may be shown in the attribute compartment of the active class symbol, since a part is an Attribute of the containing class. When an attribute is of the kind part, it describes a composition relationship between the container class and the part class.

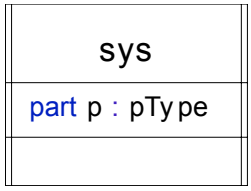



*Figure 102: A part visualized in the attribute compartment of a class symbol.*

It is also possible to give an overview of a hierarchy of parts using composition relations in a class diagram as in Figure 103 on page 302

The initial cardinality determines the number of initial instances that will be created automatically when the containing entity is created. If no initial cardinality is given, the number of initially created instances will be equal to the lower bound of the Multiplicity of the part. If no multiplicity is given, one

instance will be created automatically and there will be no upper bound for the number of simultaneous instances. These instances are instances of the classifier typing the part.

Parts may be joined by connectors attached to ports. Parts are used to describe both static and dynamically created and terminated active instances.

A part specifies that a set of instances may exist; this set of instances is a subset of the total set of instances specified by the classifier typing the part. When an instance of the containing class is terminated, the contained instances will also terminate.

A part symbol refers to an attribute in the model. The appearance of a part symbol in a composite structure diagram varies with the aggregation kind of the corresponding attribute. If the Aggregation kind is composite, the outline of the part symbol is a solid line. If the aggregation kind is reference or shared, the outline of the part symbol is dashed.

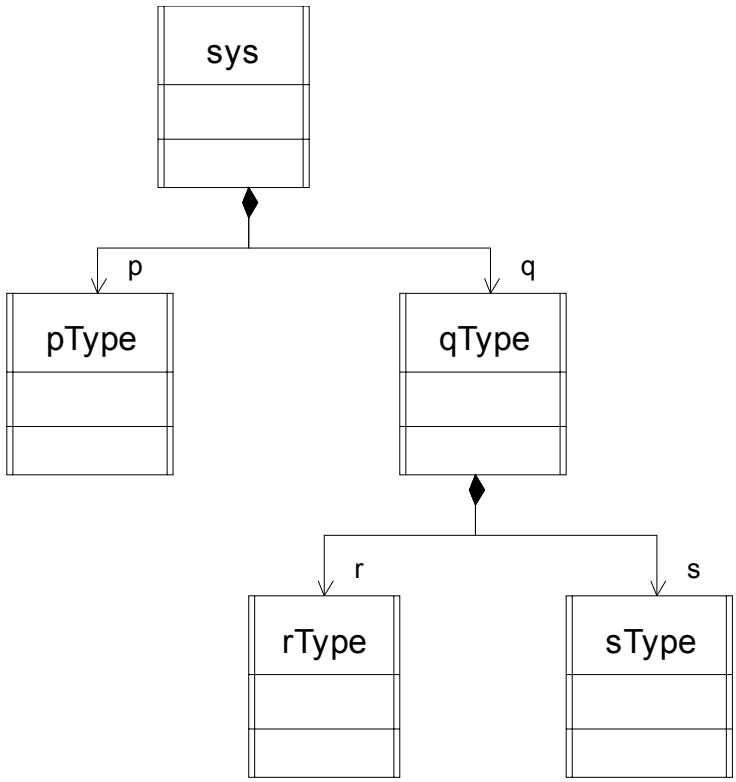



*Figure 103: A part hierarchy visualized using composition in a class diagram*

**Example 39: Simple part**

```
myP
```

**Example 40: Type-based part**

```
myP : PT
```

**Example 41: Part with initial and maximum number of instances specified**

```
myP : PT [0..10] / 1
```

# Connector

A Connector specifies a medium that enables communication between parts of an active class or between the environment of an active class and one of its parts. Connectors can visualize communication paths in an intuitive fashion.

A connector may be unidirectional or bi-directional and specifies for each direction the allowed information. Information that can be sent or conveyed on a connector can be described by: signal, attribute, signal list and interface. When the number of signals is large, it is more convenient to define an interface or a signal list to use for each direction of the connector.

By default a connector has no name, it is non-delayed and it is bi-directional. It is possible to control the properties for a connector line from the shortcut menu on the connector line.

## Symbol

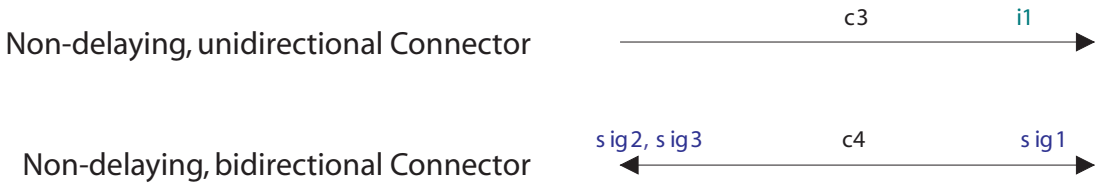



*Figure 104: Connector types*

A connector line specifies the communication path between two end points, for example ports attached to part symbols, to behavior symbols or to the frame of the diagram.

- If necessary connectors are omitted, some code generators may be able to create them implicitly.

- You can re-direct and bi-direct a connector from the shortcut menu.
- When you re-direct a bi-directional connector the signal list areas change places.
- The name of the connector is optional.
- The lists of interfaces, signals etc. associated with a connector are optional.

The structure of an active class can contain either explicit or implicit connector lines or both. Explicit connectors are visible while implicit connectors are invisible and cannot be referenced.

Implicit connectors are calculated from all matching realized and required interfaces on:

- Ports on parts contained in the containing class,
- Ports of the containing class,
- Behavior ports of the containing class.

**Note**

*If a port has explicit connectors no implicit connectors will be connected to the port.*

### Syntax

The line contains two (uni-directional connector) or three (bidirectional connector) editable text fields.

The center field specifies the name of the connector and the field placed at the end of the line specifies the signal list area. There is one signal list area for each arrowhead in the line. The signal list areas may be empty.

Stereotypes applied to the connector line are visible in a non-editable text field, positioned above the name field.

**Example 42: Connector signal list**

```
i1,i2,sl1
```

### Signal lists and interfaces

It is possible to draw a connector with signal lists to a port. In this situation the following applies:

- When there are no signals or interface given on any of the signal lists the information on the connected ports is used to deduce the signals and interfaces.
- When there are any signals or interfaces given on the signal lists associated with a connector then all transported signals and interfaces must be mentioned.

A shortcut menu choice for connector lines in composite structure diagrams **Show All Signals** is available. This fills the signal list text fields with signals and interfaces taken from the attached ports.

- Existing signals and interfaces will not be removed from affected signal lists.
- Only signals and interfaces not already existing are added to the signal list.
- The union of signals and interfaces found in the two attached ports is used. It is thus enough for a signal or interface to appear in one port, for the signal to show up in a signal list.
- If a signal is realized or required determines which signal list the signal will be put in.

### Part communication

Normally communication between parts is explicitly modeled with ports and connector lines between ports.

It is not necessary to explicitly model communication if it is unambiguous, that is if the classes for the parts in the diagram have defined ports that can be connected in only one possible way.

It is allowed to connect a connector directly to a part symbol. The behavior is that an unnamed port is created, attached to the part and the connector is connected to this part. This is allowed both when creating a connector and when reconnecting an existing connector. This port is not deleted if the connector line is deleted, the port must be manually deleted if it is not needed in the model anymore.

## Behavior port

Even if an active class has structure, that is has parts, it may also have its own behavior, expressed as a state machine. This behavior can be referenced in the Composite structure diagram using a behavior port.

The main purpose of the behavior port is when defining Connectors between a Part of the Active Class and the Behavior of the Active class; in this case it must have a Behavior Port.

It is possible to attach connectors to a behavior port in the same way as for Ports on Parts in order to define the communication interface of the state machine. It is allowed to have several behavior port in one diagram; in this case, they all refer to the same underlying behavior.

### Symbol

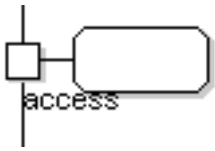



*Figure 105: A behavior port*

The Behavior port symbol specifies a reference to the unique state machine of the defined class.

- There can be several behavior port symbols present in one diagram.
- There is no text field in the symbol.

Behavior ports looks like ordinary ports in class diagrams. The appended behavior information is only shown in architecture and state machine diagrams.

**Hint**

*You can add the behavior symbol to a composite structure diagram in two ways. Either by adding a port symbol to the diagram or by dragging an existing port from the Model View browser to the composite structure diagram. In both cases you also have to choose the command behavior port from the shortcut menu, or set this property using the Properties Editor.*

## Relationships

### Dependency

The Dependency relationship in composite structure diagrams is used between parts, to show that one part is dependent of another. One common use is to indicate a create dependency between parts, that is that instances of one part can create new instances of another part.

## Update model

When the **Active Modeler** add-in is activated the shortcut menu will contain a new choice named **Update model**. This command is used on unbound entities, to invoke them in the current model.

For composite structure diagrams the following is supported:

### Composite Structure diagram

Updates the entire diagram by updating every element of the diagram.

### Connector Line

If the connector is bound and there are unbound signals in its labels, these signals are created and added to the required/realized lists of the connected ports.

If the connector is not bound (grey line) the connector is created and the labels filled with information from the connected ports. The enabled direction is also derived from the ports.

### Part Symbol

Creates an active class and updates the type of the part symbol. Adds a port to the class and the part symbol. The active class contains a simple state machine to make it possible to build directly. The name of the class is derived from the name of the part if any.

# Component Modeling

Component modeling is about identifying key Component of a system and model their Interfaces and Relationships.

The key focus when modeling components is to enforce strong encapsulation by hiding the implementation details inside a component and only expose a small set of well defined interfaces.

Weak coupling, i.e. minimized dependencies between different components, is another design principle often applied in component modeling.

# Component diagram

A component diagram describes the static structure of a system through a set of Components, their Relationships and their Realized interfaces and Required interfaces. Other model elements like Classes and Artifacts can also be shown in a component diagram to illustrate their relationships with the components.

### Example

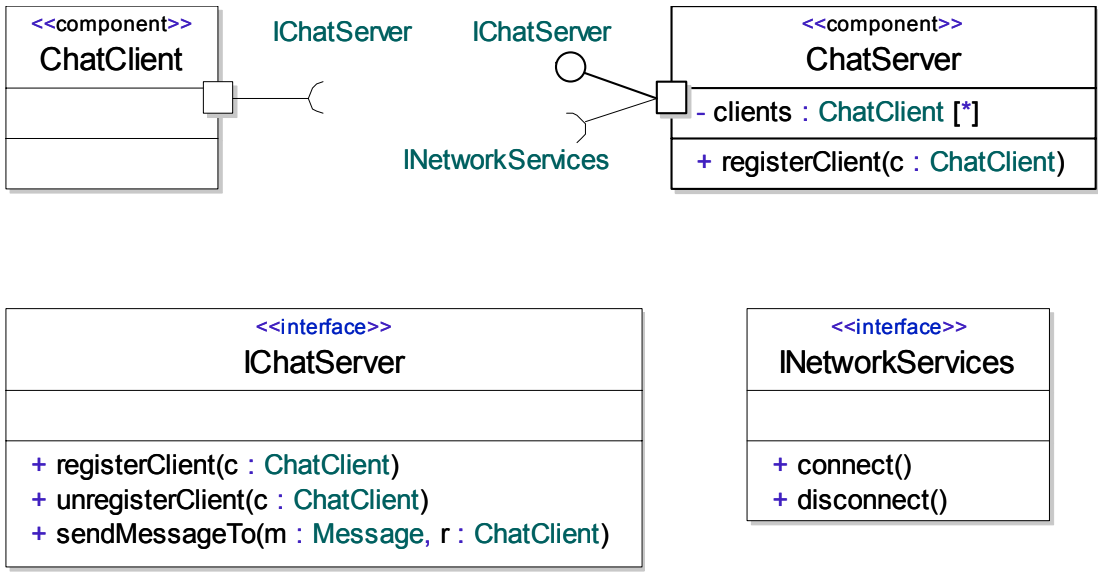



*Figure 106: Component diagram*

### Model elements in component diagrams

The following elements are found in component diagrams

- Component
- Artifact
- Class
- Interface
- Port
- Realized interface
- Required interface
- Relationships

**See also**

"Class diagram" on page 263

## Component

A component is a small part of a system that is well encapsulated and provides a well specified service.

The service provided by a component is specified through its Realized interfaces. A component should only be accessed through them. The component may also be dependent on other services; this is specified through its Required interfaces.

The implementation of the component, i.e. its behavior and architecture should not be exposed to clients. When only the Interfaces are exposed, one component can easily be substituted with another one, with a completely different implementation, without affecting the client.

The differences between a Class and a component in UML is minimal, and they can be used interchangeably. Anything that can be done with a class can also be done with a component. When using components though, the design principles outlined above should be adhered to.

### Symbol

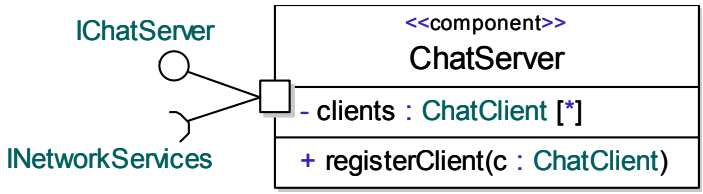



*Figure 107: Component with a port and realized/required interfaces*

The component symbol is identical to the Class Symbol, with the `keyword` «`component`» added to the top.

**See also**

"Class" on page 264.

### Relationships

The following relationships can be used in Component diagrams:

- Association
- Aggregation
- Composition
- Dependency
- Generalization
- Realization
- Manifestation
- Containment

# Activity Modeling

Activity modeling is about using Activity Diagrams to model behavior by organizing it into small behavioral units and to describe the control and data flow between these units. It can also describe how these units are distributed across a system.

Activity modeling can be used at an abstract level for business modeling or at a very low level to model behavior at action code level. It is particularly useful for the modeling of asynchronous and distributed systems.

This chapter describes the execution semantics for activity models as specified by the UML standard. In Tau this semantics is of importance when simulating activities, as described in "Activity Simulation" on page 479. Deviations from, or extensions to, the UML standard in that implementation are mentioned in notes, where applicable.

See also

"Scenario Modeling" on page 223

"Behavior Modeling" on page 330

# Activity Diagram

An activity diagram describes how a behavior is divided into small behavioral units, Action Nodes, and controls the execution sequence between them using Activity edges and control constructs such as Decision, Fork and Activity Final nodes.

Object Nodes and Pins are used to describe how objects and data are passed between the different actions.

Activity Partitions are used to group related actions into groups, for example by function or by owner.

Activity diagrams are similar to flowcharts.

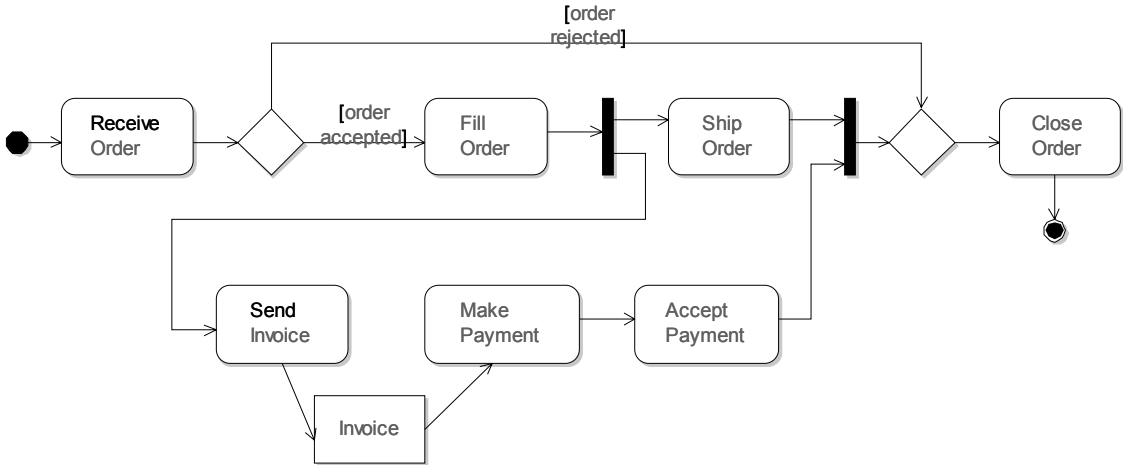



*Figure 108: Activity diagram*

## Create an activity diagram

Activity diagrams can be included in packages, classes, use cases, operations and activities.

1. Select the entity where to create the activity diagram in the Model View.
2. From the shortcut menu select **New** and then **Activity diagram**.

### Flow orientation

Although the flow in an activity diagram can go in any direction there is a setting to help you create a structured flow aligned either horizontally or vertically. This is controlled via the Tools menu, Options dialog. Select the UML Advanced Editing tab, look in the Activity diagrams section: Orientation: Horizontal or Vertical. Horizontal is default.

When horizontal orientation is chosen for activity diagrams, it is easy to create horizontal activity flows:

- Line handles are placed on the middle of the right symbol border.
- New fork/join symbols have a vertical orientation as default. (Already existing fork/join symbols are not changed when the default orientation is changed.)
- New partition symbols have as default a header size where the height is larger than the width. (Already existing partition symbols are not changed when the default orientation is changed.)

When vertical orientation is chosen:

- Line handles are placed on the middle of the bottom symbol border.
- New fork/join symbols have a horizontal orientation as default.
- New partition symbols have a default header size where the width is larger than the height.

It is possible to change the default flow orientation while appending symbols in a flow by pressing SHIFT + CTRL together.

### Activity symbols from model elements

Copying information from the Model View to an activity diagram using drag-and-drop is possible. For example it is possible to drag-and-drop an operation node to create an activity symbol which references this operation. The same can be done with interaction nodes, state machine nodes and use case nodes.

**Note**

*Actions must be selected (via the shortcut menu) for an existing activity symbol in order for the reference to be visible before dragging an activity node to the activity symbol.*

### Model elements in activity diagrams

The following elements are found in activity diagrams:

- Initial Node
- Action Node
- Object Node
- Decision
- Merge
- Fork
- Join
- Connector
- Accept Event
- Send Signal
- Accept Time Event
- Activity Final
- Flow Final
- Activity Partition
- Pin
- Relationships.

## Activity

An activity is a Signature representing the behavior of a use case, operation or any other entity that can have a behavior. An activity focuses on breaking down the behavior into small behavioral units, Action Nodes, and control the execution of these units based on a token flow model. The implementation of an activity is typically described by an Activity Diagram.

### Symbol

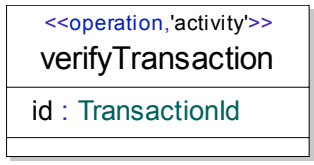



*Figure 109: Activity*

### Syntax

An activity symbol is based on the Operation symbol. It has an editable field for the name of the activity, and a compartment for Parameters of the activity.

Stereotypes applied to the activity are visible in a non-editable text field, positioned above the name field.

## Activity implementation

An activity implementation is the Implementation of an Activity signature. It contains the Activity Diagrams and a set of activity nodes connected by Activity edges. An activity implementation is normally created implicitly when creating an Activity Diagram.

### Token flows

The execution semantics of an activity implementation is based on a token flow model.Tokens flow from one activity node to other activity nodes through connected Activity edges. There are two kinds of token:

- Control token
- Data token (also known as object token)

An activity edge can transport both kinds of tokens. When a control token is transported across the edge it represents a control flow, and when a data token is transported across the edge it represents a data flow. A control flow is an activity edge with any activity nodes linked to its ends, except object nodes. A data flow is an activity edge with an object node linked to at least one of the edge's ends.

Control tokens constitute a state of logic control of a modeled system, whereas data tokens are needed to represent a state of data units which are flowing through a modeled system.

An activity edge is a directed edge which is linked to action nodes, control nodes, object nodes, pins or connectors. The direction of an edge represents the direction of the flow. The semantics of an activity edge depends on its target and source nodes.

When an activity is invoked (called) its activity implementation starts its execution by placing a control token on each Initial Node it contains. These tokens then flow downstream across outgoing activity edges and collect on the incoming activity edge ends of those activity nodes to which these edges are

connected. An activity node is allowed to start executing as soon as its **input condition** is fulfilled. Different kinds of activity nodes have different input conditions, but a typical condition is that there must be a token available on each incoming activity edge end before execution can start. When the activity node has completed its execution it delivers a token (of some kind) on all outgoing activity edge ends. These tokens eventually reach other activity nodes, and the procedure is repeated.

The activity implementation continues to execute as long as there are tokens flowing in it. If none of the activity nodes in the activity implementation has its input condition fulfilled, no tokens will flow, but the activity implementation is still in an executing mode, that is control will not be returned to the caller of the activity. Only when a special activity node, the Activity Final node, is executed will the entire activity implementation finish its execution and control is returned to the caller of the activity.

## Initial Node

An initial node specifies a starting point for the control flow in an activity implementation. When an activity is invoked and its implementation begins executing each initial node of its implementation receives a control token.

Note that an activity implementation can have any number of initial nodes, meaning that multiple control flows can be started. Note also the it is not required to have any initial nodes at all. Flows can also start from a Pin, an Accept Event and an Accept Time Event.

An initial node may not have any incoming activity edges, and therefore has no input condition. It executes as soon as it receives a control token and then offers this token to outgoing edges.

### Symbol

*Figure 110: Initial node*

# Action Node

An action node is a piece of executable functionality in an activity. The behavior of an action node can be specified in many ways, for example using an Activity, Operation, or a State machine. But it is also allowed not to associate a behavior to an action node. This can be useful at early stages of development, when the details of the behavior is not known.

The behavior of an action node, if any, can either be defined inline in the action node, or it can be referenced from the action node. Inline defined behaviors are appropriate in order to specify composite hierarchical activity implementations. Compare with Composite state. Referenced behaviors are appropriate in order to reuse the same behavior for multiple action nodes in a model. When using a referenced behavior it should normally be an activity, but in general it is possible to refer to any operation. The referenced behavior may in turn have an implementation. For example, a referenced activity may have an activity implementation.

The input condition for an action node is fulfilled when a token is available on all incoming activity edges. It then consumes these tokens and starts its execution. When it has finished its execution, control tokens are offered on all outgoing edges.

### Avoid execution deadlocks

As long as the input condition for an action node is not fulfilled it cannot execute. To avoid deadlocks in the execution it is therefore very important to understand the semantics of Token flows in an activity implementation. As an example of a common misunderstanding, consider the activity implementation in Figure 111 on page 316 below.

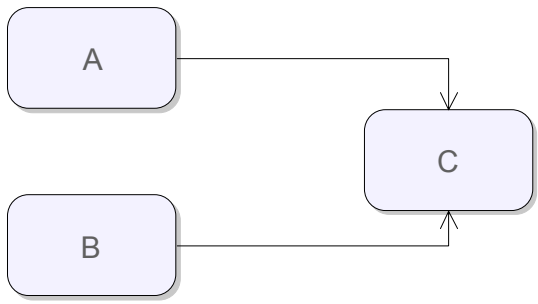



*Figure 111: Control flow between action nodes*

In this example we have three action nodes A, B, C and two control flow edges from A to C and from B to C respectively. Here C can be executed only when *both of these* edges have a token. If only the edge from A to C has a token, then the C node will wait for a token on the edge from B to C node. It is important to understand that nodes are collected on edges not on nodes.

If we instead would want the C node to execute when *at least one* of these edges have a token we should insert a Merge node between them as shown below.

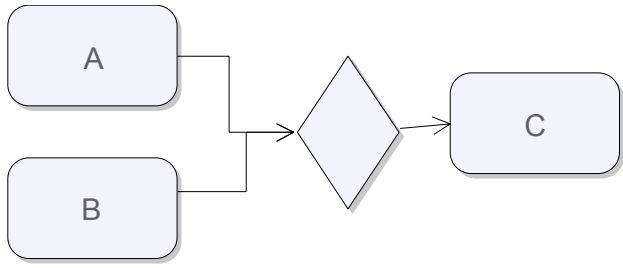



*Figure 112: Using a merge node*

**Pins**

If an action node has a behavior it can have Pins representing Parameters to its behavior. It is significant if a token reaches an action node directly via an incoming activity edge, or indirectly via an attached pin. In the former case the action node will be executed when its input condition is fulfilled. In the latter case, however, the action node itself does not execute. Instead the token flows into the behavior implementation in a “streaming” way, so that execution of the behavior implementation starts with a data token on a pin, rather than with a control token on an initial node. It is possible to combine these two mechanisms, by letting both a control token flow into the action node and data tokens flow into its pins. That is a common way of designing when the behavior needs data for its execution. It then obtains input data on the input pins, and a control token to control when execution shall begin. Before finishing the execution by executing an activity final node, output data is typically offered as data tokens placed on the output pins.

For more information about pins see Pin.

### Symbol

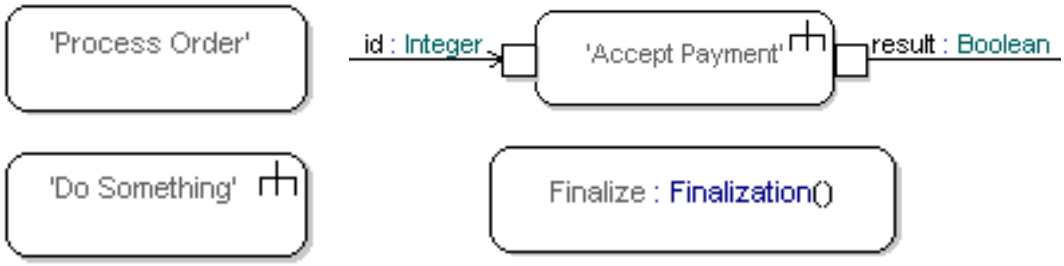



*Figure 113: Action nodes with and without pins, and with and without a behavior (inline to the left and referenced to the right)*

A shortcut menu choice **Actions** is available. When checked the a text field is added for action code. The default is to not display the Actions text field.

A shortcut menu choice **Partition Reference** is available. This command will display the text field for Partition Reference above the symbol name field. The default is to not display the Partition Reference text field.

A shortcut menu choice **Show All Parameters** is available. The Show All Parameters command will make all Pin/Parameter symbols visible for the current selection.

### Syntax

The action node symbol may have an informal name. If it references a behavior the signature of the behavior appears after a colon. If it has an inline behavior a “rake” symbol is shown in the upper right corner of the symbol.

Activity partitions in which an action node is explicitly contained, may be specified in a separate text field above the name field. The syntax is a comma-separated list of references to activity partitions enclosed in parenthesis.

Stereotypes applied to the action node are visible in a non-editable text field, positioned above the activity partition reference field.

## Object Node

An object node represents an instance of a classifier, for example a Class, participating in the flow. The instance and its values is available for use by the activity.

The input condition for an object node is that there must be a token on each incoming activity edge before it may execute. Execution of an object node simply means that a data token is placed on each outgoing edge. The type of the data token is the type of the object node, that is the classifier.

An object node does not specify how the output data is obtained. To do that an Action Node node with an output pin can be used instead. The behavior of the action node then specifies how to compute the data.

### Symbol

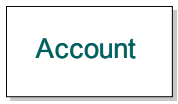



*Figure 114: Object node*

### Syntax

The object node symbol has one text label containing the name of the classifier it represents. It is also possible to give an informal name for the object node. The syntax is then `<name> : <type>`.

Stereotypes applied to the object node are visible in a non-editable text field, positioned above the name field.

## Decision

A decision node is a control node used in a flow to select one out of several outgoing flows based on guard conditions. A decision node has one incoming edge and multiple outgoing edges, each with a guard.

When a token arrives at the incoming edge of a decision node, the guards of the outgoing edges are evaluated. The order in which the guards are evaluated is not defined by UML, except that any ‘else’ guard is evaluated last. It is therefore recommended to specify guard conditions that are mutually exclusive. At most one of the guards may be an “else” guard. This guard condition is fulfilled if no other guard condition is fulfilled.

The input token will be placed on the first edge that is encountered for which its guard condition is fulfilled. If no such edge is found the token is consumed by the decision node. This is typically an exceptional situation which is best avoided by using an ‘else’ guard on one of the edges

**Note**

*The current implementation of the activity execution semantics in the Activity Simulator only supports informal decisions and decision answers. When such a decision node is executed the Model Verifier will prompt interactively for which outgoing edge to select. This is a useful feature at early stages of development, since it allows activities to be simulated before the exact guard conditions are known.*

### Symbol

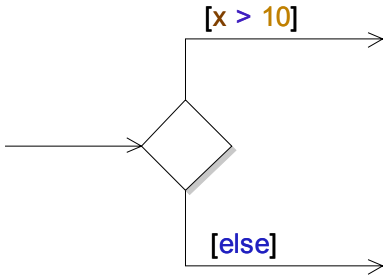



*Figure 115: Decision node*

When formally defined the guard conditions shall evaluate to boolean expressions. Any visible variables, e.g. local variables of an activity implementation, may be used in the guard condition. The keyword `else` is used in a guard to indicate that the edge is selected if none of the other guards evaluates to `true`.

To merge back multiple outgoing decision flows into a single flow, use a Merge node.

**Note**

*The Decision and Merge nodes share the same symbol in the symbol palette of the activity diagram editor.*

## Merge

A merge node is a control node used to bring together multiple flows into one. Whenever a token arrives at one of the incoming edges, it is relayed onto the outgoing edge. Unlike Join, it is not a synchronization of the incoming flows.

### Symbol

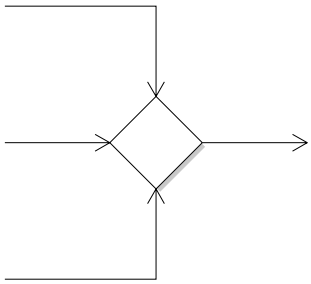


*Figure 116: Merge node*

**Note**

*The Merge and Decision nodes share the same symbol in the symbol palette of the activity diagram editor.*

## Fork

A fork node is a control node that splits one flow into multiple concurrent flows. Whenever a token arrives at the input edge it will be copied, and one copy will be placed on each outgoing edge. A fork node is thus a means for introducing parallelism in an activity model.

To join multiple concurrent flows back into one single flow, use the Join node.

### Symbol

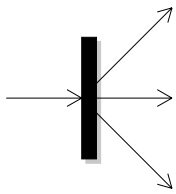


*Figure 117: Fork node*

**Note**

*The Fork and Join nodes share the same symbol in the symbol palette of the activity diagram editor.*

## Join

A join node is a control node used to join, or synchronize, multiple concurrent flows back into one single flow.

The input condition for a join node is that there must be a token available on all incoming edges. When that condition is fulfilled tokens are placed on the outgoing edge according to the following rules:

- If all input tokens are control tokens, then one single control token is placed on the outgoing edge.
- If some of the input tokens are data tokens, then all these tokens, but only these, are placed on the outgoing edge.

**Note**

*The current implementation of the activity execution semantics in the Activity Simulator does not follow this rule. Instead it is the token that arrives last to the join that will decide which kind of token that is placed on the outgoing edge.*

To fork a single flow into multiple concurrent flows, use the Fork node.

### Symbol

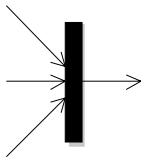


*Figure 118: Join*

**Note**

*The Join and Fork nodes share the same symbol in the symbol palette of the activity diagram editor.*

# Connector

Connector nodes are used as a graphical short hand to simplify drawing of complex flows. An Activity edge can end in a connector node and be continued at another connector node with the same name. This can be used to split an activity implementation specification over multiple activity diagrams.

A connector node may have many incoming edges, but at most one outgoing edge. Semantically a connector node is equivalent with a Merge node. A token that arrives at an incoming edge of a connector node is relayed onto its outgoing edge. If a connector node does not have an outgoing edge it is semantically equivalent with a Flow Final node.

### Symbol


*Figure 119: Connector node*

### Syntax

The connector node symbol has one text label containing the name of the connector node.

# Accept Event

An accept event node is used to indicate waiting for a specific event, typically a Signal. When the specific event is received, the flow continues by placing control tokens on all outgoing edges.

Semantically an accept event node is equivalent with an Action Node node, with a behavior that waits for the event to be received.

Data passed on with the event can be used later in the flow by using output Pins from the accept event node. An accept event node may not have any input Pins.

The accept event action is similar to an Signal Receipt (Input) in a State machine.

**Symbol**

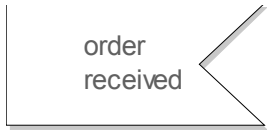



*Figure 120: Accept event node*

## Send Signal

The send signal node is used to create an instance of a Signal and send it. It is similar to the Signal sending action (output) in a State machine.

Semantically a send signal node is equivalent with an Action Node node, with a behavior that sends the signal.

A send signal node may have input Pins providing actual arguments for the formal parameters of the signal to send. It may not have any output Pins.

**Symbol**


*Figure 121: Send signal symbol*

## Accept Time Event

An accept time event is a special version of the Accept Event node. It is used to indicate waiting for a specific time event, typically a Timer timeout or an absolute time value. When the specific time event is received, the flow continues by placing control tokens on all outgoing edges.

For more information about timers, see Timer handling and time.

Contrary to an Accept Event node, an accept time event node may not have any Pins. In order to wait for a timer with parameters, use an Accept Event node instead.

**Symbol**


*Figure 122: Accept time event*

## Activity Final

The activity final node indicates the termination of an activity. When a token reaches an activity final node, all flows of the activity are terminated, and the execution of the activity is completed. Control is returned to the caller of the activity.

An activity final node may have an arbitrary number of input edges, but no output edges.

To terminate a single flow of an activity, use Flow Final nodes.

**Symbol**

*Figure 123: Activity final*

## Flow Final

The flow final indicates the termination of a single flow in an activity. Only that particular flow is terminated, not the entire activity. There might still be other ongoing flows (compare Fork) in the activity.

Tokens received by a flow final node will be consumed by it. A flow final node may have an arbitrary number of input edges, but no output edges.

To terminate the entire activity, use Activity Final nodes.

**Symbol**

*Figure 124: Flow final*

# Activity Partition

An activity partition, sometimes called a swimlane, is a grouping mechanism used to group related Action Nodes to each other. They provide a way of splitting an activity diagram into different sections to make it easy to see which section that performs a certain activity, and how data flows between the different sections.

For example, in business modeling, the different subdivisions of a company can each be represented by a partition. Another example is to let each partition represent a thread in a real-time operating system. The diagram would then show how the actions of a system are distributed among threads.

An activity partition may have a type, which typically is a Class. This expresses a constraint that those instances that perform the actions of the activity partition, must be instances of that type. The activity partition may further constrain performed actions by specifying one particular instance, which then must perform the actions. It may also specify an Attribute, which then must contain the instances that perform the actions.

### Symbol

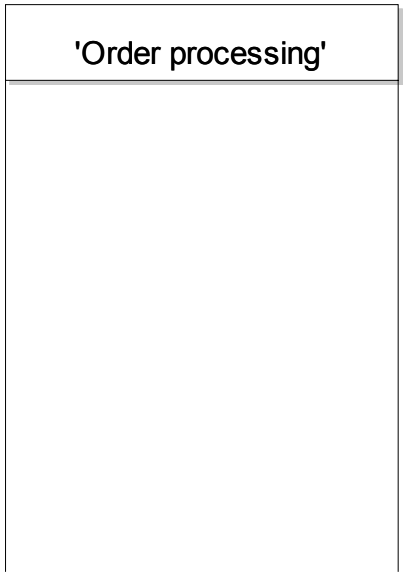



*Figure 125: Activity partition*

A constraint concerning type, instance or attribute for the activity partition is specified in a label just below the name label. The syntax is the same as is used for a Lifeline.

Action Node node symbols that are graphically contained in an activity partition symbol represent actions that belong to that activity partition. It is possible for an action node to belong to more than one activity partition. This can happen when using activity partition symbols that are rotated, so that the intersection of two activity partition symbols contains the same action node symbol. However, it is not possible to accomplish involvement in more than two activity partitions this way, because an activity diagram only has two dimensions. In order to specify that an action node belongs to more than two activity partitions an explicit list of included partitions may be specified for the action node. If an action node has an explicit list of activity partition references it overrides the implicit reference that can be deduced from the graphical position.

**Example 43: Implicit and explicit activity partition references**

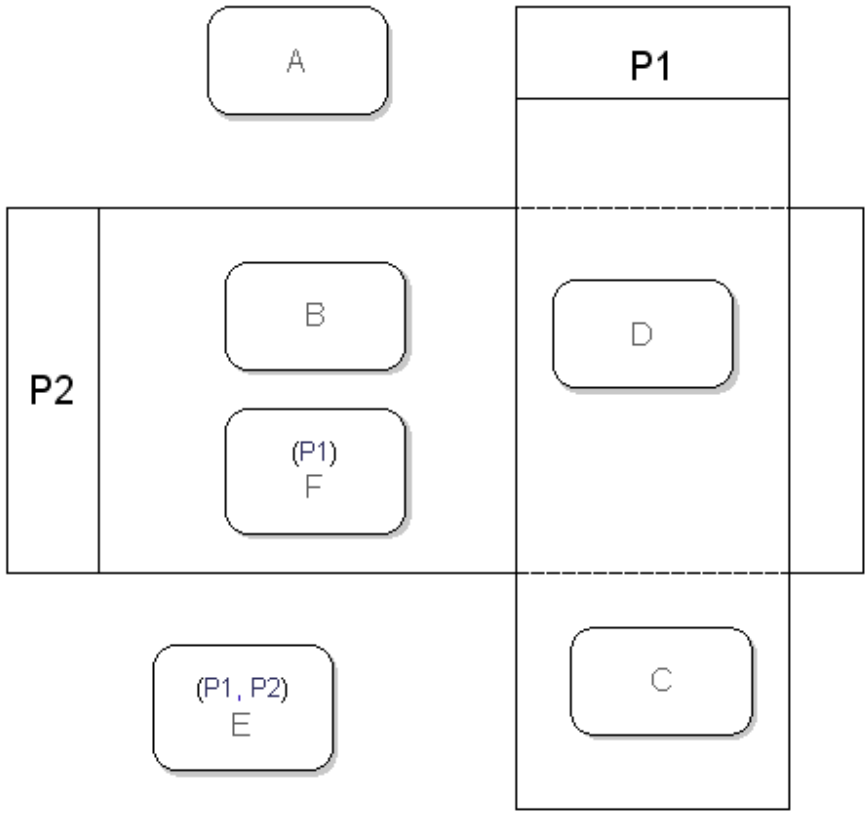



*Figure 126: Action nodes referring to activity partitions*

The action nodes above use both implicit activity partition reference (deduced from the graphical position of the action node symbol) and explicit activity partition references.

A does not belong to any partition.

B belongs to partition P2 (implicit reference)

C belongs to partition P1 (implicit reference)

D belongs to partition P1 and partition P2 (implicit reference)

E belongs to partition P1 and partition P2 (explicit reference)

F belongs to partition P1 (explicit reference)

---

**Partition symbol as Dimension Specification symbol**

When there are several rows of partition symbols, the partition symbol in the top row may be used as a dimension specification symbol. The partition symbol has a shortcut menu choice for Dimension. A partition has a plain text in the main label while a dimension has an italic font in the main label.

It is possible to use both horizontal and vertical dimensions at the same time.

# Pin

A pin represents a parameter of the behavior of an Action Node node, and are used for passing data to and from that behavior. They can be seen as Object Nodes for inputs and outputs to actions.

Pins that have incoming edges input data to the behavior, and are therefore called input pins. Pins that have outgoing edges output data from the behavior, and are consequently called output pins. The direction of the Parameters represented by the pin should match how edges are connected to the pin. For example, an input pin should only have incoming edges, and the corresponding parameter should have “in” direction.

The semantics of executing a pin is the same as for an Object Node. Hence, execution places a data token on each outgoing edge, and the type of these data tokens is the type of the parameter represented by the pin.

A pin may be streaming or non-streaming. In the streaming case the pin can execute to produce output data tokens even when the behavior of the Action Node node is executing. In fact there is no connection between the presence of tokens on streaming input pins and the condition for when the behavior of the Action Node node is invoked. In the non-streaming case, however, the behavior will not execute until tokens are available on all input pins.

**Note**

*The current implementation of the activity execution semantics in the Activity Simulator only supports streaming pins. However, a non-streaming pin can be emulated by combining a streaming pin with a Join node in the activity implementation of the Action Node node behavior. The join node then has two incoming edges; one from the pin on which the data token will arrive, and one from the Initial Node on which the control token will arrive when the behavior is executed.*

### Symbol


*Figure 127: Pin symbol*

### Syntax

The pin text has the same syntax as Parameters, i.e. `name : Type`.

## Relationships

### Activity edge

An activity edge is used to connect nodes in an activity implementation. It enables the flow of control and data tokens between the two connected nodes.

An activity edge is always directed, meaning that a token only can flow in one direction over an activity edge. The direction of an edge represents the direction of the flow. An activity edge can transport both kinds of tokens. When a control token is transported across the edge it represents a control flow, and when a data token is transported across the edge it represents a data flow.

An activity edge can have an informal name describing the flow it represents.

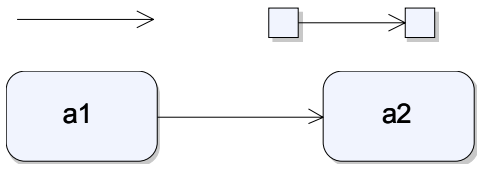



*Figure 128: Activity edge*

# Behavior Modeling

In order to obtain an executable model, the detailed behavior of operations and active classes must be specified. This is done during behavior modeling, an activity that usually takes place at the end of the design phase.

A behavior specification may contain states (that is a State machine implementation), or it may be stateless (that is an Operation body). In either case there are two ways to describe the behavior:

- As a state machine in a State machine diagram
- As a textual description in a Text diagram.

For implementations that contain states, the graphical form (State machine diagram) is often to be preferred, while for simple implementations of operations it could be enough with a textual description of the actions that constitute the Operation body.

## State machine diagram

A State machine diagram visualizes a State machine. There are two different styles of drawing state machine diagrams supported. They are described and exemplified below. It is possible to combine the two styles.

### State-oriented view

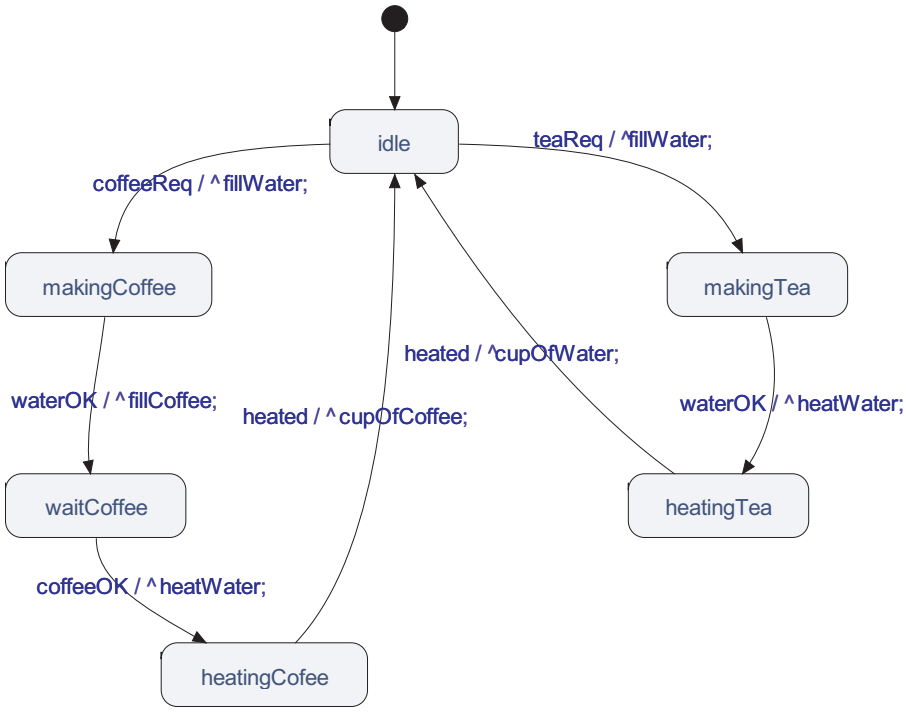



*Figure 129: State-oriented view of a state machine*

The state-oriented view of a state machine gives good overview of a complex state machine but is less practical when focusing on the control flow and communication aspects of a specific set of transitions. For this reason, it is also possible to describe the state machine in a transition-oriented way, with explicit symbols for different actions that can be performed during the transition.

### Transition-oriented view

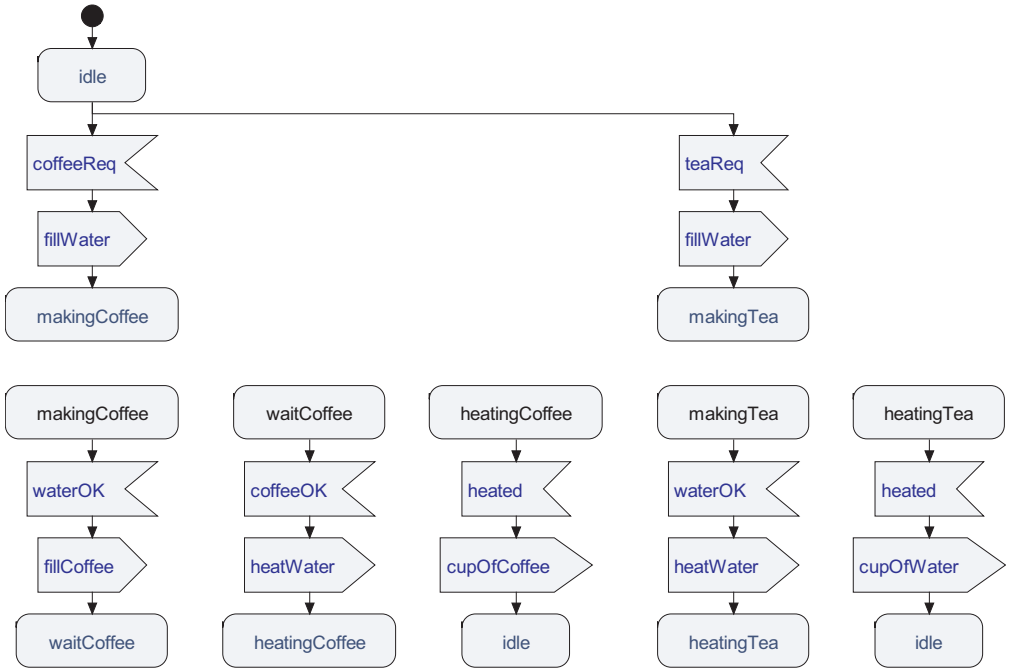



*Figure 130: Transition-oriented view of a state machine*

### Create a state machine diagram

State machine diagrams can be included in classes and operations (including use cases).

1. Select the entity where to create the statemachine diagram in the Model View.
2. From the shortcut menu select **New** and then **State machine diagram**.

## State machine

A UML state machine is a finite state machine extended with data and signal handling. The basic elements of a state machine is the state and the transition. In a model based on the state machine paradigm, execution is carried out with a certain state as the starting point and a triggering event that causes a transition to be executed. In the transition, actions can be carried out. At the end of the transition, a new state is entered. The state machine will be idle in this state until a new triggering event that may start a transition occurs. An alternative way to end a transition is to stop the entire state machine (active class).

### Hint

*State machines are most simply created either by right-click of a class in the Model View and choosing New->State machine diagram in the shortcut menu or by opening the Create Presentation dialog.*

### Symbol

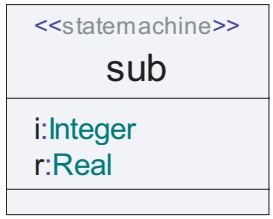



*Figure 131: State machine*

### Syntax

The symbol contains two editable text fields:

- Class Heading
- Parameters

(The Operation field is empty.)

The Parameters field contains the formal parameters of the state machine. These are used for:

- Passing values to an active class instance upon creation.
- Passing values to a composite state when entering it.

## State

A State represents a situation in a State machine where the containing object is waiting for an event that will trigger a transition to another State. This situation may have a static condition (if the state does not have substates); in this case the state machine is inactive while in the state. The situation can also be dynamic in the sense that there can be state machine behavior hidden in substates of the state.

### Symbol


*Figure 132: State*

The State symbol references one or more states and acts as source and/or target for transitions leading from or to this state (or set of states).

### Syntax

- Simple state:

  ```
  State1
  ```

- State with state list:

  ```
  St1,st2
  ```

- State with asterisk state, including list of not included states:

  ```
  *(st1,st2)
  ```

An asterisk state is a shortcut that refers to all states defined in the current state machine except the states mentioned in the list following the '*' symbol.

Since state machines are hierarchical a state may contain a sub-state machine. This is indicated in the syntax by giving the name of the state machine after a colon following the name of the state as in Figure 133 on page 334

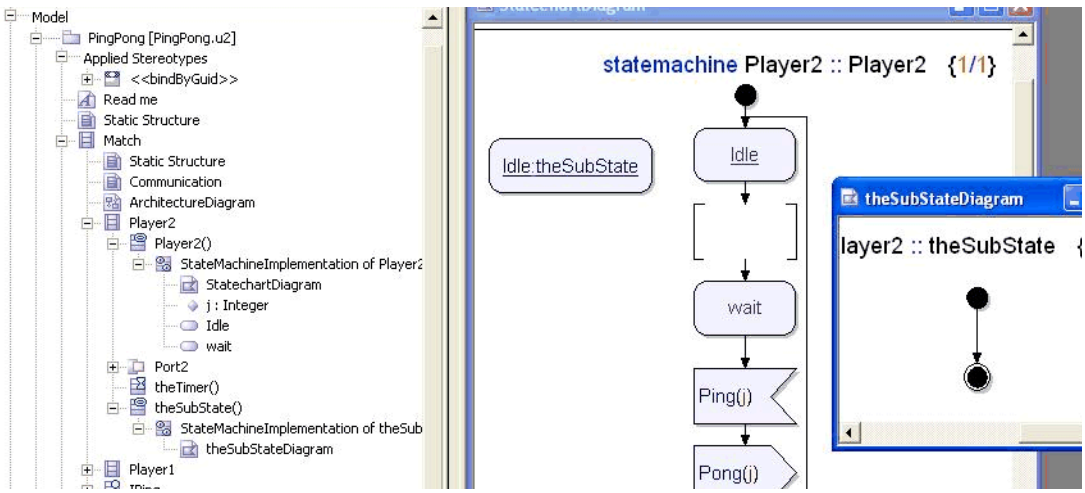



*Figure 133: Sub-state reference*

The `<state>:<state machine>` syntax may only be used for state symbols without incoming lines (i.e. the state symbol should not be a “nextstate”). It is a syntax error if a state symbol with the label `s:myStatemachine has` incoming lines.

**Note**

*The state symbol may not contain a list of states or an asterisk state definition if there are transitions that has this particular state symbol as target. State lists and asterisk states may only specify the source of transitions, not the target of transitions.*

If a state has a substate state machine and this state machine has an entry point then the entry point may be indicated in the state symbol. This may only be used if there only exist one transition that has the state symbol as its target state.

**Example 44: State with via clause**

State `St1` containing a ‘`via`’ clause that determines the entry point in a substate state machine:

```
St1 via entry1
```

If a state has a substate this is indicated in the state symbol by a “rake” in the upper right corner, symbolizing a split flow, see Figure 134 on page 335.

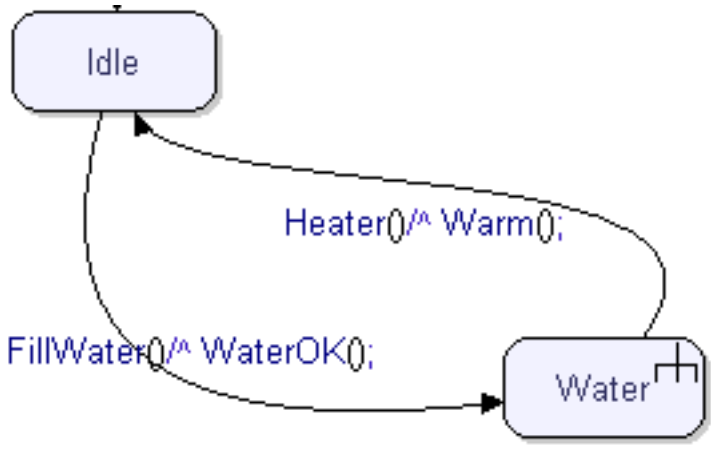



*Figure 134: State with substate*

Since the State symbol can be used for both defining a state and referencing a state (the target state of a transition), it is possible to let the symbol act as both an end point for a transition and the starting point for a new transition, thus chaining the transitions. This makes sense if using a state-oriented layout of the state machine. When using a transition-oriented layout it is however good practice to avoid this, and always separate the transitions, as in the example above, for the sake of readability.

If a state acts as source for many transitions, it is allowed to specify these in different diagrams in order to improve readability. Thus the State symbol is a partial definition of the state.

If the same transition is valid in several states, it is possible to refer to several states from the State symbol.

**See also**

"Composite state" on page 366

## Transition

A Transition is a sequence of actions that are executed when a State machine changes the active state.

The syntax used for transitions falls into two different categories depending on if a state-oriented or transition oriented syntax is used. The state-oriented transition syntax is described in "Simple transition" on page 356, the transition-oriented syntax is described by a set of trigger symbols for starting the transitions and then a set of action symbols that describe the transition details.

The different trigger symbols correspond to what event that causes the transition to be initiated. Based on this, different kinds of transitions can be distinguished:

- triggered transitions
- guarded transitions
- labelled transitions
- initial transitions

A triggered transition is characterized by the trigger that is associated with the transitions. Typically this trigger is defined by the specific signal, but it may also be defined by for example a timer or by an operation. Triggered transitions are described in more detail in section "Signal Receipt (Input)" on page 339

A guarded transition is characterized by the fact that it is *not* triggered by a specific event. Instead it is triggered either by a certain condition (Guard) that can be true or false.

Labelled transitions are not real transitions in terms of describing a state-to-state behavior. Instead they are used to decompose a transition into two (or more) parts that can be described on two different pages in a diagram. Junction is also a related construct to labelled transitions used to divide flows.

The Initial Transition (Start) is the transition that will be executed directly when the state machine is created.

A transition always ends with the state machine entering a state, with a stop, with a return or with the transfer of control to another transition.

### Guarded transition

A Guarded Transition may or may not have a trigger.

If the guarded transition has a trigger, the evaluation of the expression will be done after the triggering event has happened. If the expression evaluates to true, the transition is fired. If the expression evaluates to false, the state machine will remain in the state and the signal that caused the triggering event will be kept in the signal queue.

**See also**

"Save" on page 353

## History nextstate

The History nextstate is used at the end of a transition to return to the last visited state.

The symbol can be used to end both simple transitions and flow line (detailed) transitions.

### Shallow history

By default, the History nextstate is **shallow**. This means that when a nextstate with History is interpreted at the end of a transition, the next state will be the one in which the current transition is activated.

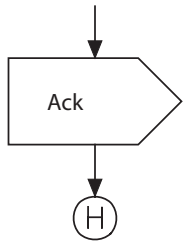



*Figure 135: Shallow History nextstate*

History nextstate can also be expressed with a normal nextstate, using a hyphen instead of a name in the symbol.

```
nextstate -;
```

### Deep history

It is possible to make the history nextstate **deep**. This means that similar to the shallow history, the next state will be the one in which the current transition is activated. This will apply recursively to all levels of substates of the entered state.

**Hint**

*You can make the history nextstate deep by selecting it and choosing the command **Deep History** from the shortcut menu.*

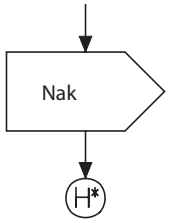



*Figure 136: Deep History Nextstate*

Deep history nextstate can also be expressed with a normal nextstate, using the following syntax:

```
nextstate ^-;
```

### Examples

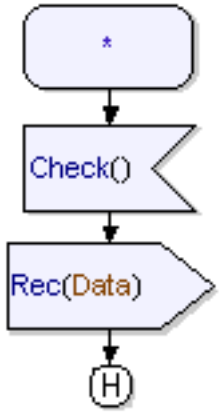



*Figure 137: Shallow History Nextstate with an asterisk state transition*

In the above example, the transition will end up in the state that was active when the transition was triggered.

## Signal Receipt (Input)

The signal receipt symbol defines which signals that should trigger a particular transition.

The transition can be guarded by a guard expression that also is shown in the symbol.

### Symbol


*Figure 138:Signal Receipt*

The signal receipt symbol receives a signal and must always be preceded by a State symbol. Together they define a transition.

**Hint**

*You can flip the symbol horizontally from the shortcut menu.*
*When you delete the signal receipt symbol, the succeeding subtree is deleted as well.*

If the same transition behavior should be invoked for several triggers in one state, it is possible to have a list of identifiers in the signal receipt symbol. This mechanism does not allow handling the parameters of each signal and all the signals will trigger the same transition that ends in one nextstate.

When receiving a signal, its parameters are normally stored in local variables. It is also allowed to ignore parameters.

The optional guard expression is defined after the trigger and is surrounded by square brackets.

### Signal queue

A State machine is associated with a **signal queue** that stores the signals that are sent to the state machine in the order they arrive.

It is not necessary to specify a transition for every possible trigger in each state. Often it is possible to predict which signals that may arrive, from your knowledge about the application or domain you are modeling. If the next signal to consume from the signal queue is not handled in the current state, that signal will be thrown away. It is also possible to Save a signal temporarily.

### Syntax

The following kinds may be referenced as triggers in a signal receipt symbol:

- Signal
- Timer
- Operation.

**Example 45: Simple signal receipt**

```
s1( i )
```

**Example 46: Signal Receipt with several triggers**

```
s1(i), myTimer, s3
```

**Example 47: Signal Receipt with virtuality**

```
redefined input s1( i )
```

**Example 48: Asterisk signal receipt**

It is allowed to specify that all triggers may invoke the transition. This is done by using an asterisk to denote all visible triggers.

```
*
```

**Example 49: Guarded signal receipt**

```
s1 [ x>10 ]
```

# Start

The Start symbol defines the starting point of a state machine or one starting point of a composite state. The start symbol thus defines the initial transition.

## Symbol

*Figure 139: Start*

## Syntax

The start symbol has one text field that can be used for:

- Referencing an entry point in a composite state

  ```
  Entry1
  ```

- Defining virtuality for the transition

  ```
  virtual
  virtual Entry2
  ```

## Action

Actions are typically done in the Action symbol using a textual syntax. The available actions are:

- Local variable definition statement
- Empty statement
- Compound statement
- Assignment
- Action
  - Signal Sending (output)
  - New
  - Set
  - Reset
- Expression statement
- If statement
- Decision statement
- Target code statement
- While statement
- For statement
- Delete statement
- Try statement
- Terminating statement
  - Return
  - Break
  - Continue
  - Stop
  - Nextstate
  - Goto (join)
  - Throw

A few of these statements also have a graphical syntax, that is a dedicated symbol. The stop, return, decision and signal sending statements have distinct symbols that allow highlighting of important operations on the transition. It is of course allowed to use the textual syntax for these statements as well. The most important actions are described below.

## Signal sending action (output)

The signal sending action in a transition allows to send signals to other State machines, the environment or within the same state machine. If the signal has parameters, expressions matching the parameter types should be provided. It is allowed to ignore parameters when sending a signal.

It is allowed to specify more than one signal at a signal sending, which will be handled as sending separate, consecutive signals.

### Symbol

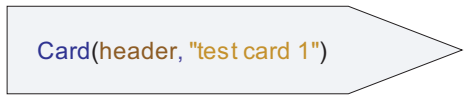



*Figure 140: Signal Sending*

The signal sending symbol sends a signal from a transition.

**Hint**

*You can flip the symbol horizontally from the shortcut menu.*

### Signal addressing

There are several ways to direct a signal to a receiver or routing the signal, including:

- Omitting addressing
- Directing the signal as a method application on the receiver
- Signal Sending via port or interface

Each of these addressing mechanisms are described below. Direct addressing of a signal is expressed using period (“<receiver>.<signal>”) for the method application on a receiver.

### Signal sending

No address or path is specified. The signal will be sent on one of the possible paths (that is a port / connector).

### Receiver is this

If the context is a state machine or an operation of an active class, **this** means the state machine of the current active instance, that is the same as self.

If the context is an operation of a passive class, **self** should be used instead, to reference to the state machine of the current instance. In this context, **this** refers to the instance of the passive class

### Signal sending via port or interface

A port identifier is given. The signal will be sent via this port

If an anonymous port that realizes exactly one interface is defined for the class the identifier can also be an Interface name. In this case it refers to the anonymous port.

### Receiver is an attribute

Either a variable or attribute is given as destination. The type of the variable or attribute must either be an interface (signal sending via) or an active class (or the special type Pid defined in the RTUtilities package).

The attribute may also refer to one of the implicit attributes **self**, **sender**, **parent** or **offspring**.

### Receiver is an expression

The expression must be typed by an interface or an active class (or the special type Pid defined in the RTUtilities package). This is a similar situation to when Receiver is an attribute. The difference is that more complex expressions can be given within the parenthesis, for example field or string extraction.

### Examples

**Example 50: Addressing mechanisms**

No address or path is specified:

```
SuspendInd
```

Receiver is an implicit attribute:

```
sender.Ack(id)
```

Receiver is an attribute, signal carries parameters:

```
Bank.Card(carddata)
```

Receiver is a Pid expression (indexed array with Pid elements):

```
(myList[10].addr).Sig1
```

An interface (referring to a port):

```
Ack(id) via myInterface
```

---

All these addressing mechanisms have the following in common:

- If there is no alive instance of a state machine at the end of the communication path, the signal will be lost.
- If the destination references a state machine instance that has terminated, the signal will be lost.
- If the receiving state machine is in a state were the signal is not handled, the signal will be lost.

## Decision

The decision construct is used to perform alternative actions in a transition dependent on the value of an expression. It is a mechanism similar to a switch. A decision has one **question** part, which contains a dynamic expression that is evaluated when the decision is executed. Furthermore, a decision has multiple **answer** parts, each containing a range expression (or just a simple expression containing a value or a constant) and leading to different partial transitions.

### Symbol

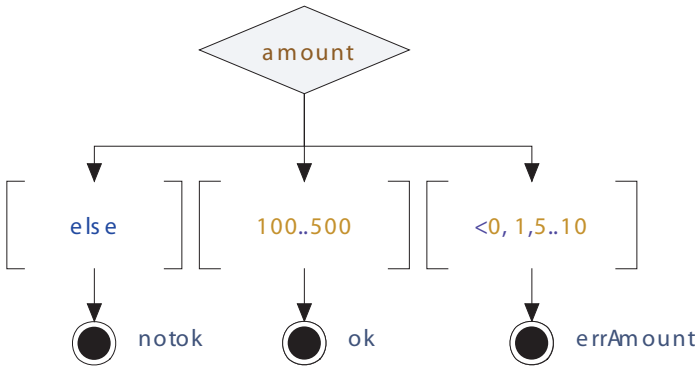



*Figure 141: Use of decision*

The Decision symbol specifies alternative paths in the behavior part of a transition.

- An expression must be defined. Each path is labeled with an answer that should match the expression for the path to be taken.
- When you delete the decision symbol, the succeeding subtree is deleted as well.

### Decision answer

The Decision Answer symbol specifies one alternative path in the behavior part of a transition and contains a range condition which is an answer to a decision question.

A range condition is given either as

- a specific value (for example "`10`" or "`true`")
- an open range (for example "`>10`")
- an closed range (for example "`2..10`")
- a comma separated list of the above mentioned alternatives.

### Informal decisions

To facilitate early verification of models it is possible to specify informal decisions. These are characterized by having an expression that is a character string and answers that also are character strings.

### Nondeterministic decisions

It is also possible to describe a nondeterministic decision. This is done by giving the expression “any” (without quotes) and leaving the decision answers empty.

### Syntax

**Example 51: Decision expression text example**

```
v+4
```

**Example 52: Decision alternative text example**

Simple example:

```
True
```

Open range:

```
>10
```

Closed range:

```
0..3
```

Several ranges:

```
<-5, 0..2, >10
```

## Guard

A guard symbol can be used for either:

- Triggering a transition based on a certain condition evaluating to “true”.
- A connect transition, that is leaving a substate via an exit point.

## Symbol

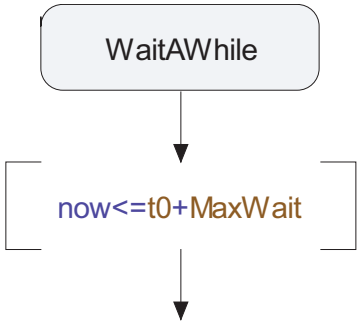



*Figure 142: Guarded transition*

If a guarded transition is based on a condition, the invocation of the transition occurs when the provided expression defining the condition is evaluated to true. The provided expression must be a simple expression and it may not cause any side effects.

If the transition is defined by referencing an exit point, then the source state of the transition must have a substate state machine. The transition is executed whenever this substate state machine exits via the specified exit point.

## Syntax

**Example 53: Guarded transition**

```
[ x>10 ]
```

**Example 54: Connect transition**

A transition triggered by leaving a composite state via a named exit point called ‘a’

```
[ a ]
```

**Example 55: Connect transition**

A transition triggered by leaving a composite state via an unnamed exit point.

```
[ ]
```

## Timer set action

The Timer Set Action creates a timer instance, which now is active. Performing set once more on an active timer instance, implicitly resets the first timer instance and creates a new timer instance.

A timer with parameters may have several timer instances active at the same time, as long as the parameter values are distinct.

### Syntax

**Example 56: Absolute time**

```
set (MyTimer, aTime);
```

**Example 57: Relative time**

```
set (MyTimer, now+10);
```

**Example 58: Timer with default duration**

```
timer MyTimer () = 5;
...
set(MyTimer);
```

**Example 59: Timer with parameter**

```
timer MyTimer (Integer id);
Integer i = 1;
...
set (MyTimer (i), now+5);
```

### See also

"Timer active expression" on page 362

## Timer reset action

The Timer Reset Action resets an active timer instance, if such an instance exists.

### Syntax

**Example 60: Reset of normal timer**

```
reset(MyTimer);
```

**Example 61: Reset of timer with parameter**

```
reset(MyTimer(i));
```

## Action (task)

The Action symbol is used for writing textual code in the behavior part of a transition, for example variable assignments, for-loops and calls of value returning procedures.

### Symbol

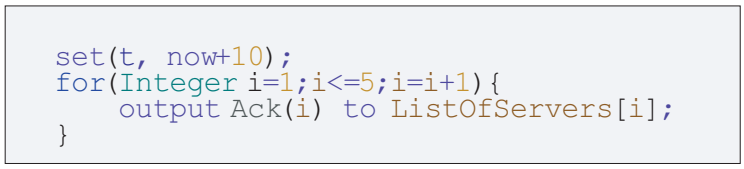



*Figure 143: Action symbol*

### Syntax

**Example 62: Simple example**

```
Integer v1;
v1 = 4;
output s(v1);
```

## Assignment

Assignments are done according to the syntax in the example below. The left hand side of the assignment can contain a variable identifier, an element of an indexed variable or a struct field of a struct or class. The right hand side contains an expression of the same type as the left hand side.

**Example 63: Various assignments**

```
Integer i = 0;
myObject = new (theType);
person.age = person.age+1;
arrival[currentDate,person] = now;
```

The assignment can also be used as an expression in itself. The returned value of an assignment expression is the right hand side expression, if the assignment is successful.

**Example 64: Assignment expression**

```
if ((a=10)==10) { output s; };
```

# Compound statement

A compound statement contains a number of statements enclosed within braces {}. It also defines a namespace which makes it possible to declare local variables within a compound statement.

## New

The **new** statement is used to create instances of both active and passive classes. To create an instance of the same class as the current class, the keyword **this** can be used. The new construct returns a reference to the created object.

It is of course always possible to communicate with the created instance using a reference to the object, so by assigning the result of **new** to a reference attribute it is possible to for example send signals directly to the created instance or to call an operation on the instance.

However, to make it possible to communicate with the instance using the ports and connectors that exists in a model, the created instance must be added to the architecture (the structure of connectors and ports) that exists in the application.

**Note**

*The way an instance is added to the architecture depends on the particular code generator that is used. The description in this section defines how to do it in the context of the Model Verifier and the C code generator. Section "Attributes" on page 1623 in Chapter 52, C++ Application Generator Reference describes the corresponding functionality when using the C++ code generator.*

When using the Model Verifier or C code generator an instance is added to the architecture simply by adding it to a composite attribute of an active object that contains a connector structure. The way this is done depends on the multiplicity of the attribute. It the attribute has multiplicity `[0..1]` a simple assignment is enough. If the attribute has a multiplicity > 1 then an append expression should be used. See examples below.

Performing new on a class or instance set with a restricted number of allowed instances (Multiplicity) is a request that will not be carried out if the maximum number of allowed instances would be exceeded. In this case the value NULL will be returned.

The dependency relationship can be used between parts in order to visualize that an instance in a part can create a new instance of another part.

**Example 65: New statements**

```
/* Type based creation based on the type 'a'.
The created instance is not inserted into any attribute
and can thus not be accessed using connectors/ports. */
new a;

/* Creation based on the type of the creator */
new this;

/* Creation plus assignment to a non-composite
attribute. Note that the instance can not be reached
using ports/connectors */
a aRef;
aRef = new a;

/* Creation plus assignment to a composite attribute.
The instance can be reached using the connector
structure of the creating object that contains the aPart
attribute (that is assumed to be defined as 'part
a[0..1] aPart;' */
aPart = new a;

/* Creation plus assignment to a composite attribute
with multiplicity >0. The instance can be reached using
the connector structure of the creating object. The
```

```
aMultPart attribute can for example be defined as 'part
a[*] aMultPart;'*/
aMultPart.append(new a);
```

## Save

It is often wanted to deal with arriving signals in a certain order. However, signals arriving from the outside world may not always arrive in the order that is expected. To temporarily save a signal in the signal queue, while looking for other signals to consume, the Save symbol should be used.

Several signals may be saved in each state, but if a saved signal is not handled in the next state, it again risks being discarded.

### Symbol

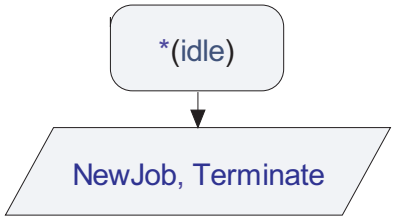



*Figure 144Using the save symbol*

The Save symbol saves signals from being discarded when being next to consumed in the state that does not handle the signal.

- This symbol should always be preceded by a State symbol.
- You cannot insert any symbols after the Save symbol.

### Syntax

**Example 66: Save**

A simple example:

```
save s;
```

Asterisk save:

```
save *;
```

# Stop

The Stop symbol stops the execution of the current instance. It is possible to delete an instance of an active class only from within the state machine of the class, by performing the stop action.

### Symbol

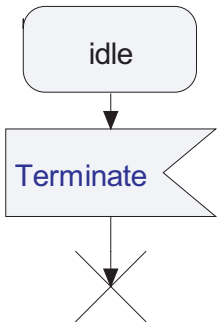



*Figure 145: Stop*

The stop action is handled in the following way:

1. If the instance is a simple state machine without any parts, the state machine will be immediately stopped.
2. If the instance contains parts, each of the part instances will be handled according to 1) above, as well as this instance.

# Return

The Return symbol finishes the execution of operations or substates and transfers the control to the calling context.

### Symbol

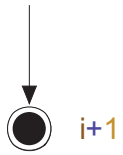



*Figure 146: Return in Operation*

### Syntax

**Example 67: Return simple example**

```
4+r
```

**Example 68: Return with exit point name in composite state**

```
exP2
```

If the operation has no return type or if the composite state exit is through the default exit point, the text field should be empty.

## Junction

Normally, the state and nextstate are sufficient mechanisms to split up a complex state machine into several diagrams. However, if a transition is very long, it might be necessary to split the description of the transition into several parts. This can be done by the Junction symbol, which is used both as a label and as a jump statement. Another reason for using the junction might be to avoid having crossing flowlines in a complex flow.

### Symbol

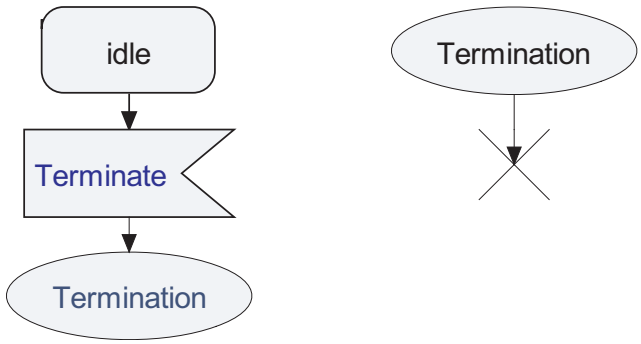



*Figure 147: Using the junction as a label and corresponding goto*

The Junction symbol corresponds to the label and join symbols but is also used in all cases where there is a need to merge flow lines.

- The Junction symbol can have more than one incoming flow line.

### Syntax

The symbol contains a text field.

## Flow

The Flow line connects two symbols in a transition.

- If you have a symbol selected in the drawing area, and then add another symbol from the toolbar while holding the <SHIFT> key down, then a flow line will automatically be created between the symbols.
- You can also create the line by drawing it from the line handle and connecting it to the next symbol.
- If you delete a symbol the line connected to the symbol will also be deleted.

## Simple transition

The Simple Transition line is used to define a transition when using a state-oriented style.

- You can draw a Simple Transition line from the State symbol only.
- You can create the line by drawing it from the line handle and connecting it to the next symbol.
- If you delete a symbol the line connected to the symbol will also be deleted.

### Syntax

There is one text field associated with a simple transition line. This text field describes both the trigger of the transition, the guard and the actions on the transition.

The trigger and guard follow the same syntax as is used in an Signal Receipt (Input) symbol. The actions follow the same syntax as in a Action (task) symbol with the exception that a short-hand is used to denote signal sending to save diagrams space: '^s' means the same as 'output s'.

Simple example

```
s1(x) / ^s;
```

Simple transition with guard

```
[ x>10 ] / myproc(x);
```

Both guard and signal receipt

```
s1 [ x>10 ] / myproc(x);
```

## Expressions

Expressions in UML are similar to expressions in most programming languages. As expected, expressions may contain references to variables (attributes), literals, constants and operations (calls).

Many expressions may be used as actions by appending a semicolon (;) after the expression. For example, the following expressions are commonly used as actions:

- Assignment Expression
- Call expression
- New expression
- Conditional expression

There are a couple of expressions for special variable access or creation of complex values:

- Field expression
- Index expression
- Instance expression
- This expression

There is also a group of expressions, that similar to variable access, depend on the underlying dynamic state of the system, they are often referred to as Imperative expressions:

- Arbitrary value (any) expression
- Now expression
- Pid expressions
  - Self
  - Sender
  - Parent
  - Offspring
- Timer active expression

Other expressions available are:

- Range check expression
- Target code expression

### Call expression

A call expression is used for calling operations. It may contain actual parameters for the operation call.

**Example 69: Call expression**

```
foo(3, true, "mmo")
```

The value of a call expression is the actual value of the operation's return parameter after the call. If the called operation has no return parameter, the call expression has no value, and must then only be used as a stand-alone expression in an expression action.

Before the operation call takes place the expressions provided as actual arguments will be evaluated. Note, however, that UML does not define the order in which the expressions will be evaluated. The actual evaluation order depends on which code generator that is used, and sometimes even on which compiler that is used for compiling generated code. Therefore, it is recommended that models do not depend on the evaluation order of actual arguments in call expressions.

**Example 70: Argument evaluation order is undefined**

```
foo(f1(), f2())
```

The operations in this example can either be called in the order 'f2', 'f1', 'foo' (right-to-left evaluation order) or 'f1', 'f2', 'foo' (left-to-right evaluation order).

**Note**

*When using the C code generator (including Model Verifier and Model Validator, but excluding AgileC) call arguments will always be evaluated from left to right, regardless of which target compiler that is used. Still it is not recommended to exploit this behavior since the evaluation order is not defined in the UML standard.*

### New expression

The new expression contains the new() construct as described in "New" on page 351.

### Conditional expression

A Conditional Expression has the form

```
expr_1 ? expr_2 : expr_3
```

where the first expression is of the boolean type and the second and third expressions are of the same type.

The expression `expr_1` is evaluated first. If it is true, then the expression `expr_2` is evaluated and provided as the resulting value of the conditional expression, otherwise `expr_3` is evaluated and given as result.

**Example 71: Conditional expression**

```
imax = ( i > j ) ? i : j; /* imax = max (i, j) */
```

### Field expression

The Field Expression is used to access a field of a structured datatype, that is an attribute of a class.

**Example 72: Field expression**

```
a.b = true;
test = a.b;
```

### Index expression

The Index Expression is used to access an element of an indexed datatype, typically an array or a string.

**Example 73: Index expression**

```
iarr[i, j] = 1;
i = iarr[k,l];
```

### Instance expression

The Instance Expression is used to create complex values in one operation. By this, it is possible to initialize a structured type in one operation, instead of initializing each field separately. Note, however, that constructors are recommended for initializing structured types.

**Example 74: Instance expression**

```
class sType {
    Integer Age;
    Charstring Name;
    Boolean MaleGender;
}
s = sType(. 'John', 44, true .);
```

Instance expressions are also used to describe stereotype instances containing tagged values.

### This expression

This refers to the current instance. If this is used in an operation of a passive class, this refers to the instance of the passive class. If this is used in an operation of an active class or in a state machine, this refers to the instance of the active class.

### Imperative expressions

Imperative Expressions include:

- Arbitrary value (any) expression
- Now expression
- Pid expressions
- State expression
- Timer active expression

### Arbitrary value (any) expression

The any Expression yields an arbitrary value of the provided type.

**Example 75: any expression**

```
anInt = any(Integer);
```

```
output resultSig(any(Boolean));
```

### Now expression

The Now Expression returns the current time value.

**Example 76: Now expression**

```
Time time_0 = now;
set(delayTimer, now + 10);
```

### Pid expressions

Pid expressions are expressions of the datatype Pid. A Pid Expression is either of **self**, **parent**, **offspring** or **sender**, as described in "Pid" on page 363.

**Example 77: Pid expressions**

```
currentClientId = sender;
new serverAgent;
if (offspring != NULL)
  output sender.serverId(offspring)
  else output sender.AllServersBusy;
```

### State expression

The State Expression can be used to check the most recently visited state in the current state machine. If the state machine contains composite states, the expression returns the most recently visited state of the nearest enclosing scope. The returned expression will be of the Charstring datatype. If no state has been visited, an empty string is returned.

**Example 78: State expression**

```
if (state  == "idle") return ;
```

### Timer active expression

The Timer Active expression is used to check if a named timer is active or not. A boolean value will be returned. A timer is active either if the timer has not expired yet or if the timer has expired but the timer signal has not been consumed yet (or discarded).

**Example 79: Timer active expression**

```
if (active(userTimeout)) reset(userTimeout);
```

### Range check expression

A Range Check Expression is used to check if an expression meets a value range condition at run-time. It has the form:

```
expr_1 in type type_ident
```

Where type_ident may be further restricted by a constraint. The range check expression will return a Boolean value depending on if the expression matches the provided type.

**Example 80: Range check expressions**

```
sender in type clientType;
intVar in type Integer constants (1..9, -9..-1);
age in type ageSyntype;
```

### Target code expression

A Target Code Expression is dependent of the selected implementation language and contains implementation language code that is not parsed by the UML parser, but instead added directly to the generated code.

Target code has the format

```
[[ target_code_details ]]
```

The target code (for example Inline C/C++) can contain any expression in the implementation language that matches the type that the UML context specifies.

If the target code contains the text

```
]]
```

this must be escaped by a # as

```
#]]
```

If the target code contains

```
#
```

this must be escaped by a # as

```
##
```

If it is needed to reference model entities from the target code, this has the form `#(name)` where `name` is an identifier in the model.

**Example 81: Target code expression**

```
Real side_a, side_b;
...
Real hypotenuse = [[ sqrt( pow(#(side_a),2) + pow(#(side_b),2) )]];
```

**See also**

"C Application" on page 978

# Pid

Each active class has access to four different Pid expressions, that can be seen as accessing implicit attributes belonging to the instance itself.

### Self

The Self expression returns a Pid value that refers to the instance.

### Sender

The Sender expression returns a Pid value that refers to the active object which this instance received its most recent signal from. If no signal has been received, the value `NULL` will be returned.

### Parent

The **parent** expression returns a Pid value that refers to the active object that created this instance. If this instance is statically created at system start-up, the value `NULL` will be returned.

### Offspring

The **offspring** expression returns a Pid value that refers to the most recent created active object. If no active object has been created (by this instance), the value `NULL` will be returned.

**Note**

*The implicit attributes corresponding to the Pid expressions in each active object are not globally updated when a state machine terminates.*

Pid values can be seen as references to active objects. The values are of the predefined datatype Pid, defined in the `TTDRTTypes` package.

**Hint**

*To have access to the* `TTDRTTypes` *package where the Pid datatype is defined, switch on the* `RTUtilities` *add-in. This is done from the Tools menu. Select Customize, then open the Add-Ins tab and select* `RTUtilities`*. Once this is done the* `TTDRTTypes` *package is available under the Library node in the Model View*

### References to active objects

Variables of the datatype Pid may be declared. However, such a variable may contain references to all kinds of active objects, which may be too unrestricted. Variables referencing active objects can be restricted by typing them as either

- an interface, or
- an active class.

This allows for static type checking to verify that a value will refer to an instance of a certain kind, for example when trying to perform an assignment.

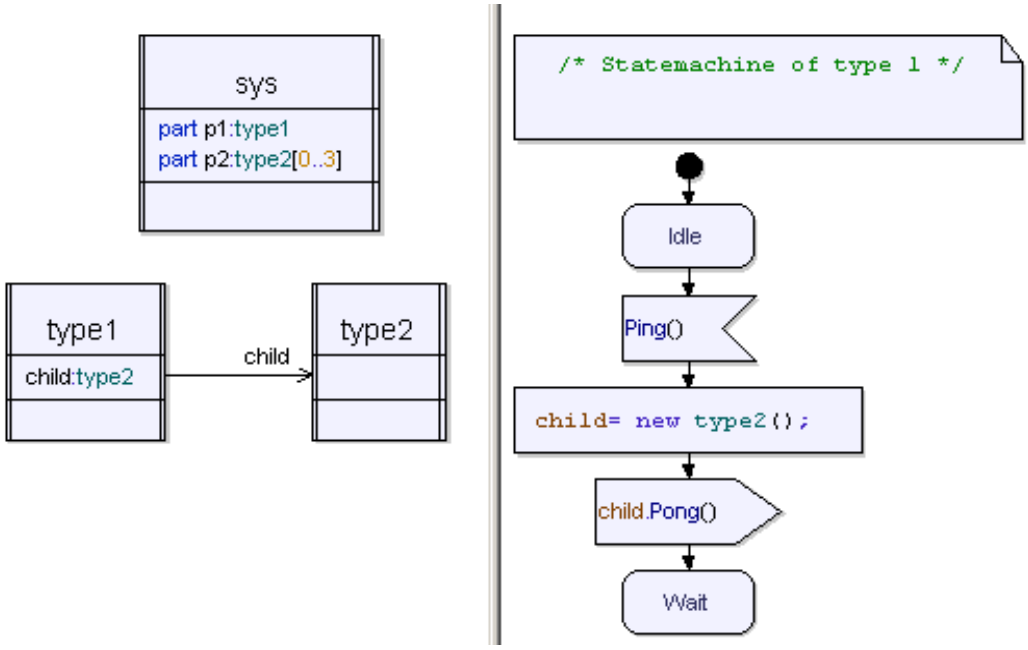



*Figure 148: Use of Pid variable*

## Timer handling and time

A timer is defined by the special Timer symbol or by a corresponding textual syntax. In the declaration of the timer it is allowed to specify a default duration, that is the time between the timer set and the timeout. If the timer needs to be cancelled, the reset action is used. A timer should be declared within the scope of the active class that the state machine handling the timer defines, typically in a class diagram of the active class.

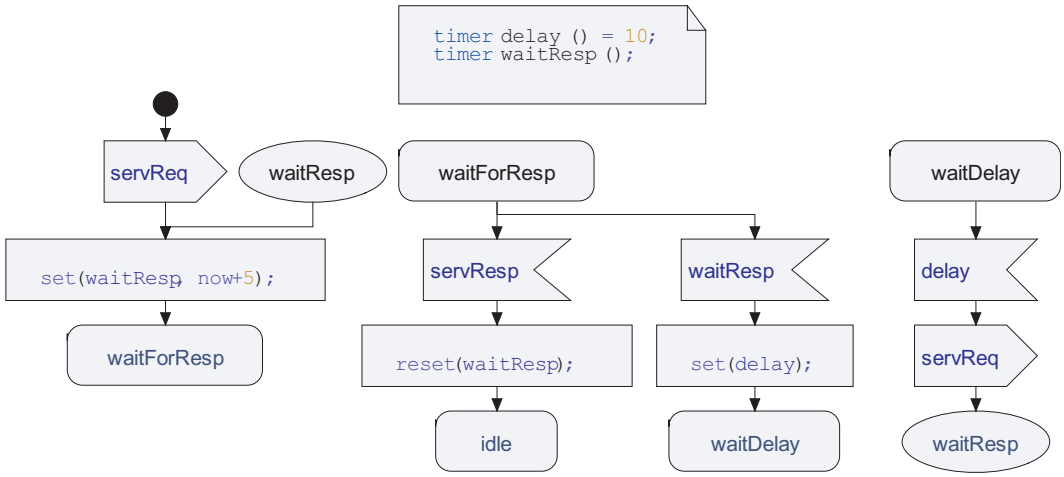



*Figure 149: Timer handling*

Timers are automatically monitored by the run-time system keeping track of all active timers. At the time of timeout, a timer signal is sent to the process who set the timer, who needs to consume the timer signal by a normal signal receipt in an appropriate state. Just like ordinary signals, timer signals that cannot be received in a state may be discarded.

**Hint**

*To have access to the* `TTDRTTypes` *package where the datatypes related to timers are defined, switch on the* `RTUtilities` *add-in. This is done from the Tools menu. Select Customize, then open the Add-Ins tab and select* `RTUtilities`*. Once this is done the* `TTDRTTypes` *package is available under the Library node in the Model View*

## Composite state

A composite state is a state which is composed by other states and transitions. While in any of the substates of the composite state, a trigger with a transition defined for the composite state will cause an exit of the composite state (and substates) for a new state.

A composite state can be created in two ways: either by an inline state machine definition or by referring to a state machine defined elsewhere.

A composite state can implicitly be created when a state machine diagram is created on a state.

A composite state is marked with a “rake” symbol in the upper right corner of the state symbol.

The composite state may have several entry and exit points, which are labelled.

Transitions in a substate has higher priority than transitions in an outer state. This applies both to transitions triggered by signals and to transitions triggered by timers.

This is in UML referred to as **transition overriding**.

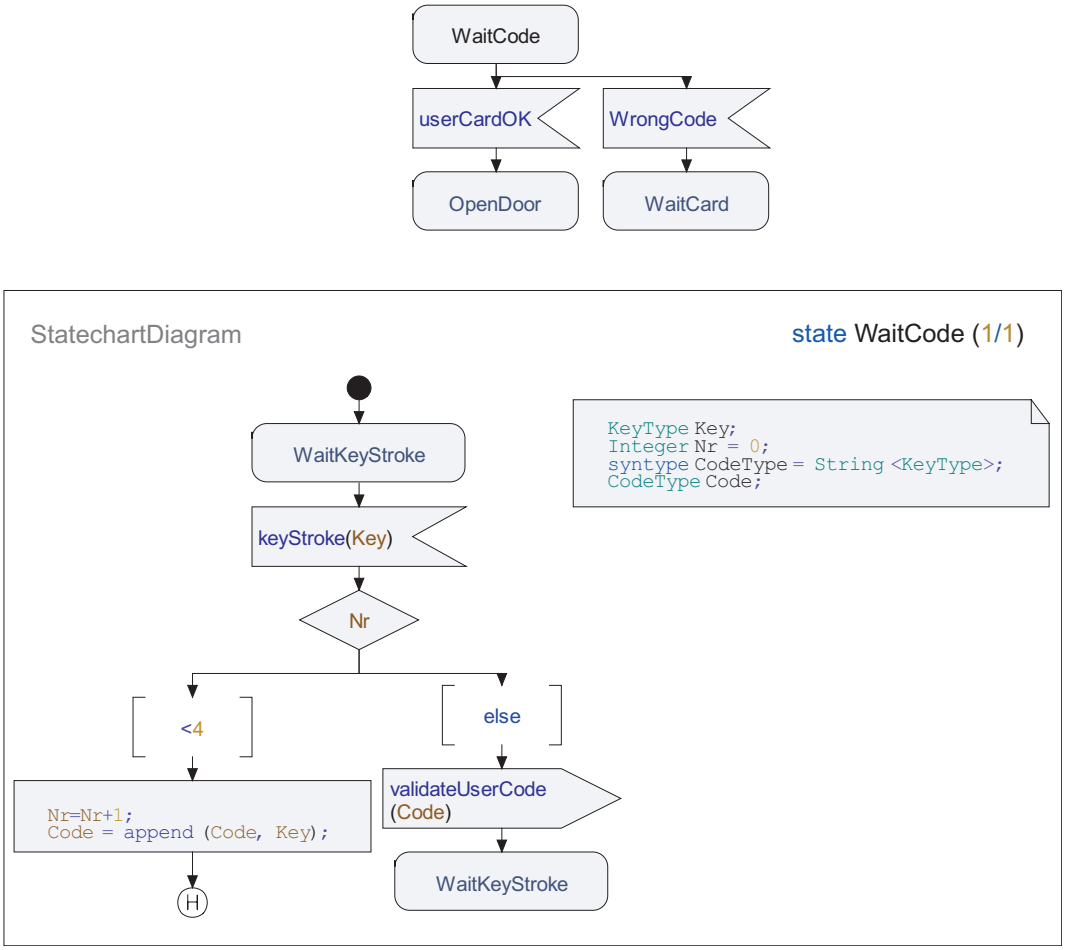



*Figure 150: Use of a composite state*

### Entry connection point

An Entry Connection Point is a named starting point for entering a composite state. Entry connection points are referenced at a start symbol inside a composite state and in the nextstate symbol when entering a composite state.

There must be at least one named or unnamed start symbol in a composite state. Not more than one start symbol can be named in a composite state.

**Hint**

*An Entry connection point is defined in the Model View by selecting a state machine and choosing the **New / Entry Connection Point** command in the shortcut menu.*

### Exit connection point

An Exit Connection Point is a named exit for leaving a composite state. Exit connection points are referenced at a return symbol in a composite state and at connect transitions leading out of composite states.

If there are more than one connect transition from a composite state, at most one of these connect transitions can be unnamed.

**Hint**

*An Exit connection point is defined in the Model View by selecting a state machine and choosing the **New / Exit Connection Point** command in the shortcut menu.*

## State machine inheritance

A State machine can be specialized, either directly by inheritance between state machines or by specialization of the active class that owns the state machine. A specialized state machine may add features or change features of the original state machine. Features that may be added include states, transitions, variables and other entities that can be declared in a state machine. In order for allowing a feature to be changed by specialization, it must be declared as **virtual** in the original state machine. A virtual definition may be redefined in the specialized state machine. The following concepts can be virtual (and thus redefined) in a state machine:

- Transitions
  - Start
  - Signal Receipt
  - Guard
  - Save
- Operation

## Operation body

An Operation Body is a method without states. The action is often a compound action, which contains a list of other actions.

**Hint**

*A text diagram is a convenient way to define an operation body. To do this simply right-click and choose New->Text Diagram from the shortcut menu for an operation in the Model View. Then type the textual definition of the operation body in the diagram.*

An Operation Body may be informal, meaning that the specification of how to execute it is not formally expressed in the UML language, but maybe in some other language. In that case the operation body will contain an informal expression containing the informal description.

**See also**

"State machine implementation" on page 369

"Internals" on page 370

"Implementation" on page 393

"Text diagram" on page 382

## State machine implementation

A State machine Implementation is a method containing states and everything else needed to realize the State machine signature. A State machine Implementation is typically implicitly defined when defining a State machine.

**See also**

"State machine" on page 332

"Internals" on page 370

"Implementation" on page 393

### Internals

Internals are used to be able to divide a class definition into one signature-oriented part and one implementation-oriented part and then store the signature for a class in a different file than the implementation of the class. The purpose of this is to facilitate component-based modeling by allowing separate version handling and delivery for the signature and the implementation.

**See also**

"State machine implementation" on page 369

"Operation body" on page 369

"Implementation" on page 393

### Text extension symbol

The text extension symbol can be connected to the action symbol to display the content of the action symbol. This is particularly useful when drawing transition oriented flows where an action with a large amount of text can disturb the overview of the diagram. The action code can be edited either in the action symbol or in the text extension symbol.

## Deployment Modeling

In deployment modeling, the run-time architecture of the system is modeled. It describes how deployable pieces of the software, Artifacts, are deployed onto Nodes representing physical computation resources. Deployment specifications are used to describe how artifacts are deployed onto nodes. Associations are used to model connections between nodes.

### Deployment diagram

A deployment diagram specifies a set of Artifacts deployed onto a set of interconnected Nodes. A Deployment specification is used to specify execution parameters used when deploying an artifact onto a node. An Execution environment can be used to model a node providing a set of services to the artifacts deployed onto it.

### Example

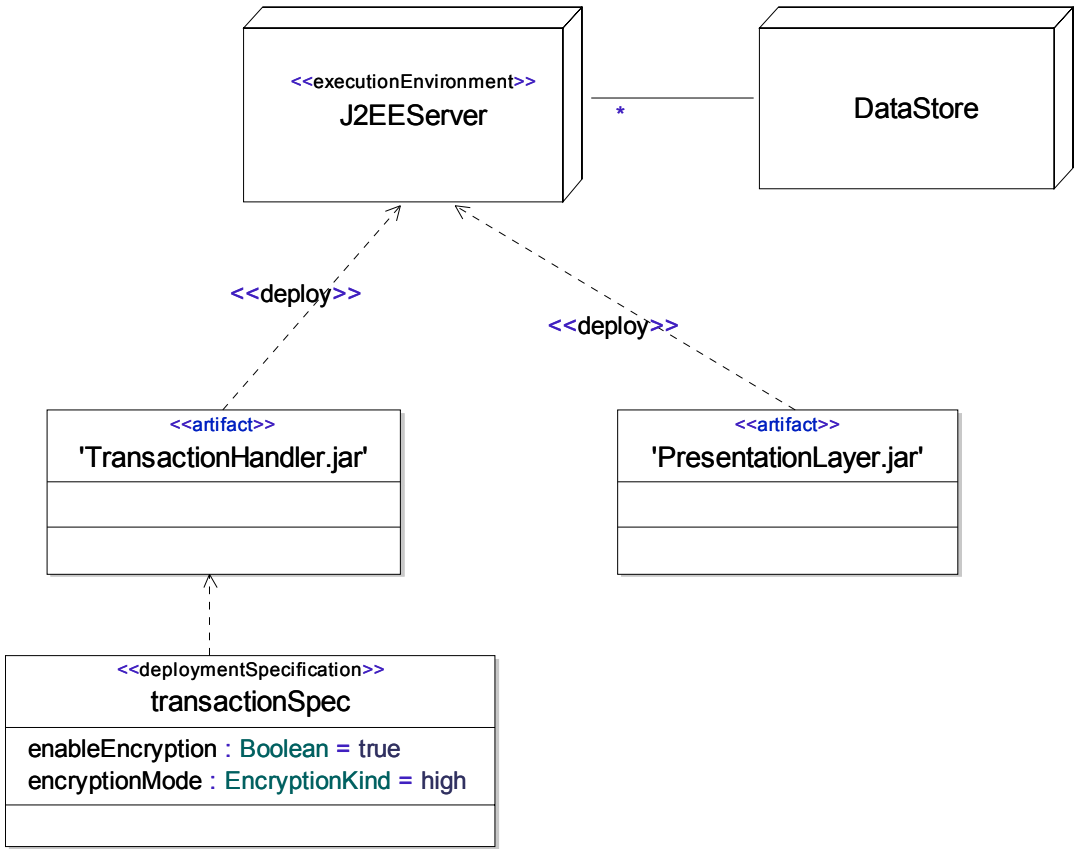



*Figure 151: Deployment diagram*

### Model elements in deployment diagrams

The following elements are found in deployment diagrams

- Artifact
- Node
- Execution environment
- Deployment specification
- Artifact
- Class
- Relationships

### See also

"Class diagram" on page 263

"Component diagram" on page 308

# Artifact

An Artifact represents a physical piece of information that is used or produced by a software development process. Examples of artifacts include source files, scripts, libraries and executable programs.

An artifact manifests a number of elements through Manifestation relations, meaning that the artifact is built up, or constructed from, these elements. For example, an artifact representing a header file in C++ can have a manifestation relation to the class declared in the header file. This information can then be used by a code generator when generating the physical header file from the model.

During deployment modeling, artifacts are deployed on nodes using the Deployment relationship.

Artifacts are similar to Classes and can have Attributes and Operations. Artifacts can also participate in the following relations: Dependency (of any element), Generalization (between artifacts), Composition (typically to other artifacts). In addition, an artifact is a namespace and can therefore own other model elements.

### Symbol

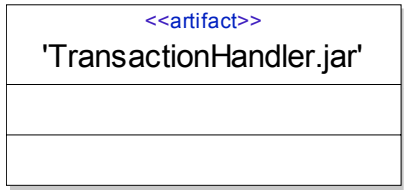



*Figure 152: Artifact symbol*

The artifact symbol is identical to the Class Symbol, with the keyword «`artifact`» added to the top.

# Node

A node is a named computational resource, typically a specific computer. Nodes can be connected using Associations to model network topologies.

### Symbol


*Figure 153: Node symbol*

### Syntax

A node is depicted as a 3-dimensional cube with the name inside.

## Execution environment

A special kind of Node offering an execution environment for the artifacts deployed onto it. The execution environment typically consists of a set of services required by the artifacts during execution.

A typical example is a J2EE server prepared for deployment of J2EE beans.

### Symbol

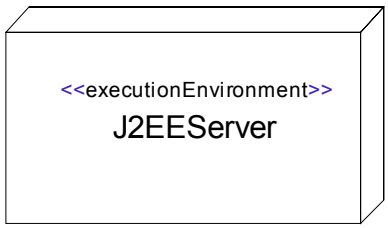



*Figure 154: Execution environment symbol*

### Syntax

Same as node with the stereotype «`executionEnvironment`» applied.

## Deployment specification

A deployment specification is used to specify a set of properties acting as execution parameters for an artifact when deployed onto a Node.

A deployment specification is applied to an artifact by drawing a Dependency from the specification to the artifact.

### Symbol

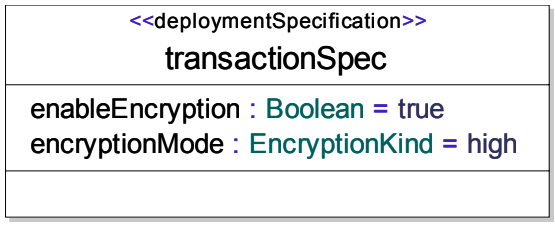



*Figure 155: Deployment specification symbol*

### Syntax

Same as class with the «`deploymentSpecification`» stereotype applied.

## Relationships

The following relationships can be used in Deployment diagrams:

- Deployment
- Manifestation
- Association
- Aggregation
- Composition
- Generalization
- Dependency

### Deployment

A special kind of Dependency used to deploy an artifact onto a deployment target, typically a Node. An artifact deployed onto a node will perform its execution in the context of that node.

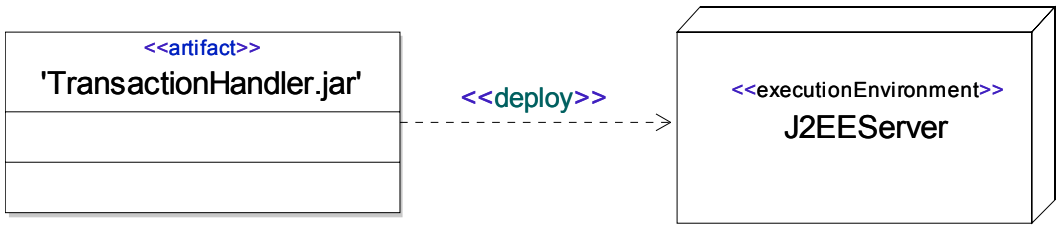



*Figure 156: Deployment dependency*

### Manifestation

Manifestation is a special kind of Dependency used from an Artifact to a set of other elements to describe that the artifact is built up, or constructed from, these elements.

For example, an artifact representing a header file in C++ can have a manifestation relation to the Class declared in the header file. This information can then be used by a code generator when generating the physical header file from the model.

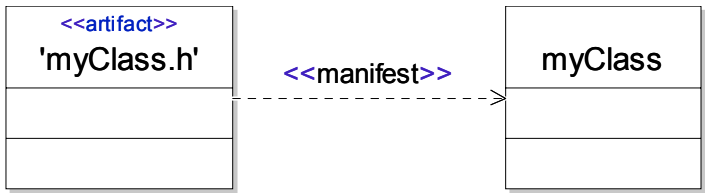



*Figure 157: Manifestation dependency*

# Relationships in UML

For general help on editing lines, please see:

"Draw lines" on page 196

"Move lines" on page 197

"Delete lines" on page 197

"Re-direct and bi-direct lines" on page 197

## Dependency

A Dependency is a relationship between two definitions, saying that one of these definitions (the *client*) is dependent on the other definition (the *supplier*) for some reason. The somewhat loose semantics of a Dependency makes it usable when the other relationship classes are inappropriate and cannot model a certain relationship.

There is one case when the dependency is used in a more specific way: indicating a creation relationship between instances of active classes, that is when an instance uses the New statement to create a new instance of a class. In this case, the dependency can be used between parts or between a part and the behavior symbol that refers to the state machine of the enclosing active class.

It is common to give dependencies a more detailed semantics by means of applying stereotypes on them. For example, see Import and Access dependencies.

## Generalization

A Generalization is a relationship between two Signatures (for example classes or operations), saying that one of these is a more general signature, and the other is a more specific one. The more specific signature inherits member definitions from the more general signature, and may also contain additional members. Because of this, the generalization relationship is also known as inheritance.

If a generalization is established between two types (for example two classes) the more specific type defines a subtype of the more general type (which is sometimes called a supertype). This means that an instance of the more general type may be substituted by an instance of the more specific type. In other words, a specialized type is assignment compatible with the more general type.

### Syntax

The generalization line has a text field, which may contain the discriminator.

## Realization

The Realization relationship is a special kind of the Generalization relationship. A Realization is used between a class and an interface to express that the realizing class conforms to (implements) the interface.

## Association

An Association is a semantic relationship between two or many Classifiers, indicating that instances of these classifiers will be related.

### Symbol

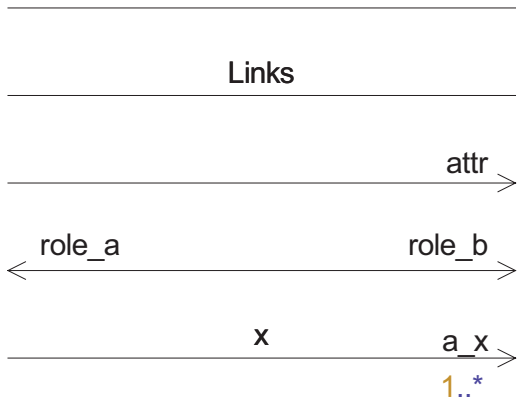



*Figure 158: Association*

The line contains one name field, two role name fields and two Multiplicity fields.

An Association has two association ends, represented as attributes. These attributes could either both be owned by the association (reflecting the situation when none of the associated classifiers are affected by the association), or one attribute could be owned by the association and one by the connected-to classifier C (reflecting the situation when the association is navigable only in the direction from C), or the attributes could be owned by one connected-to classifier each (reflecting the situation when the association is navigable in both directions). In the case when the association is unidirectional, the second (remote) Attribute will only exist if it is needed (for example if it carries a role name or a multiplicity).

An Association may also have properties that belong to the Association itself, and not to any particular association end.

An association can be navigable in both directions.

### Multiplicity

Multiplicity at an association end defines how many instances of the class that can be related by the association.

### Aggregation kind

An Association is either a normal association, an Aggregation or a Composition.

You can change aggregation type on the shortcut menu that is displayed when you click the ending parts of the line. The alternatives are Association, Aggregation and Composition. You must first add role names before you can select aggregation type.

- An Aggregation line specifies that an instance of the aggregate class is an informally considered owned by the instance of the component class.
- A Composition line specifies a stronger form of aggregation where the instance of the aggregate class exists only as long as the component class exists. The lifetime of the contained instance is thus strongly tied to the lifetime of the containing instance.

### Navigable end

A navigable end is an association end that is also an attribute of the classifier that is the type of the other end.

### Symbol

The line contains one name field, two role name fields and two Multiplicity fields.

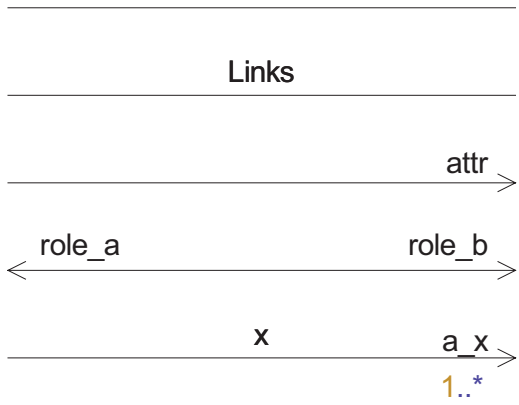



*Figure 159: Association*

## Examples

**Example 82: Role text**

```
+ myrole
```

**Example 83: Multiplicity text**

Infinite range:

```
*
```

Range condition:

```
0..3
```

Multiple range conditions:

```
1..7,>10
```

See also

"Attribute" on page 270

"Aggregation" on page 380

"Composition" on page 380

## Aggregation

Aggregation is a special kind of Association. It is a binary association that specifies an aggregation relationship (a whole/part relationship).

An aggregation has two ends, an aggregate end and a part end. An aggregation specifies that an instance of a classifier on the aggregate end aggregates an instance of the classifier at the part end. The aggregate instance may in turn be part of another aggregate.

An aggregation part may be part of more than one aggregate.

**Symbol**

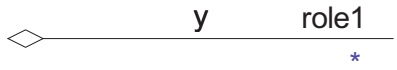



*Figure 160: Aggregation*

**See also**

"Attribute" on page 270

"Association" on page 377

"Composition" on page 380

## Composition

Composition is a special kind of Aggregation. The composite part is strongly owned by the composite and may thus only be part of one composition.

Composite parts that are typed by active classes can also be used as parts of the internal structure of a class as described by Composite structure diagrams.

### Symbol


*Figure 161: Composition and a corresponding attribute*

### See also

"Attribute" on page 270

"Association" on page 377

"Aggregation" on page 380

"Part" on page 300

"Composite structure diagram" on page 299

## Containment

The Containment relationship shows that one definition contains another definition. The contained definition appears in the scope of the container definition. When used between namespaces, such as packages, the containment relationship is sometimes also called namespace nesting.

### Symbol

The Containment line is drawn from the container definition to the contained definition, and shows a plus sign at the container side.

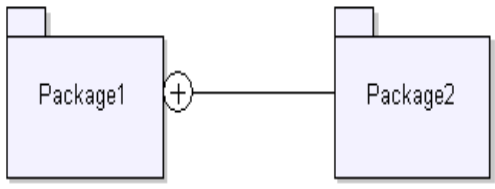



*Figure 162: Containment*

## Extension

Extension is used between a stereotype and a metaclass (a Metamodel class) to indicate that the stereotype extends the metamodel class.

## Association

### Description

Association is described in detail in the section Use Case Modeling.

# Text diagram

The text diagram can be used to show the textual syntax of the contents of a definition. For some definitions this is common and sometimes this can be an alternative to the graphical presentation in the regular diagrams. A typical example where it can be more practical to use the text diagram is when defining an Operation Body.

## Create a text diagram

Text diagrams can be included anywhere in your model where you could have any type of diagram containing a text symbol with formal UML information. A text diagram will act as a container for model elements and presentation elements, just like a text symbol in another diagram. To create a text diagram you select a suitable reference node in your model and from the shortcut menu select **New** and then **Text diagram**.

## Elements in text diagrams

There are two main ways of adding information to a text diagram.

- Type in the information into the diagram. The model is updated as you type.
- Use a drag-and-drop operation from the model in the workspace window **Model View** into the text diagram.

Entities in text diagrams are always model elements irrespective of how they have been created. This means that all editing will effect your model and any presentation elements.

It is possible to indent a selected block of text using TAB key. Use SHIFT + TAB key for negative indent.

**See also**

"Text parsing" on page 173 in Chapter 7, *Working with Diagrams*

UML Textual Syntax

# Common Symbols

## Frame

The symbols in a diagram are enclosed by the Frame symbol placed on the canvas.

- The frame has margins on all sides.
- You may resize and move the frame in all directions on the canvas, including the margins.

## Text symbol

The Text Symbol is used for defining variables, interfaces, datatypes etc.

It is not possible to connect lines to the symbol.

### Syntax

**Example 84: Including the definition of an interface and a syntype**

```
interface i {
  signal s;
}
syntype s = Integer;
```

**Example 85: Including the definition of a stereotyped class**

```
<<struct>> class X {
  private Integer I;
  void inc ( Integer incr ) {
    I = I + incr;
  }
```

```
}
```

## Comment

You use the Comment symbol to define comment text related to graphical symbols in a diagram.

Comments can also be made in the textual syntax.

### Comment symbol

The comment symbol is drawn similar to a Text symbol, but has a read-only text label in the upper left corner of the symbol. The text is set to "//", to distinguish the constraint symbol from for example a constraint symbol. It is possible to connect the symbol to another symbol with an Annotation line.

The comment symbol is connected on the left side but you can flip the symbol horizontally from the shortcut menu to connect it from the right side instead. When a Comment symbol in a diagram is not connected to any other symbol then the comment model element belongs to the element owning the diagram. If a Comment symbol is connected to two or more symbols in a diagram, then the comment model element belongs to the element owning the diagram

### Syntax

The text is informal and will not be syntactically checked.

**See also**

"Handling comments" on page 185

## Constraint

You use the Constraint symbol to define constraint text related to graphical symbols in a diagram.

Constraints can also be made in the textual syntax.

### Constraint symbol

The constraint symbol is drawn similar to a Comment symbol, but has a read-only text label in the upper left corner of the symbol. The text is set to "{}", to distinguish the constraint symbol from a comment symbol. It is possible to connect the symbol to another symbol with an Annotation line.

### Syntax

The text is informal and will not be syntactically checked.

## Stereotype instance

You use the Stereotype instance symbol to define stereotype instance text related to a model element.

Stereotype instance symbols can also be made in the textual syntax.

### Stereotype instance symbol

The stereotype instance symbol is drawn similar to a Comment symbol, but has a read-only text label in the upper left corner of the symbol. The text is set to "«»", to distinguish the constraint symbol from a comment symbol. It is possible to connect the symbol to another symbol with an Annotation line.

### Syntax

The text is informal and will not be syntactically checked.

## Annotation line

The Annotation line connects the Comment, Constraint and Stereotype instance symbol to another element.

You can draw an Annotation line from the line handle on the symbol and attach it to any symbol, inside the diagram frame, other than other Comment, Constraint and Stereotype instance symbols. It is also not allowed to attach it to a Text symbol.

# Extensibility

UML is a language that you can customize - in a controlled way. There are predefined mechanisms to extend UML constructs and to specialize them to a use for a specific purpose.

The extensibility mechanism of UML is based on the concept of Profile and Metamodel.

A metamodel is simply a special kind of UML package class model that is used to describe the information stored in a repository in a tool. A package is a metamodel if the package name is preceded by the keyword «metamodel». A metamodel typically contains a set of classes stereotyped by the keyword «metaclass» that define the metaclasses.

It is possible to define different metamodels and use the build-in repository to store user-level models based on these metamodels. The only requirement is that the metamodel must be possible to map to the object model used to define the run-time repository and storage.

A profile is a special kind of package, identified by the keyword «profile» before the package name in the heading. A profile contains a set of *stereotypes*, that have attributes (called *tagged value definitions*) and that extend one or more *metaclasses*.

In a user model the stereotype can be applied to an object that is an instance of the extended Metaclass. This will automatically make it possible to add values

## Metamodel

A metamodel is a set of metaclasses, metaattributes etc. that defines a conceptual view of the information stored in the model repository. The main practical usage of a metamodel is to form a basis for profile definitions.

A user profile can define stereotypes that extend the metaclasses in order to associate more information to model elements. The extra information is from a user's point of view editable using the Properties Editor and is stored in the model repository.

The UML tool set is able to represent different metamodels each giving a different view of a specific model.

**Hint**

*An example of a metamodel is given in the installation. Simply check the TTDMetamodel package in the Library node in the Model View. This package is a simple metamodel that describes the information stored.The purpose of the TTDMetamodel is to give a view that is very close to the underlying repository structure and each of the classes found in this metamodel corresponds directly to a core class found in the repository definition. However, the TTDMetamodel is a simplification of the core repository in the sense that only the classes that are useful to stereotype are included. Another simplification is that almost all of the associations and attributes found in the core repository model are omitted.*

## Metaclass

A metaclass is used to categorize a set of elements stored in a UML repository. It can be defined in metamodels using class symbols where the class name is stereotyped by «metaclass».

## Stereotype

A Stereotype is used to extend the information that can be stored in the model for a given entity. The extra information is described by the attributes of the stereotype.

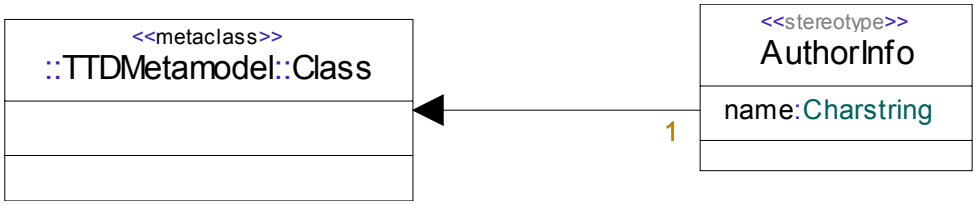



*Figure 163: Stereotype Example*

Figure 163 on page 387 shows an example of how to extend all classes with information about the author of the class definition. This is accomplished by defining a stereotype `AuthorInfo` with an attribute `name` that extends the Metaclass `TTDMetamodel::Class`.

### Tag definition

Tag Definitions are the attributes of a stereotype. When the stereotype is applied, the tag definitions are used by giving them specific values.

### Tagged value

Tagged values are the values that can be given tag definitions. These values are set using the Properties Editor.

### Showing Applied Stereotypes

It is possible to see stereotypes applied to a model element in the Model View. To see the stereotypes applied to a model element, do like this:

- On the **Tools** menu, click **Options**.
- In the **Options dialog** box, select the **UML Basic Editing** tab.
- Check the **Show stereotypes instances** check box.
- Click OK.

**See also**

"Extension" on page 388

## Profile

A profile is a special kind of package that is identified by the stereotype «profile». Profiles are used to extend the information that can be stored in a UML repository by defining stereotypes that extend metaclasses. Figure 163 on page 387 shows an example of a simple profile.

A profile is applied by using the package Import or Access constructs, for example if a model should adapt a certain profile, the top package of the model should have an import or access that references the package that defines the wanted profile.

## Extension

Extension is used between a Stereotype and a Metamodel class to indicate that the stereotype extends the metamodel class.

There is one text field associated with the extension line. This can have the text '1' or '`0..1`'. If the text is '1' then all elements that are instances of the extended Metaclass will automatically have the stereotype applied.

If the text is '`0..1`' then you will have to manually apply the stereotype. In Figure 163 on page 387 is an example of an extension line.

When the stereotype is manually applied, some symbols (class symbol, signal symbol etc.) will show the applied stereotype in the symbol.

# Predefined Data

The data modeling constructs in UML are powerful and allows for modeling and defining data in numerous ways. However, UML does not contain many built-in datatypes. Instead UML can be extended with different sets of datatypes depending on the application area. This is done by defining datatypes in model libraries (also often referred to as predefined packages).

For convenience there are different sets of predefined data that can be used:

### Predefined

This package contains generic datatypes with operations that always can be used.

### Profile TTDRTTypes

This package contains datatypes and operations supported only in the Model Verifier and in the C Code Generator.

### Profile TTDCppPredefined

This package contains datatypes and operations supported only in the C++ Application Generator.

**Hint**

*The details of the predefined packages are most easily checked in the tool itself. To access a predefine package you typically must switch on an add-in. This is done if you from the Tools menu select Customize, in the dialog go to the Add-Ins tab. This tab allows you to switch on the suitable profiles. To check the packages mentioned in this sections switch on the* `RTUtilities` *and* `CppTypes` *add-ins.*

**See also**

"Datatype" on page 288

# Predefined

The package Predefined is a proprietary extension to UML, which is always available in a project. This package is automatically used by the model defined in a project. The package defines a number of datatypes, but also a few other utilities.

Some of the datatypes exist in OMG UML (e.g. Integer or Boolean), but the Predefined package provides operations for these datatypes which is not done in the standard.

For each datatype, there is a set of operations to be applied on expressions of the type.

The package contains the following definitions:

| Kind | Definitions |
|---|---|
| Datatypes | Boolean, Character, String, Charstring, Integer, Natural, Real, Array, Any |
| Constants | PLUS_INFINITY, MINUS_INFINITY |

The Package Predefined is also available for inspection or browsing directly in the Model view. Each project has a node called predefined package. Expanding this node lets you browse the available datatypes, operators and other definitions.

### PLUS_INFINITY

PLUS_INFINITY is a constant of the datatype Real. It can be used as a reference to the largest Real number that can be used on host, or on a specific target.

### MINUS_INFINITY

PLUS_INFINITY is a constant of the datatype Real. It can be used as a reference to the largest negative Real number that can be used on host, or on a specific target.

## Profile TTDRTTypes

### None

The package also includes a predefined signal: `none`.

This signal is a system built-in signal used to model indeterministic behavior. It is only used in abstract specifications or model intended for simulation, not in models which applications should be built from.

The usage of `none` is as a trigger, to define an indeterministic transition (spontaneous transition). The cause of this event cannot be controlled; not when it occurs and not how frequently it occurs.

It can be controlled by the Model Verifier, though, during simulation. In the Messages window, it is possible to insert the signal `::NONE`.

# Metamodel Classes

A few of the more important metaclasses are described below.

### Metamodel profile

The TTDMetamodel is available for inspection and browsing directly in the Model view. When adding a project, there is always a node with the applied profiles, TTDMetamodel is one of these profiles. Expand Library node and the TTDMetamodel profile package to see the language model elements, abstract metaclasses and the relationships between these.

## Classifier

Classifier is a Metaclass in the UML language.

A Classifier is a description of data and is the Signature of a set of instances or Instance Sets. A Classifier defines a type, which for example may be the type of a `StructuralFeature`. A Classifier may be associated to other Classifiers by means of Associations.

Most class-like model elements are classifiers, including:

- Class
- Datatype, Syntype, Choice

- Stereotype
- Interface
- Collaboration

## Signature

Signature is a Metaclass in the UML language.

A Signature is an entity that can be the basis for the definition of another Signature. There are two main mechanisms that enable this:

- Specialization, or inheritance
- Parameterization

Specialization means that a super-signature may be specialized into a set of sub-signatures. Each sub-signature shares all the properties of the super-signature and may have some additional ones too. In the Metamodel the specialization mechanism is modeled by the Generalization class which is owned by Signature.

Parameterization means that a Signature may have a list of formal context parameters. Such a Signature is known as a *template*. Formal context parameters of a template may be substituted by actual context parameters when the template is instantiated (for example in a `TemplateTypeInstantiation`). Parameterization could make a Signature more flexible for use in different contexts. In the metamodel the parameterization mechanism is modeled by the `ContextParameter` class which is owned by Signature.

In addition to these two mechanisms for defining new Signatures based on another Signature, there is a third such mechanism that only one Signature has; the Syntype. This mechanism defines a new Signature by possibly constraining another one.

Some Signatures may have an Implementation. In that case the Signature acts as a façade for the Implementation, hiding all details which users of the Signature do not need to know. A façade allows for separating of a definition from its implementation and is what enables separate compilation of parts of a system. Compare for example with the use of header files in C programming. The following statements are true for a façade:

- A façade does not depend on its implementation.
- A façade does not depend on its uses. (This is in fact true for all Definitions.)

An implementation may only depend on façades.

The following model elements are signatures:

- Classifier
- Operation, signal, timer

## Implementation

An Implementation describes details about a Signature which users of the signature do not need to know about, but that are necessary from an execution point of view. While a Signature typically describes static properties of an entity, the corresponding Implementation is more concerned with the dynamic properties.

There are two main kinds of Implementations; Internals and Method. An Internals describes how a Class is structured, both physically and from a communication point of view, while a Method describes an Operation, a StateType, or a Class from a run-time execution point of view.

An Implementation only depends on Signatures (also referred to as façades), not on the usage of these Signatures. This is important in order to enable separate analysis of parts of a system.

## Method

A Method is the implementation of an Operation. It describes how it is executed at run-time. There are three kinds of methods, each of which has its own semantics of execution:

- Operation body – a stateless method which is executed by executing the Action of the OperationBody.
- State machine implementation – a method with states and transitions which is executed by executing the Action associated with a Transition that can be initiated in the active state.
- Interaction – a method which describes the interaction and information exchange between a set of attributes. Contrary to other methods, an interaction may not only provide a complete specification of how the operation shall be executed, but it may also be used to describe how it actually is executed (that is describing a trace), or provide a partial description of how it must execute (thereby putting semantic requirements on its other Methods).

- Activity implementation - a method executing a controlled set of small behavioral units.

## Signature and implementation

Signature and Implementation are two metaclasses in the UML language. A signature declares an entity and an implementation defines the same entity. The idea is that these concepts make it possible to separate the signature physically from the implementation (compare header files for C and C++).

The concepts for which it is possible to do this are:

### Operation

Operation signature and Operation body, Activity implementation, State machine implementation or Interaction.

### Activity

Activity signature and Activity implementation.

### State machine

State machine signature and State machine implementation.

### Class

Class signature and Internals.

# Collection Types and Multiplicity

There is a strong relation between the Multiplicity concept and collection types. This section details the aspects of this relationship.

## Implicit collections

When defining an attribute in UML the Multiplicity can be used to define that the attribute is multi-valued. From a UML point of view the multiplicity defines a constraint on the implementation of the attribute. An implicit instantiation of the String collection type is by default used as implementation. So, when using the attribute in action code you can use the operations defined for the String collection type.

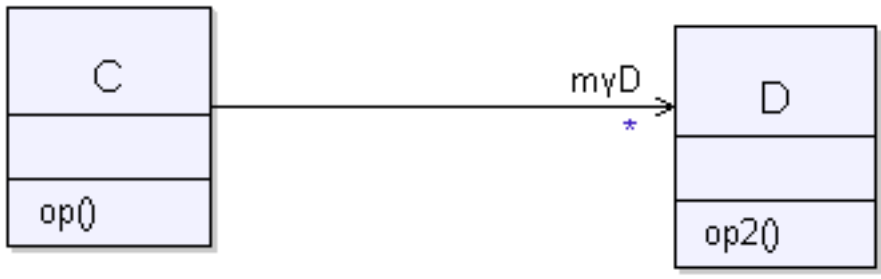



*Figure 164: Class with * multiplicity*

**Example 86: append, length and indexing operations available for String.**————

Consider the situation in Figure 164 on page 395, showing a class C with an attribute `myD` with multiplicity *.

When using the `myD` attribute in action code you can now use the operations available for the String collection type. You can for example define the `op` operation as in Figure 165 on page 396.

---

```
void op() {
  Integer len;
  myD.append(new D());
  myD.append(new D());
  len = myD.length();
  for ( Integer i = 1; i < len; i = i + 1) {
      myD[i].op2();
  }
}
```

*Figure 165: Operator definition*

## Changing the implicit collection type

In some situations you can decide to change the implicit type, in order to use a different set of operations or different implementation characteristics.

This can be done in two different ways; by using informal multiplicity or by using the <<containerType>> stereotype.

### Informal multiplicity

Informal multiplicity means that the UML multiplicity should not imply any special container type to be used for the collection. Instead the type of the attribute is assumed to be the container type.

Assume that you want to use the `Bag` collection type instead of the standard String.

If you use the property pages you should change the Type of the `myD` attribute to `Bag<D>` and select the `InformalMultiplicity` property. By selecting this option you make sure that the Multiplicity of the attribute is viewed as a constraint only and that the collection type is the specified `Bag<D>` and not the implicit String type. If informal multiplicity is not turned on, then the implicit collection type is still used and the complete type definition for the attribute would be `String< Bag<D> >`. However if you set the `InformalMultiplicity` property no implicit collection type is used.

If you make the change in the textual syntax used in a text symbol you change the definition of the attribute from `D[*] myD;` to `Bag<D> {[*]} myD;`. The curly braces syntax is used for the multiplicity to be interpreted informally as a constraint.

If you choose to do the change in the attribute compartment you would change the text from `myD:D[*]` to `myD: Bag<D> {[*]}`. Notice that the curly braces notation is also here used to show that the given multiplicity will be interpreted in an informal manner that will not imply an implicit collection type.

### The <<containerType>> stereotype

If you have a large number of attributes with non-single multiplicity and want to change collection type for all of them, it can be cumbersome to mark them all as having informal multiplicity. In that case you can instead use the predefined `<<containerType>>` stereotype. That stereotype can be applied on the Model level, or at any package or classifier in the model. It contains a tagged value 'Type' which specifies the implicit container type for all attributes contained in that scope.

If the specified container type is a template type it should have just one type template parameter. For example:

```
<<containerType (. Type = MyContainerType<Any> .)>>
package P {}
```

'Any' here refers to the actual type of the attribute, i.e. the element type of the collection.

It is possible to apply `<<containerType>>` on different scope levels in the model. If there exist more than one `<<containerType>>` instance in the scope path from an attribute to the Model node, the one that is closest to the attribute scope-wise will be used.

If you change container type you may have to invoke the 'Check All' command to force the model to be rebound, taking the new container type specification into account.

**Note**

*When creating a UML model targeted at a language in which no representation of the UML String type exists (e.g. Java or C#), the* `<<containerType>>` *stereotype is typically always applied (often at Model level) specifying an appropriate default container type that is available in the target language.*

## Multiplicity and composition

In general the aggregation kind is used to determine the lifetime dependency between two related objects. A composition implies a lifetime dependency, a reference (and also shared dependency) does not imply any lifetime dependency. Composition also implies that one specific instance may only be part of one container. From UML point of view this is only a static constraint, but from an implementation point of view several things can be deduced from this when combining composition with a static Multiplicity.

Similar to multiplicity Tau can use the aggregation kind either as formally or informally. If used formally Tau will automatically deduce an implementation. If used informally the aggregation kind is only viewed as an informal constraint on the model.

Tau uses by default a formal interpretation of composition plus static multiplicity to give an implementation with properties.

**Example 87: Multiplicity**

```
part A[1] a;
```

The 'a' object is allocated and terminated together with the container. For example assume that A is a passive class, then the C Code Generator would generate code where the attribute 'a' is written inline together with the containing C.

```
part A[4] a;
```

Four A objects are allocated and terminated together with the container. For example assume that A is a passive class, then the C Code Generator would generate code where the attribute 'a' is written inline together with the containing C struct, in the sense that it is generated as an array of A's as opposed to an array of pointers to A.

As with multiplicity the formal interpretation of aggregation kind can be turned off. It is controlled by the same mechanism as the formal/informal multiplicity. If informal multiplicity is defined it automatically implies informal aggregation kind and vice versa.

### Value<> template

If informal multiplicity/aggregation kind is used then you can define if you want value or reference semantics in the definition of the type of the attribute. This can be done using the Value<> template.

**Example 88: Value semantics**

```
Value<A> a;

String<Value<A>> {[4]} a;
```

**Example 89: Reference semantics**

```
A a;//

String<A> {[4]} a;
```

When using a composite attribute there are some aspects to be aware of as illustrated in the following example. In particular when mixing the usage of composition and usual references and when mixing Value instantiations and UML level composition.

**Example 90: Mixing composition and instantiation**

```
class C {}
class D {
Value<Value<C>> myC1; // Means the same as
                      // 'Value<C> myc1'
part Value<C> myC2; // Means the same as 'Value<C> myc2'
                    // and as 'part C myC2'
Value<C>[*] CList1; // Means the same as
                    // 'part C[*] CList1'
}
```

## Summary of multiplicity and collection types

To summarize the implicit generation of collection types and the usage of composition semantics:

- If an attribute has multiplicity >1 it will get an implicit String collection type.
- If an attribute has a fixed multiplicity AND it is has composite aggregation kind instances will automatically be allocated as part of the container (but in a code generator specific way)
- If the default implementation of multiplicity and aggregation kind is not what you want, it can be turned off using the 'Informal multiplicity' property available for the attribute.

**Note**

*These rules apply in general to all entities that can have a multiplicity, for example operation and signal parameters.*

### Value template in updated models

In earlier Tau versions the Value<> template was not present. When upgrading from these models the tool automatically adds Value elements where needed, see Example 91 on page 400.

**Example 91: Model upgrade**

Model from 2.2:

```
class Class2 { }
syntype Class2String = String < Class2 > constants
(0..10);
class Class1 {
    public part Class2String myPart;
}
```

Upgraded model:

```
class Class2 { }
syntype Class2String = String < Class2 > constants
(0..10);
syntype Class2String_Value = String<Value<Class2> >;
class Class1 {
    public Class2String_Value myPart;
}
```

# SysML

This section contains a listing of the diagrams and symbols of the SysML support in Tau.

SysML is a visual modeling language for specification, analysis, design, verification and validation of systems that may include hardware, software, data, procedures and facilities.

SysML is UML adapted for systems engineering.

SysML is a UML 2.x profile.

SysML is a response by the SysML Partners to an RFP issued by the Object Management Group (OMG) and co-sponsored by the International Council of Systems Engineering Model Driven System Design working group (INCOSE MDSD) and ISO Application Protocol for the interchange of systems engineering data (ISO AP-233) working group for a “UML for System Engineering” modeling language.

**Main goals of SysML**

- improve communications across the system development lifecycle
- enhance knowledge capture
- increase re-use of designs
- permit early verification of designs
- lower maintenance costs.

SysML will enable model based design, leading to consistent, unambiguous designs and specifications. Since SysML is based upon UML 2.0, the same model can be re-used as a specification and initial starting point for those parts of the system allocated to software. This can help to reduce errors and maintenance cost by bridging the gap between systems and software engineering.

SysML is UML adapted for systems engineering (hardware and software). With SysML, you can model things like:

- Requirements, see “Modeling Requirements” on page 1715
- Allocations, e.g. functional allocations from behavior to structure.
- Parametric constraints

### SysML in Tau

SysML is one of the Add-Ins distributed in the installation. To create a model using the SysML language this addin must be activated.

- From the Tools menu select Customize.
- Go to the Add-ins tab and check SysML.

  As a result a SysML menu will appear.

You can switch the Model View between the **SysML view** and the **Standard view**.

- From the View menu select **Reconfigure Model View** and point to the desired view.

**Note**

*The requirements part of SysML is uses the Requirements profile. For more details, see "Modeling Requirements" on page 1715.*

### SysML diagram types

- Activity diagram
- Block definition diagram (related to Class diagram)
- Internal block diagram (related to Composite structure diagram)
- Parametric block diagram (related to Composite structure diagram)
- Requirement diagram (related to Class diagram)
- Sequence diagram
- State machine diagram
- Use case diagram

## SysML diagrams and symbols

### Activity diagram

The activity diagram used in SysML is a normal UML Activity Diagram.

### Block definition diagram

Related to Class diagram, with the following symbols and lines:

- Artifact symbol

- Block symbol
- Parametric definition symbol
- Collaboration symbol
- Flow Port
- Flow Specification
- Interface symbol
- Operation symbol
- Package symbol
- Port symbol
- Primitive/enumeration symbol
- Realized interface symbol
- Required interface symbol
- Signal symbol
- State machine symbol
- Stereotype symbol
- Timer symbol
- Association/aggregation/composition line
- Dependency line
- Extension line
- Generalization/realization line
- Text symbol
- Comment symbol
- Constraint symbol
- Stereotype Instance symbol
- Text symbol
- Comment symbol
- Constraint symbol
- Stereotype Instance symbol

### Internal block diagram

Related to Composite structure diagram, with the following symbols and lines:

- Flow port symbol
- Part symbol
- Port symbol
- Binding line
- Connector line
- Text symbol
- Comment symbol
- Constraint symbol
- Stereotype Instance symbol
- Text symbol
- Comment symbol
- Constraint symbol
- Stereotype Instance symbol

### Parametric block diagram

Related to Composite structure diagram, with the following symbols and lines:

- Constraint Parameter
- Flow port
- Parametric Use
- Part symbol
- Port symbol
- Binding connector line
- Text symbol
- Comment symbol
- Constraint symbol
- Stereotype Instance symbol

### Requirement diagram

Related to Class diagram, with the following symbols and lines:

- Block symbol

- Package symbol
- Requirement symbol
- Allocate dependency line
- Association/aggregation/composition line
- Dependency line
- Derive dependency line
- Generalization/realization line
- Satisfy dependency line
- Text symbol
- Comment symbol
- Constraint symbol
- Stereotype Instance symbol

### Sequence diagram

The sequence diagram used in SysML is a normal UML Sequence diagram.

### State machine diagram

A normal UML State machine diagram:

### Use case diagram

The use case diagram used in SysML is a normal UML Use case diagram.

## Stereotypes on SysML diagram types

### Auto-applied

The following stereotypes are auto-applied on SysML diagram types.

- «`internalBlockDiagram`» (related to Composite structure diagram)
- «`parametricDiagram`» (related to Composite structure diagram)
- «`requirementDiagram`» (related to Class diagram)
- «`blockDiagram`» (related to Class diagram)

### Stereotype that can be applied to Diagram

«diagramDescription» with Version, Description, Reference and Completeness tagged values.

### Stereotypes that can be applied to Class

- «constraint» with an Equation tagged value.
- «block»
- «requirement» with Text and Id and tagged values.

### Stereotypes that can be applied to Comment

- «rationale»

### Stereotypes that can be applied to Dependency

- «allocate»
- «binding»
- «deriveReqt»
- «copy»
- «satisfy»
- «trace»
- «verify»

### Stereotypes that can be applied to ObjectNode and ActivityEdge

- «continuous»
- «discrete»
- «overwrite»
- «noBuffer»

### Stereotypes that can be applied to ActivityEdge

- «optional»
- «control»
- «stream»
- «probability» with a Value tagged value

### Stereotypes that can be applied to Operation and Activity

- «controlOperator»
- «nullTransformation»

### Stereotypes that can be applied to InformalConstraint

- «precondition»
- «postcondition»
- «resourceConstraint»
- A ControlValue enumeration: disable, enable.

## SysML reports

The following SysML specific reports are available:

- SysML Dependency Matrix
- SysML Dependency Report
- SysML Requirements Report
- SysML Requirements Gap Report

### SysML Dependency Matrix

Displays source and target elements of dependencies in a matrix. The different dependencies each has its own report. The following reports are available:

- All
- All - Reversed
- Allocate
- Allocate - Reversed
- Derive
- Derive - Reversed
- Satisfy
- Satisfy - Reversed
- Verify
- Verify - Reversed

The source elements are listed vertically and target elements horizontally, but if the reverse matrix is displayed the source is displayed horizontally and the target vertically.

### SysML Dependency Report

Lists all dependencies in a table. For each dependency, the following properties are displayed:

- Trace kind
- Source name
- Source kind
- Target name
- Target kind

The table can be saved in a .CSV file.

### SysML Requirements Report

Lists all requirements in a table. For each requirement, the following properties are displayed:

- Id
- Text

**See also**

"Requirement Reports" on page 1718

### SysML Requirements Gap Report

A requirements gap report in tabular format. For each requirement, the following properties are displayed:

- Id
- Text
- Incoming requirement dependencies
- Outgoing requirement dependencies
- Connected name
- Connected type

**See also**

"Requirement Reports" on page 1718

## Deprecated concepts

Since the SysML specification is still evolving, the set of implemented features is subject to change, and therefore some concepts have been deprecated.

They have been moved to a package called `SysMLdeprecated` and usage of them is strongly discouraged. The elements in this package (and any instances of them, such as stereotype instances) will be removed in future releases.

To avoid losing data from any of these concepts, the data must be transferred someplace else either manually or programmatically by using the APIs.

The following concepts have been changed or deprecated:

- «requirement»

  The «requirement» stereotype has been replaced with the one from the Requirements profile. The new version has only two attributes, Text and Id. Instances of the old stereotype has been replaced with two instances, one of the new requirement stereotype and one of the «requirementDeprecated» stereotype. The tagged values from the original «requirement» stereotype missing in the new one, are found in this instance.
- verifyMethodKind
- riskKind
- optimizationDirectionKind
- «effectiveness»

# Profile for Schedulability, Performance, and Time

This section lists all stereotypes, tagged values and enumerations of the **UML Profile for Schedulability, Performance, and Time** also commonly referred to as the UML Real-time profile.

**Note**

*Some tagged values can only be edited using the textual syntax. These are marked as italic in this document.*

## RTresourceModeling

### GRMacquire

GRMblocking : Boolean

### GRMcode

### GRMrealize

*GRMmapping : GRMmappingString*

### GRMdeploys

### GRMrelease

### GRMrequires

## RTtimeModeling

### RTaction

*RTstart : RTtimeValue*

*RTend : RTtimeValue*

*RTduration : RTtimeValue*

**RTclkInterrupt**

**RTstimulus**

*RTstart : RTtimeValue*

*RTend : RTtimeValue*

**RTclock**

RTclockId : Charstring

**RTdelay**

**RTevent**

*RTat : RTtimeValue*

**RTinterval**

*RTintState : RTtimeValue*

*RTintEnd : RTtimeValue*

*RTintDuration : RTtimeValue*

**RTnewClock**

**RTnewTimer**

*RTtimerPar : RTtimeValue*

**RTpause**

**RTreset**

**RTset**

*RTtimePar : RTtimeValue*

**RTstart**

**RTtime**

RTkind : RTkindEnum

**RTtimeout**

**RTtimer**

*RTduration : RTtimeValue*

RTperiodic : Boolean

**RTtimeService**

**RTtimingMechanism**

RTstability : Real

RTdrift : Real

RTskew : Real

*RTmaxValue : RTtimeValue*

RTorigin : Charstring

*RTresolution : RTtimeValue*

*RToffset : RTtimeValue*

*RTaccuracy : RTtimeValue*

*RTcurrentVal : RTtimeValue*

**RTkindEnum**

Literals:

- dense
- discrete

# RTconcurrencyModeling

### CRaction

CRatomic : Boolean

### CRasynch

### CRconcurrent

### CRcontains

### CRdeferred

### CRimmediate

CRthreading : CRthreadingEnum

### CRmsgQ

### CRsynch

### CRthreadingEnum

Literals:

- local
- remote

# SAprofile

### SAaction

SApriority : Integer

*SAblocking : RTtimeValue*

*SAdelay : RTtimeValue*

*SApreempted : RTtimeValue*

*SAready : RTtimeValue*

*SArelease : RTtimeValue*

*SAworstCase : RTtimeValue*

*SAabsDeadline : RTtimeValue*

SAlaxity : SAlaxityEnum

*SArelDeadline : RTtimeValue*

**SAengine**

SAaccessPolicy : SAaccessControlPolicyEnum

*SAcontextSwitch : TimeFunction*

SAschedulable : Boolean

SApreemptible : Boolean

*SApriorityRange : Range*

SArate : Real

SAschedulingPolicy : SAschedulingPolicyEnum

SAutilization : Real

SAaccessPolParam : Real

**SAowns**

**SAprecedes**

**SAresource**

*SAacquisition : RTtimeValue*

SAcapacity : Integer

*SAdeacquisition : RTtimeValue*

SAconsumable : Boolean

SAaccessControl : SAaccessControlPolicyEnum

SAptyCeiling : Integer

SApreemptible : Boolean

SAaccessCtrlParam : Real

**SAresponse**

SAutilization : Real

*SAspare : RTtimeValue*

*SAslack : RTtimeValue*

SAoverlaps : Integer

**SAschedRes**

**SAscheduler**

SAschedulingPolicy : SAschedulingPolicyEnum

**SAsituation**

**SAtrigger**

SAschedulable : Boolean

*SAoccurrence : RTarrivalPattern*

SAendToEnd : Charstring

**SAusedHost**

**SAuses**

**SAlaxityEnum**

Literals:

- hard
- soft

**SAschedulingPolicyEnum**

Literals:

- rateMonotonic
- deadlineMonotonic
- HKL
- fixedPriority
- minimumLaxityFirst
- maximizeAccruedUtility
- MinimumSlackTime

**SAaccessControlPolicyEnum**

Literals:

- FIFO
- priorityInheritance
- noPreemption
- highestLockers
- priorityCeiling

# PAprofile

**PAclosedLoad**

*PArespTime : PAperfValue*

PApriority : Integer

PApopulation : Integer

*PAextDelay : PAperfValue*

**PAcontext**

**PAhost**

PAutilization : Real

PAschdPolicy : PAschdPolicyEnum

PArate : Real

*PActxtSwT : PAperfValue*

*PAprioRange : Range*

PApreemptable : Boolean

PAthroughput : Real

**PAopenLoad**

*PArespTime : PAperfValue*

PApriority : Integer

*PAoccurrence : RTarrivalPattern*

**PAresource**

PAutilization : Real

PAschdPolicy : PAschdPolicyEnum

PAcapacity : Integer

*PAaxTime : PAperfValue*

*PArespTime : PAperfValue*

*PAwaitTime : PAperfValue*

PAthroughput : Real

**PAstep**

*PAdemand : PAperfValue*

*PArespTime : PAperfValue*

PAprob : Real

PArep : Integer

*PAdelay : PAperfValue*

*PAextOp : PAextOpValue*

*PAinterval : PAperfValue*

**PAschdPolicyEnum**

Literals:

- FIFO
- priority

## RSAprofile

**RSAclient**

*RSAtimeout : RTtimeValue*

RSAclPrio : Integer

RSAprivate : Integer

**RSAconnection**

RSAshared : Boolean

RSAhiPrio : Integer

RSAloPrio : Integer

**RSAmutex**

**RSAorb**

**RSAserver**

RSAsrvPrio : Integer

**RSAchannel**

RSAschedulingPolicy : RSAschedulingPolicyEnum

*RSAaverageLatency : RTtimeValue*

**RSAschedulingPolicyEnum**

Literals:

- FIFO
- RateMonotonic
- DeadlineMonotonic
- HKL
- FixedPriority
- MinimumLaxityFirst
- MaximizeAccruedUtility
- MinimumSlackTime

# 9

# *Error and Warning Messages*

This document is a reference guide to error and warning messages from the UML tool set.

# General Application Errors and Warnings

## Tau minidumps (Windows)

Tau has built in debug information capturing capabilities on the Windows platform. If at anytime during running the tool you receive a window saying Tau has crashed and a minidump has been created please contact your local Tau support. The minidump contains the current call stack and can help identify which calls have been made. This can help to identify if an error occurred due to internal tool calls and in these cases may make it possible to resolve problems not already identified. With consideration to dependencies on operating systems and third party calls this information can also be of help to improve integrations to the environment and publish clearer requirements for third party software which Tau is dependent on.


*Figure 166: Using special characters in identifiers*

### Minidump location

The minidumps are by default created in a local settings directory but can be relocated using an environment variable.

Default location:

```
C:\Documents and Settings\<user>\Local Settings\Temp
```

Environment variable example:

```
TAU_DUMP_PATH=c:\DevTools\Telelogic\minidumps\
```

### Minidump contents

The minidumps only contain the call stack and registers, and no memory. this means that there is no information about the model that the minidump originated from.

# Errors and Warnings from Build

It should be observed that there can be restrictions on constructions that may be allowed in your UML design, but that will not be allowed to generate code from. These restrictions may vary for different build types (code generators).

### Phases and identifiers

There are several phases involved when transforming a UML model to another language, or format. During the processing of the model, error and warning messages may be presented from each phase to help you identify where the problems occur. The Verbose mode may be useful to switch on to get as much information as possible from the build process. The prefixes identifying the phases are:

- TSX: Syntax Analysis
- TSC: Semantic Check
- TNR: Name Resolution
- TAB: Application Build
- TCI: C/C++ Import
- TIL: Intermediate Language
- TCC: C Code Generation
- TCG: C++ Generation
- TSI, OGC: SDL Import
- TUI: UML 1.x Import

### TSX: Syntax Analysis

The syntax analysis checks how language elements are constructed and put together in order to form correct UML constructions.

### TSC: Semantic Check

The semantic check verifies that the UML model is complete and that the relations between language constructs are meaningful.

Complete listing of the semantic checks per code generator stereotype.

### TNR: Name Resolution

The name resolution identifies names of the UML entities and attempts to bind them to the correct definition in the model.

### TAB: Application Build

The application builder manages the entire process of generating another representation or application from the UML model.

### TCI: C/C++ Import

The C/C++ import places external C or C++ header files into UML packages and transforms the data type declarations into a UML data model.

### TIL: Intermediate Language

The intermediate language phase makes any necessary transformations from the UML model to an intermediate representation required for efficient C code generation. The intermediate model is then used by the C Code Generator which outputs C code and the necessary makefiles to build an application.

### TCC: C Code Generation

The intermediate model is used as input in which the C Code Generator outputs C code and the necessary makefiles to build an application. Most output from this stage will result in compiler errors or warnings, but some messages will also be visible from the generator itself.

### TCG: C++ Generation

The C++ generation creates a set of C++ declarations based on the UML model.

### TSI, OGC: SDL Import

Error Messages from the SDL import operation will be printed in the Script tab of the Output window.

### TUI: UML 1.x Import

Error Messages from the UML import operation will be printed in the Script tab of the Output window.

# TSX: Syntax Analysis

The syntax analysis checks how language elements are constructed and put together in order to form correct UML constructions.

The direct cause of syntax errors will in most cases be possible to locate in the UML model.

Errors and warnings from this phase are prefixed with TSX.

```
Internal error: <string>
```

These kinds of errors should not appear. If they do, please contact Tau Support.

## TSX0026: Port should not contain two in or two out parts

This error does not occur by normal usage of the tool. If incorrect customization or Add-Ins have been used, they can create such models. The correction should then be made in the customization or add-in.

## TSX0047: Tagged values are not allowed here

In some places, for example inside a class symbol, you are prohibited to edit properties (Tagged values). Only the stereotype itself can be added.

The preferred way to edit properties is by using the Properties Editor.

# TSC: Semantic Check

## About semantic checks

The semantic check verifies that the UML model is complete and that the relations between language constructs are meaningful.

Semantic errors occur when there are incomplete constructs in your model. The UML Language Guide can be useful to identify supported constructs.

Errors and warnings from this phase are prefixed with TSC.

Complete listing of the semantic checks per code generator stereotype.

## TSC0123: A cyclic dependency was found in definition of the %n. (via <string>)

This is a cyclic dependency error. Since two classes cannot be containers for the other one at the same time, this is illegal.

The following is an example of this error:

**Example 92**

```
class X {
    part Y y;
}

class Y {
    part X x;
}
```

## TSC0134: Incomplete transition. A transition must end with stop, nextstate or join action

A decision must cover all answer possibilities, including 'else'.

## TSC0092: A corresponding 'virtual' or 'redefined' operation was not found in the parent signatures (or parent signatures does not exist).

There are a number of situations that may be the cause of this error. The following examples shows the situations which can occur.

Using a redefined operation in an active class that does not have generalizations:

**Example 93: Class without generalizations.**

```
active class P {
  redefined void Op() { }
}
```

Using a redefined operation in a generalization of an active class can cause this error:

**Example 94: No matching operation in the parent class.**

```
active class P {
}
active class C : P {
     redefined void Op() { }
}
```

When the operation (`Op`) in the parent class has a different signature there can be the following situations:

**Example 95: Virtuality must be “virtual” or “redefined”.**

It is not possible to redefine non-virtual operations.

```
active class P {
     void Op () {   }
}
active class C : P {
    redefined void Op() { }
}
```

**Example 96: Different return type.**

```
active class P {
```

```
    virtual Integer Op () { return 1; }
}
active class C : P {
    redefined void Op() { }
}
```

**Example 97: Different count of formal parameters.**

```
active class P {
    virtual void Op (Integer x) {  }
}
active class C : P {
    redefined void Op() { }
}
```

**Example 98: Different type of formal parameters.**

```
active class P {
    virtual void Op (Integer x) {  }
}
active class C : P {
    redefined void Op(Real x) { }
}
```

# TSC0196: A finalized operation cannot be redefined.

Operation in the parent class is finalized, but it has the same signature as in the child.

**Example 99: Finalized operation**

```
active class P {
    finalized void Op () { }
}
active class C : P {
    redefined void Op() { }
}
```

# TSC0236: Operation '<name>' cannot be specified as 'Realized' on a port.

The check will detect the following case:

```
active class <class name>
```

```
{
    port <port name> in with <in_name>;
}
```

where `<in_name>` is bound to some operation with the same name.

**Example 100**

```
active class a {
   void foo()  {}
   port p in with foo;
}
```

This will be reported as an error. To remedy this, `foo()` must be defined in an interface to the active class a.

---

# TSC0237: Operation '<name>' cannot be specified as 'Required' on a port.

The check will detect the following case:

```
active class <class name>
{
    port <port name> out with <out_name>;
}
```

where `<out_name>` is bound to some operation with the same name.

**Example 101**

```
active class a {
   void foo()  {}
   port p out with foo;
}
```

This will be reported as an error. To remedy this, `foo()` must be defined in an interface to the active class a.

---

# TSC2300: Expression 'any (type)' cannot be of interface or state machine type

The following is an example of this error:

**Example 102**

```
interface I {
}

active class X {
    Integer Op () {
        switch (any (I)) {
            case 5  : { return 1; }
            default : { return 0; }
        }
    }
}
```

## TSC2302: An association from a datatype may not have a navigable remote association end

Since datatypes cannot have attributes, it is illegal to have an association from a datatype. The navigability must always be to the datatype.

This error does not occur by normal usage of the tool. If incorrect customization or Add-Ins have been used, they can create such models. The correction should then be made in the customization or add-in.

## TSC2303: At most one association end may be aggregate or composite

Since aggregation and composition are different kind of "part-of" constructs, two classes cannot be containers for each other.

**Example 103**

This situation could occur in the situation shown in Figure 167 on page 431.

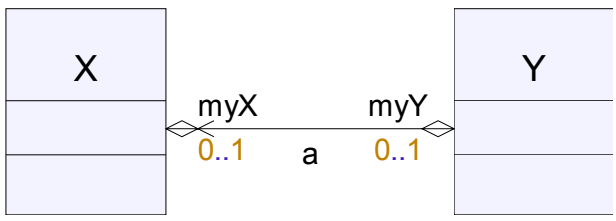



*Figure 167: Classes with circular references.*

## TSC2304: An attribute that is not a part may not have initial count

In UML it is not possible to specify the initial count for regular attributes. That is something that is only possible for parts.

The following is an example of this error:

**Example 104**

```
class Z {
    Integer [1..*] a / 1;
}
```

## TSC2305: A part cannot have a default value

Parts are instances of active classes and they cannot have default values. The following is an example of this error:

**Example 105**

```
active class X {
    part Y a = 10;
}
```

## TSC2306: A composite attribute or association end may not be typed by a datatype

Composite attributes also known as parts in UML must not be instances of datatypes.

The following is an example of this error:

**Example 106**

```
class X {
    part Integer d;
}
```

## TSC2307: A composite attribute may not have a type, which owns this attribute (directly or indirectly)

This is a cyclic dependency error. Since a class cannot be a container for itself this is illegal.

The following is an example of this error:

**Example 107**

```
class X {
    part X y;
}
```

## TSC2308: The 'via' of a call expression should reference either a port or a connector

The following is an example of this error:

**Example 108**

```
class Y {}
signal sig ();
active class X {
    port p out with sig;
    void Op () {
        output sig via Y;
    }
}
```

## TSC0269: Generalization between 'Interface I' and 'Class Y' is not allowed

The following is an example of this error:

**Example 109**

```
class Y {
}
interface I : Y {
}
```

## TSC2325: Cyclic inheritance

This error is caused if a Signature is based on itself, directly or indirectly.

The following is an example of this error:

**Example 110**

```
class X : Y {
}
class Y : X {
}
```

## TSC4001: When generating C code, return values must be handled in left hand side of assignment expression

Return values from for example value returning operations must not be ignored. Such return values must be saved in for example an attribute.

**Example 111**

Consider an Operation `Op`, returning an Integer:

```
Op ():Integer
```

Call to `Op`:

```
...
Integer i;
...
i=Op(); // Correct way of calling Op
Op(); // Error is reported
...
```

This check is performed only when the semantic checker is run in the context of a build which involves any of the C code generators and build types. (Model Verifier, C Code Generator and AgileC Code Generator).

# TNR: Name Resolution

The name resolution identifies names of the UML entities and binds them to the correct definition in the model. Name resolution errors are caused by inconsistencies in your model. This may happen when you change names on entities in such a way that ambiguities occur and can not be resolved in a deterministic way.

Errors and warnings from this phase are prefixed with TNR.

TNR errors where the **Subject** of the error refers to the project file (`.ttp` file) rather than to a UML entity, should be reported to Tau Support.

## TNR0023: Failed to locate element referred by: <name>

Entities can be located by its name (name binding) or by its GUID (GUID binding).

Name binding uses the name to refer to an entity in the current scope. GUID binding means that an entity is referred by its unique id (GUID). That means that this error occurs if an entity for some reason is removed and it is referred somewhere in the model by its GUID.

Solutions are to load the entity with the correct GUID, remove the reference or change the reference so it uses name binding.

# TAB: Application Build

The application builder is a Tau application that manages the entire process of generating another representation or application from the UML model.

Errors from the toolset responsible for this process could be related to an incorrect set-up of your development platform. Errors that are not related to the restrictions found in the release notes should be reported to Tau Support.

Errors and warnings from this phase are prefixed with TAB.

# TCI: C/C++ Import

The C/C++ import places external C or C++ header files into UML packages and transforms the data type declarations into a UML data model.

This phase works on external data models, which must be correct with respect to C/C++. Details of supported constructs and the resulting UML model are found in "Operation Principles" on page 546 in Chapter 15, *C/C++ Import*.

Errors and warnings from this phase are prefixed with TCI.

# TIL: Intermediate Language

The intermediate language phase makes any necessary transformations from the UML model to an intermediate representation required for efficient C code generation. The intermediate model is then used by the C Code Generator.

This phase operates on a model, which has been found semantically correct with respect to UML. Errors that are not related to restrictions found in the release notes should be reported to Tau Support.

To help diagnose the situation, please send a complete model to Tau Support. so that the case can be addressed promptly.

To assist support in solving the situation, you can set the environment variable TTD_INTERMEDIATE_U2_MODEL to 1 and rerun the build. This will result in saving on file an intermediate model which represents the transformations needed to generate code. The file is named `intermediate_model.u2` and is located in folder defined as Target Directory. Including this with your UML model will help support resolve the problem promptly.

Errors and warnings from this phase are prefixed with TIL.

# TCC: C Code Generation

The intermediate model is used as input in which the C Code Generator outputs C code and the necessary makefiles to build an application. Most output from this stage will result in compiler errors or warnings, but some messages will also be visible from the C Code Generator itself.

To help diagnose the situation, please send a complete model to Tau Support. so that the case can be addressed promptly.

Errors and warnings from this phase are prefixed with TCC.

# TCG: C++ Generation

The C++ generation creates a set of C++ declarations based on the UML model.

This phase works on a model, which has been found semantically correct with respect to UML. Errors that are not related to restrictions found in the release notes should be reported to Tau Support.

To help diagnose the situation, please send a complete model to Tau Support. so that the case can be addressed promptly.

Errors and warnings from this phase are prefixed with TCG.

# *UML for Model Verification*

The chapters listed under UML for Model Verification describe how the simulation and test features can be applied on a UML model.

# 11

# *Verifying an Application*

The Model Verifier allows you to verify the behavior of your UML model and that the implementation is correct.

When you are building an instrumented application for the Model Verifier, you are performing similar instructions as you are when building an application with the C Code Generator. This section lists the basic build functionality and it covers the basic usage of the Model Verifier.

**See also**

Chapter 12, Model Verifier Reference.

Chapter 55, Debugging a C++ Application.

# Overview of the Model Verifier

The Model Verifier allows you to verify the behavior of your UML model and to verify that the implementation is correct. Using the Application Builder, you generate a C code executable program, that is an instrumented application, from your model and you link it with a predefined run-time library which is customized for simulation purposes. To simulate the model means that you run the executable program using various commands and breakpoints. You can run the simulation automatically or can manually step through transitions, send signals, etc.

You can control the Model Verifier from the user interface or you can control it using Model Verifier Console commands. During the simulation session you can view different aspects of your implementation. You can focus on the internal behavior of the model, or maybe you are just interested in verifying the signaling to and from the environment.

The execution of the simulation session can be traced graphically in state machine or activity diagrams, textually in the Output window or in sequence diagrams. The simulation session produces logs that can be saved as text files.

If your application communicates with the environment, this behavior can also be simulated. By sending messages, you simulate signals going in to your model from the environment.

During the simulation session, you have a number of views available that displays different aspects of the simulation. This allows you to monitor values, steps, etc.

You can also use the Model Verifier for simulating activities. See Activity Simulation for more details.

# Generating an Instrumented Application

There are several methods available for generating an instrumented application. Which to use is dependent on your needs.

**Important!**

*You must have a C/C++ compiler installed to generate an executable Model Verifier application.*

### Using Build Artifacts

This method allows you to specify the build settings by applying stereotypes and attributes in order to customize the build of the application, should the default build settings not be adequate.

To use a Build Artifact to build a Model Verifier:

- If required, create the build artifact. Right-click and choose **Model Verifier** from the shortcut menu, followed by the submenu **New Artifact**.
- Right click the build artifact for the Model Verifier of your choice. On the shortcut menu, select **Build (Model Verifier)**, and then the submenu **Build**.

**Hint**

*This build artifact can now be reused for the build and launch of future Model Verifier sessions.You do not have to create a new build artifact for each build.*

### Building a Selective Model Element

Since build artifacts are mandatory for a build to take place, it is not possible to order the build of a Model Verifier on a UML model element without first having to create a build artifact.

However, the tool provides the functionality to simplify this.

1. Right-click the desired Build Root in the model view
2. Select the menu choice **Model Verifier** followed by the menu choice **New Artifact**.
   - A build artifact with default settings suitable for the build of a Model Verifier, and with a default name **ArtifactNNNN** is created for you.
3. Now select the command **Model Verifier** again and the submenu **Launch**.

### Using Configurations for Build

This method allows you to build several build artifacts at the same time. However, at most one of those build artifacts should be a Model Verifier build artifact, since only one instance of the Model Verifier is allowed to execute at the same time.

**Note**

*To simulate an activity model you need to use a specific Add-in (ADSim) which creates a Build Artifact automatically for you from the activity to simulate. This build artifact can then be launched as usual, as described above. For more information on the topic of activity simulation see Activity Simulation.*

**See also**

Chapter 27, Building Applications Reference

# Running the Model Verifier

You can start the Model Verifier after a Model Verifier build is completed without errors. When the Model Verifier starts, the Instances tab opens in the Workspace window. In the Output window, the Model Verifier tab opens.

## Start the Model Verifier

You can either start a Model Verifier executable resulting from a previous build, or build and start a Model Verifier with one command.

### Start Model Verifier without building

You can use this command if you have already built a Model Verifier. Avoiding a rebuild speeds up the launch process, however no checks are performed to verify that the Model Verifier originates from the model currently loaded in the workspace. If that is not the case, or if the model has changed since it was built, then incorrect or incomplete information may likely be displayed by the Model Verifier.

1. On the **Build** menu, select **Start Model Verifier**.
2. In the dialog that is displayed, specify the name and location of a file that contains a Model Verifier executable (this file has to be created by a previous build).
   - The **Browse** button is a handy feature to browse through the file system using a standard Open dialog.

After the OK button is pressed, Tau attempts to launch the Model Verifier on the specified executable, after performing some non-exhaustive checks that the file seems to contain a Model Verifier executable.

### Launching after Using Build Artifacts

This method is useful if you already have created a build artifact for a Model Verifier.

1. In the Model View, right-click the build artifact manifesting the Model Verifier to launch.
2. On the shortcut menu, select **Build (Model Verifier)**, followed by the menu choice **Launch**.

A Model Verifier is built and, if the build is successful, the newly built Model Verifier is launched.

### Launching after Building a Selective Model Element

If you do not yet have any suitable build artifact for a Model Verifier, you can proceed as follows:

1. In the Model View, right-click the class to become the Build Root.
2. On the shortcut menu, select **Model Verifier** followed by the menu choice **New Artifact**.
   - A new build artifact named **ArtifactNNNN** is created
   - If desired, rename the newly created build artifact, and adjust the build settings (optional).
3. On the shortcut menu, select **Model Verifier** again, then select the name of the newly created artifact (named **Artifact0001** by default) and lastly **Launch.**

### Launching after Using Configurations for Build

This method is handy in case you want to build multiple build artifacts, of which one is a Model Verifier, and launch the newly built Model Verifier. One special application is when your configuration contains exactly one Model Verifier build artifact.

- In the project toolbar, click the button **Execute Configuration**. This orders a build of all build artifacts contained in the active configuration and launches the newly built Model Verifier

**Note**

*At most one Model Verifier can be launched at the same time. If multiple Model Verifier build artifacts are found, all are built but only one will be launched.*

## Exit the Model Verifier

To exit the Model Verifier, you can proceed with either of following:

- On the **Verify** menu, select **Stop Model Verifier**.
- On the **Build** menu, select **Stop**

## Instances

When you start the Model Verifier, the Instances tab opens. It displays:

- A tree of active class instances that are being executed. Inheritance relationships and instantiations are flattened and new nodes that are representing live active class instances with their instance number are shown.
- Run-time dependent objects: the ready queue, the timer queue, the system environment instance and the active timer list if that object exists.

Attributes and formal parameters are displayed as child nodes of active class instances nodes. Their values, however, are displayed in the Watch window.

Implicit attributes are created to represent:

- The control state of state machines, the state attribute
- Predefined identifiers of data type Pid: Sender, Parents, Offsprings
- The call stack of instances of active classes, the CallStack attribute
- The message queue of the instances of active classes, the Queue attribute.

### Ready queue

The ReadyQueue object is shown as a list of identifiers. The syntax of the identifiers is: `instance_name[instance_number]`

The following information is displayed on the same line as the identifier:

- The current state of the active class instance: **<state_name>**. If the instance is executing a transition, indicating that it is leaving its state, the state is instead displayed as: **(state_name)**

- The next signal to be consumed: `>`**signal_name**
- The length of the message queue: `]`**number**

**Timer queue**

The elements in the TimerQueue object are similar to those in a message with an additional field of the type Time.

# Tracing the Execution

When you are running your Model Verifier, you can easily obtain trace information of the execution. This allows you to track each transition and event in your application. You can select between three different tracing methods.

- Textual tracing
- UML model tracking
- Sequence diagram tracing.

You can enable any combination of the three methods above.

## Textual trace

The textual trace displays each executed step in the simulation in the Model Verifier tab in the Output window. You can select the extent of the displayed information by setting the trace level of the output. Trace levels from 0 to 6 are available. The trace level 1 is set by default. Level 0 indicates that the textual trace is disabled.

You can also decide which unit that the trace levels should be applied to. A unit is an active class instance. If you do not specify any unit, the trace settings apply to all units.

The trace level can only be changed using the Model Verifier Console input.

- To change the trace level, type `set-trace <optional unit name> <trace-level>`.
  - If you type '?' as the unit name, a list with all available units will be presented,
- To disable the textual trace, type `set-trace 0`.

**See also**

"Textual trace levels" on page 496

"Set-Trace" on page 529

## Custom textual trace

It is possible to trace custom text messages to the Model Verifier console. This can be used to print application specific debug information, and could be a useful complement to the built-in trace information.

In order to trace a text message in the console window the utility function `xPrintString()` can be used. The function is defined in the libraries for the Model Verifier, in the file `scttypes.h`

**Example 112: Using xPrintString()**

```
#include <scttypes.h> /* For the definition */
...
xPrintString("Bugs Bunny\n");
...
```

### Formatting of strings prior to trace

`xPrintString()` expects a preformatted string to trace. If you need to format the string you can either do this before calling `xPrintString()` (see Example 113 on page 450), or you can use the function `xWriteBuf_Fmt()`. This function accepts "printf"-style arguments.

**Example 113: Formatting of strings**

Achieving the results expressed using the C statement:

```
printf("x=%d\n", 4);
```

Should be achieved in two steps, as follows:

```
char str[20]; /* array long enough to hold the result */
sprintf(str, "x=%d\n", 4);
xPrintString(str);
```

Or, alternatively:

```
xWriteBuf_Fmt("x=%d\n", 4);
```

### Textual trace for applications

You may also combine the textual trace used when running the Model Verifier with the ability to print trace messages while running the target application. To manage the trace code and distinguish its use between Model Verifier and target application sessions, you could combine both uses by conditional compilation looking at a significant C macro (such as XTRACE), as exemplified below:

**Example 114: Combining Model Verifier and application traces**

```
#ifdef XTRACE
  /* XTRACE is defined for Model Verifier */
  #define myprintf(S) xPrintString(S)
#else
  /* If not Model Verifier, then assume application*/
  #define myprintf(S) printf(S)
#endif
```

## UML model tracking

This tracking method allows you to follow execution in the UML diagrams that were used for defining the simulated model. For example, it is possible to follow execution of state machine transitions and statements in state chart, class and text diagrams of your UML model. Using the ADSim add-in, execution can also be tracked in activity diagrams (see Activity Simulation for more information).

When the tracking starts, the diagram with the selected statement opens. When the execution continues, the next statement in the diagram is highlighted and so on.

**Note**

*UML model tracking is enabled by default.*

### Enabling UML model tracking

To enable UML model tracking, perform one of the following tasks:

- On the **Verify** menu, click **Show next statement**.
- Click the **Show next statement** button on the Model Verifier toolbar.

UML model tracking is enabled when the button is pressed in.

### Disabling UML model tracking

To disable UML model tracking, execute the command **Show next statement** again.

### State machine vs. Activity tracking mode

By default the Model Verifier executes in **state machine mode**. In this mode the focus is on displaying the current point of execution, that is the statement that is about to be executed next. A green triangle is inserted beside the symbol or the statement within a symbol that is about to be executed.

When simulating activity models, which use an execution model based on token flows, it is more appropriate to use the **activity mode**. Then the focus is to track the execution of activity nodes, as a consequence of token flows. Activity node execution is tracked textually, and graphically by selecting the executing activity node symbol.

When starting the Model Verifier it will automatically select the appropriate tracking mode for your model. If needed, you can switch between state machine and activity tracking mode by using the commands `statemachine-mode` and `activity-mode` in the Model Verifier Console input.

**Note**

*If the* `activity-mode` *command is not used when simulating an activity model, tracking will be done in state machines generated by the ADSim add-in. This can be confusing, and is generally not recommended. It is not possible to track the execution to both state machines and activities simultaneously.*

**See also**

"Change sequence trace level and UML model tracking level" on page 479

"Execution tracking levels" on page 497

## Sequence diagram tracing

This tracing method allows you to follow each transition in a sequence diagram, and to observe signal sending that takes place in the transitions.

### Enabling sequence diagram tracing

To enable tracking in sequence diagrams, perform one of the following tasks:

- On the **Verify** menu, click **Tracing in Sequence Diagram**.
- Click the **Tracing in Sequence Diagram** button on the Model Verifier toolbar.

Sequence diagrams tracing is enabled when the Tracing in sequence diagram button is pressed in.

The sequence diagram trace window can be opened as a normal window, or as a docked window. This behavior can be controlled by the option **Dock sequence diagram trace window.**

**See also**

"Sequence diagrams" on page 2464

### Disabling sequence diagram tracing

- To disable sequence diagram tracing, execute the command **Tracing in Sequence Diagram** once more.

### Navigating to the UML source

If you double-click a symbol in the sequence diagram, you will navigate to the position where the symbol originated.

### Interrupting the trace

The tracing is interrupted if the model is changed during the tracing, for instance if you move the newly created sequence diagram from its default location. However, you can open a new sequence diagram and continue the tracing in that diagram.

### Changing sequence diagram trace levels

The granularity of the information presented when tracing in sequence diagrams can be changed to the user's convenience.

**Hint**

*When using both UML model tracking and sequence diagram tracing at the same time, it is convenient to dock the sequence diagram window at one side of the Tau IDE. To do this, right-click on the title bar of the sequence diagram window and select "Docked to...".*

**Hint**

*For any realistically sized application it is typical that the sequence diagram that results from a trace will contain a large number of lifelines. A sequence diagram trace of an activity simulation session typically also leads to a large number of lifelines. In this situation it is often useful to zoom out the diagram to get an overview. Use for example "Zoom -> Zoom to fit" in the diagram context menu. In this overview mode texts on symbols and lines are typically too small to read. But by resting the mouse over a symbol or line in the diagram its text will be displayed in a tool-tip.*

**See also**

"Change sequence trace level and UML model tracking level" on page 479

"Sequence diagram trace levels" on page 497

# Executing the Application

There are several different commands available to start the execution. Select the command that suits your needs.

## Start the execution

The execution can be started using one of the following methods:

- On the **Verify** menu, click the command you want to use.
- Select the command from the Model Verifier toolbar.
- Type the command in the Model Verifier Console.

**See also**

"Syntax of commands" on page 507

"Model Verifier Console" on page 478

## Stop the execution

When you have started the executed with, for instance, the Go command, you can only stop the execution by using one of the following methods:

- On the **Verify** menu, click **Break**.
- Select **Break** from the Model Verifier toolbar
- Type `break` in the Model Verifier Console

**Note**

*The execution also stops at each active breakpoint, when the application becomes idle or when a dynamic error has occurred.*

## Re-start the execution

You can run the same simulation over and over again by using one of the following methods:

- On the **Verify** menu, click **Restart**.
- Select **Restart** from the Model Verifier toolbar

The simulation always restarts from the beginning, that is simulation time zero. If you want to run the simulation from a specific position, it must be run in replaying mode.

**See also**

"Replaying Mode" on page 468

## Run-Time prompting

Some constructs like informal decisions, ANY decisions and non-implemented operators require user input to continue the execution.

### Decision prompting

When an informal decision or an ANY decision is reached in the execution, a dialog opens displaying the possible choices. Select your choice, click OK and the execution resumes.

### Operation prompting

When an operation of a passive or active class is declared but has no implementation, its execution opens a dialog. This dialog contains a tree structure similar to a Watch window.

The tree structure shows the stack frame of the operator call, that is a root node with the name of the operation, with child nodes for each argument, and a child node called **result**. If the operation belongs to a passive class and is not static, there is also a node named **itself** which is the object on which the operation is applied.

- You can modify any element of the tree. Each assignment is registered in the current scenario.
- You can also use other commands to modify the state of the model, for instance if the operation belongs to an active class it may want to send a signal
- Click the **OK** button to continue execution.

## Insert and remove breakpoints

Breakpoints can be set to allow you to stop the execution at positions that are of interest. Breakpoints can be set in the model at any time, it is not necessary to have a started Model Verifier application.

### Insert breakpoints

To insert a breakpoint:

1. Open the diagram that contains the symbol where you want to insert the breakpoint.
2. Right-click the symbol, or click in its text, and select **Insert/Remove Breakpoint** from the shortcut menu. A red dot is added to the symbol frame or next to a statement within a symbol frame to indicate that the breakpoint has been inserted.

You can also insert a breakpoint via commands in the Verify menu from the Model Verifier toolbar.

Inserted breakpoints are listed in the Breakpoint window.

### Precise positioning of breakpoints

Below are some examples of how to set breakpoints to achieve the exact position you desire.

- To set a breakpoint to a specific statement, when an action symbol contains several statements, position the cursor before the ending semicolon of the statement.This also applies to setting breakpoints in text diagrams.
- To insert a breakpoint on a `for` statement, position the cursor anywhere in the keyword `for`.
- To insert a breakpoint on a `while-do` statement, position the cursor anywhere in the keyword `do`.
- To insert a breakpoint on a nextstate action, insert it on the flow line going to the state symbol.

### Remove breakpoints

To remove a breakpoint:

1. Open the state machine diagram that contains the symbol where the breakpoint is inserted.
2. Right-click the symbol and click **Insert/Remove Breakpoint** from the shortcut menu.

You can also remove breakpoints from the Breakpoints window.

### List breakpoints

To list breakpoints:

All inserted breakpoints can be listed in the Breakpoint window. This allows you to locate any existing breakpoint in your model.

- On the **Edit** menu, click **Breakpoints**

You can enable and disable breakpoints by selecting or clearing the check boxes in the Breakpoints window. Disabling a breakpoint is not the same as removing the breakpoint. Removed breakpoints are not listed in the Breakpoints window.

If you double-click the breakpoint entry, the diagram where the breakpoint is set opens.

**See also**

"Model Verifier Console" on page 478

## Send messages

To simulate the behavior of the environment, you can manually send messages from the environment to your model. The messages you want to send must be inserted in a message list. A message is a signal with its parameters including sender and receiver paths.

You can also add internal messages to the list. An internal message is a message that has both its sender and receiver within your model. This allows you to verify incomplete systems. You can manually send messages that are not yet implemented. This feature also allows you to test the robustness of your model. You can test, for instance, that the model can handle unexpected signals.

You can send messages any time during the simulation session.

### Create a message

To create a message:

1. On the **View** menu, point to **Model Verifier Windows** and click **Messages**. The Message list opens in the Output window.
2. Right-click the first row in the list and click **Insert** from the shortcut menu that appears.
3. Click each column to display a list of available signals, senders etc. Make your selections.

Elements of the lists in the separate boxes are given with their complete path, so that they can be precisely identified. Examples of paths are "a_block.a_process[an_integer]" or "a_package::a_signal".

The syntax of the Parameters field is the UML syntax for a list of expressions. Two predefined values are provided, a list of the parameters sorts and [[-]] sign. When you select [[-]], the Model Verifier selects a "default NULL" value for you. For example, default NULL value for integer is 0, and for a character string it is an empty string (" ").

You can type text instead of choosing an element in the box. In this case you can specify an incomplete path. This means that you can omit the beginning of the path. However, you must not omit the end, not even the instance number.

### Send a signal

To send a signal:

1. In the Message list, right-click the signal you want to send.
2. Click **Send** to send the signal.

### Complex signal parameter values

Sometimes you may have very complex values that need to be inserted in the Parameters field. The following instructions will help you to define the parameter value:

1. Choose the [[-]] expression in the Parameters field.
2. Send the message.
3. Open a Watch window on the Queue of destination instances.
4. Open the Watch tree to observe the message that you have sent.
5. Press F2 and set the parameter values.
6. Select the tree nodes which represent the parameters.
7. Apply the Deep Copy command.
8. Paste the result in the Parameters field.

Complex parameters (classes, strings etc.) that are part declared will be referred by value, parameters that are not part declared will be referenced by a pointer. To assign a new value to a parameter that are not part declared it must be enclosed in curly brackets ({}). Classes must be type casted to be accepted. Elements of passive classes must be formally assigned to the element name.

**Example 115: Signal parameters**

```
class Class2 {
    Integer p1;
    Real p2;
}
interface i {
    signal mySig1( myString par1);
```

```
}
syntype myString = String<Class2>;
```

To send a signal `mySig1` here is an example of how to enter a value of type `myString` in the parameter field of the Messages window.

```
{Class2 (. p1=8, p2=5.1 .), Class2 (. p1=4, p2=5.6 .)}
```

This example represents a string with two elements of type Class2.

---

**See also**

"Passive class values" on page 504 in Chapter 12, *Model Verifier Reference* for information on how values can be entered in other situations and using the syntax of other supported languages like SDL and ASN.1.

## Watch window

During the simulation session, you can watch in a separate window any of the model objects that are available in the Instances view. You can watch active class instances, active class instance sets, attributes of active class instances, attribute elements and run-time objects such as message queues, timer instances and call stacks.

The values are displayed in a Watch window. You can watch as many objects as you want and you can open multiple watch windows.

The objects are stored in the Watch window even if you close the view window.

### Watching instance objects

To watch instance objects:

1. Right-click the object in the Instances view.
2. Click **Watch**.

A Watch window opens displaying the object and its values. If a watch window is already opened, the object is added to the latest selected watch.

You can also drag and drop an object from the Instances view if a Watch window is already opened. Any object in the Instances view can be watched.

### Removing instance objects from the Watch window

To remove instance objects:

- In the Watch window, right-click the object you want to remove and click **Unwatch**. This removes the selected tree root from the view.

### Opening an empty Watch window

To open a new, empty Watch window:

- From the **Verify** menu, click **Open new watch**.

## View and edit via the Console window

For viewing/editing complex types such as an array or a structured data type during a debugging session the Model Verifier Console window can be an effective tool. With some console commands you can print/edit the whole attribute, a range of elements or one element.

**Example 116: Print an array element**

Consider an attribute named `myData`, which is a pointer to a struct, and contains an array as one of its elements.

```
Examine-Variable (MyClass:1) myData -> bits  22
```

This will print the 22nd element in the array.

---

**Example 117: Print a range of array elements**

Consider an attribute named `myData`, which is a pointer to a struct, and contains an array as one of its elements.

```
Examine-Variable (MyClass:1) myData -> bits  22 25
```

This will print element 22 to 25 in the array.

---

**Example 118: Initialize an element**

Consider an attribute named `myData`, which is a pointer to a struct, and contains an array as one of its elements.

```
Assign-Value (MyClass:1) myData -> bits 22 1
```

This will set the 22nd element to 1.

---

**Example 119: Print an array element**

Consider an attribute named `myData`, which is a pointer to a struct, and contains an array as one of its elements.

```
Examine-Variable (MyClass:1) myData
```

This will print the whole array.

---

### Print address and value of a pointer

This command will set the Model Verifier mode to print only the address of pointers. This is the default mode.

```
REF-Address-Notation
```

The following command will change the Model Verifier mode to follow the pointer and print the value of the element referred to:

```
REF-Value-Notation
```

## Change element values

During the simulation session, you can change the values of the elements manually. This feature allows you to quickly test different values for an element.

To change the element value:

1. In the **Watch window**, click the element and press F2.
2. Type the new value in the text field that appears. Press ENTER.

If you type an illegal value, the element will preserve its previous value.

## Display element values

In some situations, for example at a certain point in time during the simulation session, it is of interest to know the values of attributes, signals, states, etc. The values are displayed in the Model Verifier tab in the Output window. This means that the values are saved if you want to log the debug results.

1. In the **Instances view**, select the elements you want to display.
2. Right-click the elements you selected and click **Display**. The values are displayed in the Output window.

**See also**

"Log the result" on page 465

## Copy and paste element values

It is possible to copy and paste the values of the elements in the Instance view or in the Watch view. The commands that are available are Copy, Deep copy and Paste.

**Note**

*No element altering commands are provided (such as Cut), except for predefined text operations in edit mode.*

### Copy

The Copy command can be applied on objects that are not active, nor operation, nor messages. The command makes a copy of a text representing the value of the object as a UML expression. When several objects are copied at the same time, their values are separated by commas.

The copied text can then be pasted to assign a given value to another element during the session.

When an object contains references to other objects, it is the physical address of the other object that is copied. When a physical address is copied, it is not recommended to paste it in your model since physical addresses are not valid in later sessions.

**To copy using the Copy command:**

1. Click the object in the **Instances** View or in the **Watch** window.
2. From the **Edit** menu, click **Copy**.
3. Select the object you want to assign the value.
4. Press **F2** to enter edit mode.
5. Right-click the field and click **Paste** from the shortcut menu.

**Deep copy**

This command is similar to the Copy command. The only difference is that when the object contains a reference to another object, it is the value of the other object which is copied rather than the physical address.

The resulting text can then be pasted, for instance, to assign a given value to another object, or in the parameters field in the Messages dialog, or in your model to define named constants, or to define values of signal parameters in Sequence diagram, etc.

The Paste command can be applied when the copied object is a UML static constant attribute: the UML expression which defines the constant is assigned to the object.

**To copy using the Deep Copy command:**

1. Right-click the object in the **Instances** View or in the **Watch** window.
2. From the shortcut menu that opens, click **Deep Copy**.
3. Select the object you want to assign the value.
4. Press **F2** to enter edit mode.
5. Right-click the field and click **Paste** from the shortcut menu.

**See also**

"UML Expressions" on page 474

## Create or delete instances

To be able to create and delete instances enhances the possibilities to verify your model. You can create and remove instances in either the Instances window or in the Watch window.

**To create a new instance**

1. Right-click the active class instance you want to make a new instance of.
2. Click **New**.

**To delete an instance**

1. Right-click the object you want to delete.
2. Click **Delete Instance**. This command:
   - Stops the selected object if it is an instance of active class
   - Resets the selected object if it is a timer instance
   - Deletes the selected object if it is under a pointer object
   - Removes the selected object if it is a messages of a message queue
   - Removes the selected object if it is a passive list object.

## Locate objects

The definition of the objects in the Instances view and in the Watch window, can easily be located by double-clicking the object. The definition is displayed in the diagram type that best describes the object.

## Log the result

The result of the execution can be logged. You can log the textual trace and the sequence diagram tracking.

### Log the textual trace

The textual trace is displayed in the Model Verifier tab in the Output window. To save the result in a text file, perform the following tasks:

1. In the Output window, click the **Model Verifier** tab.
2. Right-click anywhere in the text area, and click **Select all** on the shortcut menu that opens.
3. Right-click once more in the text area, and click **Save as** on the shortcut menu.
4. Decide a name for the log file and save the file.

### Log the sequence diagram trace

To save the result, perform the following tasks:

1. Make sure that **Tracing in Sequence Diagram** is enabled.
2. Start the execution.
3. Right-click the package DebugTrace that is available in the Model View and click Save in New File.

To save the result in a text file, perform the following tasks:

1. In the Model Verifier Console, type `start-batch 2 <filename.txt>`
2. Start the execution.
3. When you are done, type `stop-msc-log`.

The file is saved in the same folder where your model is saved.

## Static coverage views

The static coverage views are displayed as separate tabs in the Output window. The available tabs are Coverage Statistics, Code Coverage and Transition Coverage.

You can sort the lines in the report tabs according to any column by clicking on the column header. To sort according to a "primary" column and a "secondary" column, you sort first by the secondary column, then by the primary one.

### To view coverage statistics

Follow the instructions below to view the coverage statistics:

1. From the **Verify** menu, click **Show Coverage Statistics**. The Coverage Statistics tab opens in the Output window.
2. In Coverage Statistics tab, right-click a line and from the shortcut menu that opens, click **Show Coverage Details**. Depending on what kind of line it is, the Code Coverage tab or the Transition Coverage tab opens.

   An alternative way to open the coverage details tabs is to double-click the line in the Coverage Statistics tab.
3. Select one of the coverage details tabs and right-click a line. From the shortcut menu, click **Locate**. The corresponding source will now be displayed in a state machine diagram.

The coverage information is updated each time you issue this command. Furthermore, the “Code Coverage” and “Transition Coverage” views are updated if they are defined.

### Coverage Statistics tab

The Coverage Statistics tab lists the operations of the model that is being executed. The columns of the “Coverage Statistics” report are:

- **Operation**: This is the name of the operation
- **Path**: This is the full path of the operation
- **Kind**: The available values are:
  - **statements** if the line describes the coverage of statements
  - **transitions** if the line describes the coverage of transitions
- **Number**: This is the number of statements or transitions of the operation
- **Covered**: This is the number of statements or transitions that are covered, that is executed at least once during the session.
- **% Covered**: This list how much of the statements or transitions that has been executed.
- **Maximum queue length**: For operations which are constructors of active classes, this columns lists the maximum length of the queue of the instances of the active class, which was reached during the sessions.

### Code Coverage tab

The Code Coverage tab lists detailed information about the statements of the operations. The columns of the Code Coverage report are:

- **Operation**: This is the name of the operation.
- **Path**: This is the full path of the operation.
- **Statement**: This columns lists the statement that is executed.
- **Coverage**: This lists the number of time the statement was executed during the session.

### Transition Coverage tab

The Transition Coverage tab lists detailed information about the transitions of the operations. The columns of the Transition Coverage report are:

- **Operation**: This is the name of the operation.
- **Path**: This is the full path of the operation.
- **State**: This is the state from which the transition starts.
- **Signal**: This is the path of the signal which triggers the transition.
- **Coverage**: This lists the number of time the transition was executed during the session.

# Replaying Mode

The replay mode allows you to record all performed executions steps and user commands from the initial state of the application. The execution steps and user commands are saved in a scenario. The scenario can be saved and be replayed to any given state within the application.

You can for instance record the scenario which passes through a complex initialization phase of your application, and replay it whenever you need. Or you can record a scenario if the debug session must be interrupted and continued later.

Execution steps that are recorded in a scenario are:

- Transitions
- Time-out of timers.

User commands recorded in a scenario are the commands which modify the state of the application. They are:

- Signal Sending
- New
- Delete
- Re-arrange
- Assignment.

## Open a scenario

When you open a scenario, that scenario becomes the current scenario. The previous contents in the current scenario will be deleted.

The scenario can be viewed in the Scenario window.

### Load a scenario file

To load a scenario file:

1. On the **File** menu, click **Open Scenario**.
2. Select the scenario file you want to use and click **Open**.

### Backward compatibility

If you open a scenario file that was saved in a previous version of Tau, signal sending and assignment steps are transformed to ensure backward compatibility.

- For signal sending steps, parameters of messages are marked with an SDL prefix. This means that the parameters will not be handled as UML expressions.
- For assignment steps, the assigned value is transformed as a UML information expression with an SDL prefix. This means that the values will not be handled as UML expressions.

**Important!**

*The names of implicit instances can change between releases. This can thus require you to edit or regenerate scenarios that have been generated in an earlier version.*
*A Receiver of a message can for example change from:*
*mm_om.AAA.@part_@implicit_process[1]*
*to:*
*mm_om.AAA.@part_@implicit_process_0[1]*

**Note**

*A scenario generated in a previous version of Tau may be invalid due to the fact that the scenario file is not converted to reflect internal model changes. Such changes in the model files are automatically converted when you open a* `.u2` *file in a newer version of Tau.*

## Save a scenario

Each time you execute a simulation, the executions steps are listed in the current scenario. This scenario is overwritten each time you start a new execution.

However, you can save the current scenario into a file, to replay it later. Scenario files have a suffix `.ttdscn` and the suggested default name of the current scenario is `default.ttdscn`.

**Save a scenario**

To save a scenario:

1. Execute the steps you want to include in the scenario. The performed steps are listed in the scenario window.
2. On the **File** menu, click **Save Scenario**.

## View the contents of a scenario

Each execution step is listed in the scenario window. The scenario is a list of text elements describing each step. This list is not editable from the Scenario window.

When you are running the scenario, you can follow each execution step. Executed steps are marked in the respective check boxes.

**To open the Scenario window**

- On the **View** menu, point to **Model Verifier Windows** and click **Scenario**.

**See also**

"Execution steps" on page 534

"User commands" on page 534

## Execute a scenario

The scenario cannot be executed if the steps do not match the application. In this case you will be notified in the Model Verifier tab of the Output window.

The scenario always starts from the first execution step. When executed, each step is checked.

**To execute a scenario:**

1. On the **Verify** menu, click **Replay Mode**, if it is not already enabled.
2. Select the command you want to run:
   - Use the **Go** command to execute all steps in the current scenario. The execution is only interrupted by breakpoints.
   - Use the **Next Transition** command to execute the next step of the current scenario. It can be a transition, a time-out, or a user command. This command is disabled if the current position in the current scenario is at the end.
   - Use the **Step Into** command to step into the transition which is described by the next scenario step.
   - The **Step Into**, **Step Over** and **Step Out** commands behave as usual when a transition is already under execution, otherwise they are disabled.

**Note**

*The behavior of the commands above are only valid if the Model Verifier is set in replay mode.*

# Model Verifier Configuration

A Model Verifier configuration lists properties and actions that you have performed during the execution. The information in the configuration includes:

- Messages that you have inserted in the Messages window.
- Breakpoints that you have inserted and that are listed in the Breakpoints window. Break conditions that are set through Model Verifier Console commands Breakpoint-* are not included.
- Sequence diagram trace levels that you have set to another value than “according to parent”.
- Execution tracking levels that you have set to another value than “according to parent”.
- Instances that you have added to each Watch window.

**Note**

*The graphical layout of Watch windows is not saved.*

Each time you execute a model, a current configuration is created. The current configuration can be saved and thus be reused later. This means that each time you execute a model, you do not have to insert your breakpoints, messages etc.

You can save the whole configuration or parts of it. When you stop the Model Verifier, the current configuration is saved in the Target Directory and named after the Model Verifier executable. The next time the Model Verifier is launched, the file is loaded at start-up.

**See also**

"Save Model Verifier configurations" on page 472

"Replaying Mode" on page 468

## Save Model Verifier configurations

Each time that you stop the Model Verifier, the current Model Verifier configuration is automatically saved in a file with the extension `.ttdcfg`. The name of the configuration file is derived from the name Model Verifier executable and it is saved by default in the Target Directory.

All information in the current configuration is saved in the default configuration file. However, if you want to save a subset of the configuration information, or if you want to save the configuration without stopping the Model Verifier, you can manually save the configuration file.

**Note**

*Only trace and tracking levels that differ from "according to parent" are saved.*

**To manually save a Model Verifier configuration:**

1. Execute the steps you want to include in the configuration.
2. On the **File** menu, click **Save Model Verifier Configuration**.
3. In the dialog that opens, select what type of information that you want to save in the configuration.

   The **All** check box is selected by default. To save a subset of the configuration options, clear the All check box and select the desired check boxes.

4. In the **File name** field, name the configuration file and select where to save it. The default name of the file is `default.ttdcfg` and the default position is in the Target Directory.

## Load Model Verifier configurations

When you start the Model Verifier, the configuration file is automatically loaded. This file contains the settings from the last time the project was used to run an execution.

However, you can also load a manually saved configuration file. There are two methods available to do this. You can either:

- Open a configuration.
- Include a configuration.

If you want to automatically load a manually saved configuration file the next time you start the Model Verifier, you must replace the generated configuration file with your saved file after you have stopped the Model Verifier.

### Open a Model Verifier configuration

To open a configuration means that you overwrite the settings in the current configuration with the properties that are saved in the configuration file that you want to open.

Only the objects that are saved in the configuration will overwrite objects in the current configuration. For instance, if the configuration file is saved with the Messages check box cleared, then the messages in the Messages window of the current configuration will not be overwritten.

1. From the **File** menu, click **Open Model Verifier Configuration**.
2. In the dialog that opens, select the `.ttdcfg` file you want to open and click **Open**.

**Note**

*If you try to open a configuration generated in a previous version of Tau that does not reflect changes that you may have made to the model, elements that are no longer valid will be ignored.*

**Note**

*A configuration generated in a previous version of Tau may be invalid due to the fact that a configuration file is not converted to reflect internal model changes. Such changes in the model files are automatically converted when you open a* `.u2` *file in a newer version of Tau.*

### Include Model Verifier configuration

To include a configuration means that the objects in the configuration file merge with the objects in the current configuration. For Breakpoints and Messages, merging means that duplicated objects are ignored. For trace and tracking levels, merging means that the new levels are applied after the current ones.

For instance if you have message A in the saved configuration and message B in the current configuration, both messages will be available after include. Compare this with the Open command where signal B would be overwritten.

1. From the **File** menu, click **Include Model Verifier Configuration**.
2. In the dialog that opens, select the `.ttdcfg` file you want to include and click **Open**.

## Console commands

Model Verifier configurations can be saved and loaded in the Model Verifier Console mode using the commands !U2::Debug save and !U2::Debug open.

# UML Expressions

The UML syntax of expressions is used to assign or to represent the values of objects in the Model Verifier. To achieve this, the Model Verifier includes conversion algorithms between the model of UML expression and the model of Model Verifier objects.

For the cases where no ordinary expression can represent a value, informal expressions (also called target expressions) are used. The contents of the informal expressions are specific to the Model Verifier, and cannot be handled by the rest of the tool.

A subset of UML constants expressions are supported by the Model Verifier as input.

**Note**

*In the following sections, expressions are described by their name given in the UML textual syntax definition, for instance < integer name>, not by their name in the Metamodel, which would have been IntegerValue instead of < integer name>.*

## Mapping of values to expressions

The kind of expression that is used to represent the value of an object depends on the type of the object:

| Object | Expression |
|---|---|
| Integer-like objects and objects of type Null | <literal> of the kind <integer name> |
| Real-like objects | <literal> of the kind <real name> |
| Time and Duration-like objects | <literal> of the kind <real name>, without exponent |
| Boolean-like objects | ‘true’ or ‘false’ identifiers |
| Charstring-like objects | <literal> of the kind <character string> (with double-quotes as delimiter, and use of ‘\’ to include double-quotes in the string |
| `Octet`-like and `OctetString`-like objects | <literal> of the kind <hex string> |
| BitString-like objects | <literal> of the kind <bit string> |
| Bit-like objects | ‘0’ or ‘1’ |
| Character-like objects | • <literal> of the kind <character><br>• If the previous notation is not possible, <target expression> containing a hexadecimal number, for instance ‘[[0x7f]]’ |

| Object | Expression |
|---|---|
| Pid-like objects | • The 'NULL' literal<br>• For non-NULL Pid values: <target expression> containing the full path of the instance which is referenced, for instance `[[Match.p2[1] ]]`<br>**Note the space after the '[1]'** |
| Enumerated objects | <identifier> referring to the literal name |
| Passive class objects | <structure primary>, where the names of attributes are mandatory, and all non-optional attributes must be specified, even private attributes. |
| String-like and Array-like objects (including multiple attributes) | <list expression> |
| Choice objects | < structure primary>, where only the active attribute is specified. |
| References to other objects ("pointers") | • The 'NULL' literal<br>• When the Deep Copy command is used, or in Sequence diagram traces: the value of the referenced object is directly used<br>• Otherwise: a <target expression> containing a hexadecimal number representing the physical address, for instance [[0x12efbc]] |
| "General Array" objects | The value cannot be mapped to an expression. |

When pointers are processed by a Deep Copy command or in a Sequence diagram trace, the recursive processing of referenced objects can result in a graph of objects. To represent this as expressions, the objects which are referenced more than once are given a kind of label the first time they are referenced and their value is represented. The next time they are referenced, only the label is given.

The label is represented as an informal expression. Its value is made of an underscore character and an integer, for instance [[_1]].

The label is associated to the object value by using an assignment expression, for instance "`( [[_1]] = MyClass (. ...attribute values... .)`".

Such labels are valid only within an expression, or without the list of expressions of a message definition.

**Example 120: Expression using object labels**

Given a class:

```
class MyClass {
  MyClass previous;
  Integer value;
  MyClass next;
}
```

an expression can be

```
( [[_1]] = MyClass (. previous NULL, value 1, next
MyClass (. previous [[_1]], value 2, next NULL .) .) )
```

**Note**

*The Deep Copy processing is not suitable for all possible data structures, since the tool cannot guess the semantics of your model. But it is suitable for frequent patterns like linked lists or trees, provided that you do not refer to a sub-object having "owner" or "parent"-like links.*

**Important!**

*The Deep Copy algorithm is not able to detect that a pointer refers to a part of a larger object. The algorithm always considers that a pointer refers to an object on its own.*

## Mapping of expressions to values

When the Model Verifier uses a UML expression to build or assign the value of an object, it supports:

- Expressions that the Model Verifier would build to represent the value, as described in "Mapping of values to expressions" on page 475.
- Parenthesis expressions: '('<expression> ')'
- In the case of expressions copied from the UML model: <identifier> that refers to static constant attributes. In case the expression that specifies the value of the constant is used, the address of the constant is never used.

- Informal expressions containing only a dash sign '[[-]]', which tells the tool to create some value according to the type of the object.
- When you describe a graph of objects by associating labels to objects (as described in "Mapping of values to expressions" on page 475), the labels must have an underscore as first character, but they can have any string after it, so you can for instance use "`[[_FirstElementOfTypeMyClass]]`" instead of "`[[_1]]`".

**Note**

*Operation calls are not supported in expressions, nor references to attribute objects.*

# Error Handling

Violations of the dynamic rules of UML causes dynamic errors during the execution of a simulation. Dynamic errors are displayed in the Output window.

After a dynamic error has been detected, the execution of the simulation is resumed until the current statement is ended.

# Model Verifier Console

It is also possible to run the Model Verifier using textual command input. There are two types of commands that you can use. The commands that start with '!' have a counterpart in the user interface.

Follow the instructions below to start the console window:

1. On the **View** menu, point to **Model Verifier Windows** and click **Console**. The console opens.
2. Type the command you want to use and the required commands parameters.
3. Press **ENTER**.

The output of the commands is presented in the Model Verifier tab in the Output window.

**Hint**

*Commands that recently have been issued are saved and can be re-issued by clicking the arrow to the right of the console field.*

**Note**

*You cannot undo any commands.*

Some of the commands require that you enter parameter values. If you do not type the required parameters, you will be prompted to do so:

- Type '?' to get a list of possible parameters.
- Type '-' to accept default values for the parameters. If no default value exists, a list of possible values is displayed.

**Note**

*The '?' and '-' commands do not apply to commands that start with '!'*

**See also**

"User commands" on page 534

"Console commands" on page 510

# Trace and Tracking levels

**Change sequence trace level and UML model tracking level**

In the **Instances view**, right-click the instance you want to change and click **Trace and Tracking Level** from the shortcut menu.

To change Sequence diagram trace levels:

- Click the **Trace level** you want and close the dialog.

To change Execution tracking levels:

- Click the **Tracking level** you want and close the dialog.

# Activity Simulation

You can use the Model Verifier to simulate activity models. The execution semantics of UML activities is based on tokens flowing into and out from activity nodes. See Activity Modeling for more detailed information on the semantics of activities.

The capability to simulate activity models is provided by an add-in called **ADSim**.

## Activating the ADSim Add-In

To be able to simulate activity models you must first activate the `ADSim` add-in. Perform the following steps:

1. In the **Tools** menu, select **Customize...**.
2. Click the Add-Ins tab and check the `ADSim` add-in.
3. Click **OK**.

## Starting the Activity Simulation

To simulate an activity, perform the following steps:

1. In the Model View select the activity you want to simulate.
2. Do one of the following:
   - Right-click the activity and select **Create simulation model**, or
   - In the main menu, select **ADSim > Create simulation model**, or
   - Right-click the activity and select **Activity Simulator > New Artifact**. A build artifact for activity simulation will be created. Right-click this artifact and select **Build (Activity Simulator) > Create simulation model**.

ADSim performs an analysis of the activity and translates it into an active class that implements the activity's behavior. ADSim also creates a wrapper active class that initiates the activity, as well as a Model Verifier build artifact manifesting the wrapper active class. To avoid modifying the original activity model, the wrapper active class and the Model Verifier build artifact are placed in a new top-level package that is stored in its own .u2 file.

When the analysis completes, you can use the Model Verifier to build and launch an executable from the generated build artifact. For more information, see Start the Model Verifier. By default, the Model Verifier runs in Activity-Mode when started on a build artifact generated by ADSim. See State machine vs. Activity tracking mode for more information about the activity vs. the statemachine tracking mode.

**Hint**

*The ADSim menu also contains the **Simulate** command. This command creates the simulation model and automatically launches the generated build artifact.*

## Commands to Step Through the Activity Model

You can simulate the execution of the activity as usual with the Model Verifier, however, the execution commands are slightly different in context. The two main commands that are intended to be used for activity simulation are:

- **Next-Transition**: This command executes one node in an activity.
- **Go**: This command runs the simulation until one of the following events occurs:
  - input from the environment, or
  - a decision by the user.

If a breakpoint is hit, execution stops at the activity node where the breakpoint is set.

## Textual Trace

During the activity simulation session, textual messages are traced to the Model Verifier console. These messages describe on an action/token level what is occurring in the simulation model. For example, you get a message when an activity node executes, or when tokens are sent in the model.

The textual trace is always available.

## Activity Diagram Trace

When the activity diagram trace is activated, the activity node symbol is selected in the activity diagram. Note that the behavior is different compared to the similar functionality in state charts. For activity diagrams it is the most recently executed activity node that is selected in the diagram. The green execution marker triangle also appears on the executed activity node.

The activity diagram trace is activated or deactivated by the **Show Next Statement** option available in the **Verify** menu. It is enabled by default.

## Trace Colorization

As activity nodes execute, their symbols change color in the activity diagram, allowing you to see the parts of an activity implementation as it is executed.

The color information is stored in a package called ‘Trace data for <name> (<date>)’, where <name> is the name of the simulated activity and <date> is the date and time when the simulation session started. You can save that package into a file of its own for future analysis and reference.

**Hint**

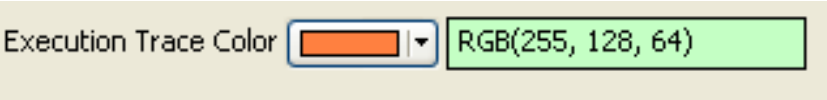
*If you save the trace data packages from multiple simulation sessions in their own files, you can later load all the files into the model in order to visualize the union of all execution paths taken during these simulation sessions. This can be useful when looking for parts of the activity model that are never executed (coverage analysis).*

To remove the colorization data, delete the trace data package from the model.

### Setting trace color

To set the color to use for tracing:

1. In the Model View select the Model Verifier build artifact which is located in the created Activity Simulation Model.
2. Open the Properties Editor (**Alt** + **Enter**)
3. Select **Activity Simulation Build Artifact** in the Filter list.
4. Set the **Execution Trace Color** property, either by clicking the button or entering the RGB value directly in the text field.




## Sequence Diagram Trace

You can use the Model Verifier sequence diagram trace functionality when simulating an activity. When you click the toolbar button for sequence diagram tracing during an activity simulation session, a dialog appears which provides the following options:

- **No tracing**
  Select this to turn off sequence diagram tracing.

- **Node based tracing**
  With this tracing the sequence diagram will show one lifeline for each activity node. Message lines are used to show the flow of control and data tokens between the activity nodes.
- **Partition based tracing**
  With this tracing the sequence diagram will show one lifeline for each partition in the activity implementation. For activity implementations that do not contain any partitions the default is to use node based tracing. The dialog allows you to fully customize which partitions and activity nodes to include in the trace.

You can bring up the trace dialog at any time during the simulation session in order to turn on or off the sequence diagram trace, or to switch between node based and partition based tracing.

## Breakpoints

You can use breakpoints to stop execution at specific activity nodes. A breakpoint will be hit just before the activity node is about to execute, which is when all needed tokens are available for it to consume. When multiple tokens are flowing in different parts of an activity model it can be slightly confusing because execution is not sequential, but is based on independent tokens. Such a situation is similar to the traditional debugging of a multi-threaded program.

Breakpoints can be set on the following nodes:

- Initial
- Activity/Action
- AcceptEvent
- SendSignalAction
- Decision/Merge
- Fork/Join
- ActivityFinal
- FlowFinal

## Supported Activity Nodes

The following activity nodes are supported for activity simulation:

- Initial
- Activity/action
- Decision/merge
- Fork/join
- Connector
- Accept event
- Send signal
- Activity final
- Flow final
- Object

Control and data flows, partitions, streaming pins and nested activities are also supported.

## Sending Signals to the Activity

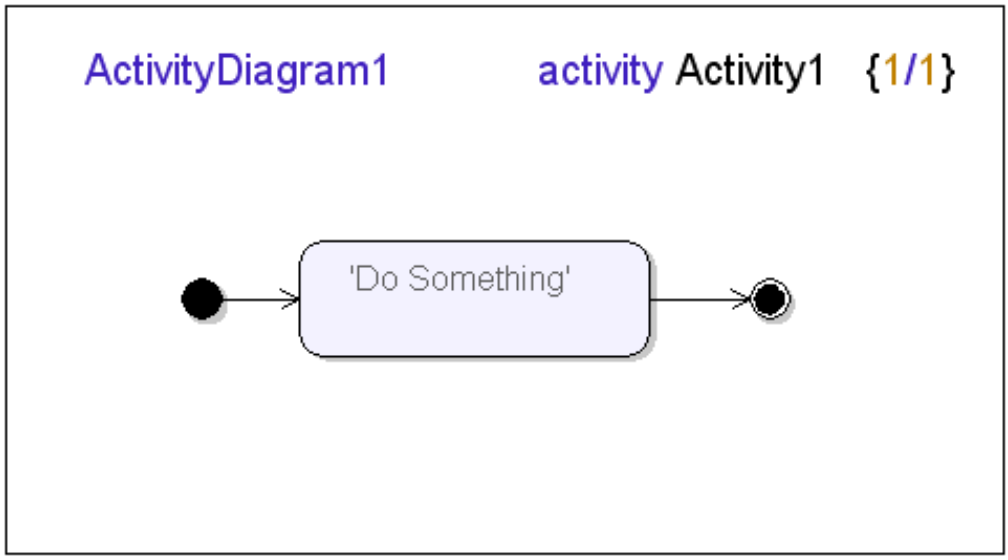
When Accept Event nodes are present in the activity, you can to use the normal Model Verifier Message view to send signals from the environment of the activity. To activate the Message view, click **View->Model Verifier Windows->Messages**.

Note that the receiver of the messages sent to the activity must be the instance in the simulation model called **Communicator**. This instance is part of the main activity.

## An Example

**Example 121: Performing an activity simulation** ———————————————

Assume we want to simulate an activity with the following implementation:




We activate the ADSim add-in and select Activity1 in the Model View. We then invoke the **Create simulation model** command in the ADSim menu. The following messages are displayed in the Message tab:

```
Running the ADSimGenerate
Preparing activity Activity1 for simulation...
Activity transformation completed!
Start simulation by launching the generated Model
Verifier build artifact Build_MV_Activity1.
```

We can then launch the Model Verifier from the generated build artifact and start the simulation. To single-step the execution we use the Next-Transition command (see User commands).

During execution we will get the following printout in the Model Verifier console:

```
Initial_1 executed
    Sent control token to Do Something_2
Do Something_2 executed
    Sent control token to ActivityFinal_3
ActivityFinal_3 executed
```

---

The messages that are traced to the Model Verifier console during simulation provide information on the following:

1. Execution of activity nodes.
2. Sending of control or data tokens from one activity node to another. If a data token is sent its type is also printed.

3. Reception of a token that does not lead to the execution of an activity node. The current number of tokens that have been received by the activity node is printed, as well as the total number of tokens that must be received before it can execute.

### Getting Started with Activity Simulation

The example projects available in the New wizard includes a model suitable for activity simulation. The project is called **umlActivitySimulation** and can be found in the **Samples** tab available from the **File->New** command.

# Web Service Simulation

It is possible to call web services in a model that is simulated with the Model Verifier. This feature allows you both to simulate web services you have developed yourself, and to include functionality provided by external web services in your UML model.

The capability to call web services from a UML simulation model is provided by an add-in called **WSSim**.

**Note**

*Web Service simulation functionality is only supported on Windows operating systems.*

### Activating the WSSim Add-In

To be able to call a web service from UML you must first activate the `WSSim` add-in. Perform the following steps:

1. In the **Tools** menu, select **Customize...**.
2. Click the Add-Ins tab and check the `WSSim` add-in.
3. Click **Close**.

**Note**

*Activating the WSSim add-in will also automatically activate the WSDL Add-in in order to be able to import the WSDL file of the web service to be called.*

If you create a new project for WSDL/XSD modelling there is an option **Support simulation of web services**. If this option is set the WSSim add-in will be loaded automatically.

## Calling a Web Service from UML

Before you can call a web service from your UML model you must first import its WSDL file into Tau. This is done using the WSDL/XSD Import Wizard. It is recommended to have the **Model** node selected when using the wizard so that the resulting WSDL package is placed as a new top-level package in the model. If you are working in the WSDL view you can select the project node instead, since the Model node will not be visible then.

In the WSDL/XSD Import Wizard there are two checkboxes available:

- **Generate UML Web Service Interface**
  By selecting this option a UML model acting as an interface for the imported web services will automatically be generated. See Generating a UML web service interface from a WSDL package for more information about this interface model.
- **Generate web service consumer**
  By selecting this option the importer will create a template model for consuming the imported web services. See Generating a web service consumer for more details.

### Generating a UML web service interface from a WSDL package

When a UML interface model is created for an imported WSDL package the following happens:

1. The UML web service interface package is created next to the WSDL package. It constitutes a UML API for the web services described by that WSDL package and contains definitions which you can use from your UML model in order to call the web services. For example, a web service is represented by means of an active class which exposes appropriate operations, allowing you to call the web service from UML.
2. Glue code needed for calling the web service is generated. This code is placed in a directory called `__wssim<N>`, where N is a number appended to make the name unique. This directory is located in the project directory.

If you have already imported the WSDL without using the options described above, you can use an explicit command for generating the UML web service interface. To do so right-click on the WSDL package, and perform the command **"Generate/Update UML Web Service Interface"** which the `WSSim` add-in has made available in the context menu.

The context menu of a WSDL package also contains a command called **"Generate UML API"**. It only performs step 1) above, skipping glue code generation, and can be useful if the intent only is to represent usage of a web service in a UML model, and to simulate the model but not call the real web services. For example, in case the web service is not yet implemented, it can obviously not be called for real. But a client UML model containing calls to it, can still be developed and simulated.

### Using the generated UML web service interface

Now you are ready to start using the definitions of the generated UML web service interface package from your UML model. Typical steps include:

1. Adding an <<import>> dependency from a package in your model to the generated UML web service interface package.
2. Creating an instance of the active class that represents the web service. This class is located in a subpackage called "Services" in the generated UML package.
3. Set the URL of the web service. This is done by calling the "set_serviceUrl" operation that is available on the web service active class. If the URL is the same as specified in the imported WSDL file (in the `<soap:address>` tag) you may pass as argument the Charstring attribute located in the "URLs" subpackage of the generated UML package. That attribute has the URL from the WSDL file stored as its default value.
4. Calling an operation on the web service active class which performs the actual web service call. This operation is prefixed with "call_" and there are usually several overloaded versions to choose between. Which one to use depends on how you want to specify the actual arguments to the web service and how to handle errors that may arise. See Error Handling below for more information.
5. Adding a Model Verifier build artifact for your client model. You should use a Make-Template file for this build artifact which can be found in the generated `__wssim` target directory (the file is called `soap.tpm`).

### Generating a web service consumer

The `WSSim` add-in provides a useful command in the context menu of the generated UML API package, called **"Generate Web Service Consumer"**. As the name suggests this command will generate a template model which consumes the imported web services. Basically this means an automation of the above mentioned steps. The template model will contain a Model Verifier build artifact which can be launched to test the web services instantly.

### Using multiple web services

It is possible to use more than one web service from the UML model. This can be achieved in several ways. The easiest is if all the WSDL files for the web services to use are imported from the beginning using the WSDL/XSD Import Wizard. A template web service consumer model generated by the wizard will in that case consume all the imported web services.

It is also possible to do this step after the WSDL/XSD Import Wizard has been run. By selecting multiple WSDL packages in the Model View and invoking the context menu command "Generate/Update UML Web Service Interface", then one interface package will be generated for each selected WSDL package, but they will share the same `__wssim` directory, containing glue code for all WSDL packages. You can then select these generated packages and invoke the context menu command "Generate Web Service Consumer" in order to generate one single model for consuming all the selected web services.

If you need to start using a new web service from an already existing consumer model, you could import the new web service separately, generating a template consumer model for it. Then you can copy relevant parts from that generated consumer model into your existing model.

## Type Mapping

WSDL elements representing input and output data from a web service are translated to UML classes by `WSSim`. These classes are located in a subpackage called "Types" in the UML API package. The classes contain attributes typed by ordinary UML types. This is achieved by mapping XSD types to UML types according to the table below:

| XSD Type | UML Predefined Type |
|---|---|
| anyType | Charstring |
| anySimpleType | Charstring |
| duration | Charstring |
| dateTime | Integer |
| time | Integer |
| date | Charstring |
| gYearMonth | Integer |
| gYear | Integer |
| gMonthDay | Integer |
| gDay | Integer |
| gMonth | Integer |
| boolean | Boolean |
| base64Binary | Charstring |
| hexBinary | Charstring |
| float | Real |
| double | Real |
| anyURI | Charstring |
| QName | Charstring |
| NOTATION | Charstring |
| string | Charstring |
| normalizedString | Charstring |
| token | Charstring |
| language | Charstring |
| Name | Charstring |
| NMTOKEN | Charstring |
| NCName | Charstring |
| NMTOKENS | Charstring |

| XSD Type | UML Predefined Type |
|---|---|
| ID | Charstring |
| IDREF | Charstring |
| ENTITY | Charstring |
| IDREFS | Charstring |
| ENTITIES | Charstring |
| decimal | Integer |
| integer | Integer |
| nonPositiveInteger | Integer |
| long | Integer |
| nonNegativeInteger | Integer |
| negativeInteger | Integer |
| int | Integer |
| unsignedLong | Integer |
| positiveInteger | Integer |
| short | Integer |
| unsignedInt | Integer |
| byte | Integer |
| unsignedShort | Integer |
| unsignedByte | Integer |

Simple XSD types containing enumerations are mapped to UML Datatypes with literals. Simple XSD types without enumerations are mapped to syn-types.

### Date and Time

As can be seen in the above table the XSD `date` type is represented by a `Charstring` in UML. The format of this string is `YYYY-MM-DD`. To facilitate working with the XSD `dateTime` type some utilities exist in the `WSSType` UML library.

## SOAP Headers

Some web services require the use of SOAP headers for passing data additional to the parameter data. Typically SOAP headers are used for contextual data which remains constant in multiple calls to a web service. Examples of such contextual data may include user account information (user name and password), or a session identifier in a conversation spanning over multiple web service calls.

In UML SOAP header data is represented by means of attributes on the web service active class. To set the SOAP header data, you should assign values to these attributes before calling the web service operations. The attributes have private visibility but there exist public accessor operations for getting and setting their values.

## Asynchronous Web Service Calls

The generated UML web service interface model supports calling web services asynchronously. The web service is represented by an active class which has a simple state machine. This state machine can handle signals representing asynchronous calls of the web service. The definition of these signals can be found in the “Interfaces & Signals” subpackage of the generated UML package. There is one signal for passing input parameters in the web service call, and one reply signal that carries output parameter data.

## Error Handling

Some overloaded versions of the generated web service “call_” operations support error handling while some do not. If you are not interested in handling errors that may arise when calling a web service you should call an operation with the “NR” suffix (NR = No Return value). If, however, you want to handle such errors you should call one of the other “call_” operations which return an instance of the `SOAP_CallResult` class, defined like this:

```
class SOAP_CallResult
{
  public Boolean wasError;
  public Charstring errorDescription;
}
```

The ‘wasError’ attribute is set to true upon error, and the ‘errorDescription’ attribute is then set to an error message.

### Error Reporting

When a web service invocation error is detected it is reported in the following ways:

- It will be reported in the WSSim tab. The subject of the error is the UML operation that represents the call to the web service, which makes it easy to find which web service invocation that fails, in case multiple web services are called in a model.
- It will be printed in a log file called `tlog_mv_soap.log`, which will be placed in the Model Verifier target application directory.

## Troubleshooting

Most web services can be called from Tau but you should be aware of some limitations that may prevent certain web service calls from working, or require work-arounds.

### Web service call-backs

Certain web services require the client to implement certain call-back interfaces. Such web services cannot be directly called from Tau, since there is no automated support for creating such call-back web services realizing the required interfaces.

### Non-Supported types

Web services having parameters typed by a non-supported type cannot be called from Tau. See Type Mapping for a list of all supported types and their mapping to predefined UML types.

### Non-Supported bindings

The only web service binding that is currently supported is SOAP 1.1. Web services which use other bindings, such as SOAP 1.2 or HTTP, can not be called from Tau.

# 12

# *Model Verifier Reference*

This section is a reference manual to the Model Verifier. Instructions on how to use the Model Verifier are available in "Verifying an Application" on page 443.

# Trace Levels

This section lists the following trace and tracking levels:



## Textual trace levels

Textual trace levels 0 to 6 are available.

### Trace level 0

The textual trace is disabled.

### Trace level 1

This level only displays signals sent to and received from the environment.

### Trace level 2

This level shows what causes a transition to occur. An example of this information is:

```
*** TRANSITION START
*       PId    : p1:1
*       State  : Idle
*       Input  : Plong
*       Sender : p2:1
*       Now    : 0.0000
```

### Trace level 3

This level adds important actions to the output, for example signal sending, next state, stop, etc.

```
*** TRANSITION START
*       PId    : p2:1
*       State  : Idle
*       Input  : Pling
*       Sender : p1:1
*       Now    : 0.0000
*    OUTPUT of Plong to p1:1
*** NEXTSTATE  Idle
```

### Trace level 4

This level shows additional tasks like action and decision.

### Trace level 5

This level adds results of actions, for example null transitions, discarded signals, etc.

### Trace level 6

This level also prints parameter values for signals, timers, etc.

## Execution tracking levels

The following levels are available for tracking in state machine diagrams:

- **Never**

  The execution tracking is disabled.
- **When yielding control**

  If the current execution point is inside a transition and the execution is interrupted, the next statement to be executed is tracked.

  If the current execution point is outside a transition, the last statement that was executed in the last transition is tracked.
- **Continuously**

  Each statement being executed is tracked while the execution is going on.
- **According to Parent**

  This level sets the tracking level identical to the parent element in the model.

## Sequence diagram trace levels

A sequence diagram trace always includes a sender instance and a receiver instance.

- **Never**

  This level disables the trace of the events in the sequence diagram.

- **If source or target**

  This level enables trace only if the receiver's sequence diagram trace level has not been set to Never.
- **Always**

  This level enables sequence diagram trace regardless of the receiver's sequence diagram trace level.
- **On block level**

  This level enables trace of the given agent, hiding enclosed agents.
- **According to Parent**

  This level sets the trace level identical to the sender's parent element in the model.

# User Interface Commands

The Model Verifier user interface commands listed in this section can be started either from the toolbar or from the Verify menu. The commands can also be run from the Model Verifier console.

**See also**

"Syntax of commands" on page 507

"Console commands" on page 510

## List of user interface commands

The Model Verifier user interface commands listed in this section can be started either from the toolbar or from the Verify menu. The commands can also be entered in the Model Verifier console.

- **Go**

  The execution runs until the users stops it (using for example the Stop Debugging command) or when a breakpoint is reached.

- **Next Transition**

  Use this command to run the system until the next transition in the State machine diagram. The execution stops at the next State symbol or at a Stop symbol.

  If the `Realtime` option for Simulation kind is used and the next transition is a timer scheduled in the future, (more than a second from now), then the command Next-Transition will run the simulation for one second and then pause.

  Giving a Next-Transition command within a transition will execute the remaining part of the transition.

- **Step Into**

  Use this command to step statement by statement through the transitions. (An action symbol may contain several statements.)

  The Step into command also steps into operations.

- **Step Local**

  This command will step one statement in the current state machine, as opposed to the Step Over command that steps to the next statement in any state machine. (This command is not available from the tool bar.)

- **Step Over**

  This command is similar to Step into, with the difference that all steps in an operations are executed at once.

- **Step Out**

  Use this command to execute the remaining steps in an operation, including the return symbol.

- **Insert/Remove Breakpoint**

  Use this command to insert or remove breakpoints. A red dot is attached to the symbol where the breakpoint has been inserted.

- **Break**

  Use this command to break the execution. The execution continues from the location where the execution was halted when the go command or one of the step commands is executed.

- **Restart**

  Use this command to reset the application to its initial state.

- **Stop Debugging**

  Use this command to stop the debug session.

# Console

The console commands in the following section can be used to run the Model Verifier from the Model Verifier console window.

## Input and output of values of passive types

Values are represented as UML Expressions However, there are situations where UML expressions cannot be used. If that is the case, values must be represented as describes in this section.

The following situations describe when values as UML expressions cannot be used:

- Values displayed in textual execution traces when the command Set-Trace has been used.
- Values that are found in scenarios and Model Verifier configuration files that were created in previous versions of Tau (2.2 and earlier).
- Values that you can specify in Watch windows using the syntax: `[[SDL:value]]`. This may be used in cases where no UML expressions are available, for instance for general array types.
- As arguments for the Set-Timer command.

The syntax of literals of the predefined data types follows the SDL definition of literals. There are, however, some extensions that will be described where applicable. As an option, it is also possible to use the ASN.1 syntax for values. On input both value notations are supported, while there are commands to select the type of output to be produced (SDL-Value-Notation and ASN1-Value-Notation).

### Integer and 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 notation with an '`E`' for exponents, in the same way as in many programming languages.

**Example 122: 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 described by Example 123 on page 501.

**Example 123: Real values in ASN.1 syntax**

```
{mantissa 23456, base 10, exponent -3}
```

This is the value 23.456.

---

### Time and Duration values

The format for Time and Duration values follows the SDL standards, that is real values without exponent 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 124: 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 either using hexadecimal notation, `0xFF` or according to the SDL standard, including the literals for the non-printable characters.

### 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 character (‘), is however replaced by the double quote character (“).

**Example 125: 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 active class and an instance number, which is an integer greater than 0.

The first instance of an active class 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 instance name and No is the instance number.

On input the 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 instance of the active class. If, however, the command parameter is a unit that might be an instance of an active class, only a colon (':') is allowed between the name and the instance number and the colon must follow directly after the class 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 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.

### BitString

For BitString 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 OctetString can also be used.

### Octet

The syntax used for an octet value is two HEX digits. Examples:

```
'00'h '46'h 'F2'h 'a1'h 'CC'h
```

The characters `0-9`, `a-f`, and `A-F` are allowed.

### OctetString

The syntax for `OctetString` 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.

### ObjectIdentifier

The sort ObjectIdentifier is 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. 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.

### 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.

### Passive class values

A passive class 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 126:**

```
(. 23, true, 'a' .)
```

If ASN.1 syntax is used, the component names will also be present.

**Example 127:**

```
{ 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.

Optional components that are not present, will not be printed. That means that there is an empty position between two commas.

### 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.

### 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). 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 C Code Generator has selected for the array implementation. The easiest way to determine which syntax to use on input is to look at an attribute of the array sort. If an array is a simple array (that is index type is a simple type with one range condition and a limited range), the SDL syntax for an array value is according to Example 128 on page 505.

**Example 128: Simple array value**

```
(: 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 Example 129 on page 505 should be used.

**Example 129: General array value**

```
(: (others:2), (10:3), (11:4) :)
```

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 (that is 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 130:**

```
(. 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 131: Power set value**

```
[ 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 132: Bag value**

```
{ 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 133: Multiple identical values in bag**

```
{ 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.

### Own and ORef values and passive object references

There are two possible syntaxes for pointer type values (Own and ORef). 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 134: Address notation**

```
0x23A20020

HEX(23A20020)
```

## See also

"Complex signal parameter values" on page 459 in Chapter 11, *Verifying an Application* for more information on how to enter complex values to signals in the **Messages** window in UML syntax.

# Syntax of commands

### Introduction

As there are two types of commands, the syntax between them differ slightly. The information below is mainly applicable for commands that do not start with '!'.

### 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-Breakpoints`. The command name may be typed `List-B` or `L-B`. However, as there are more than one command beginning with the word "List", typing `List` could not distinguish between them. In that case the message

```
Command was ambiguous, it might be an abbreviation of:
```

would be displayed followed by a list of all commands starting with "List".

There is no distinction made between upper and lower case letters.

### Parameters

Command parameters are separated by one or several spaces, carriage returns or tabs. A colon is also accepted between a name and an instance number, when a Pid value is required. If the parameter list following a command name is not complete you will be notified in the Output window. You will be asked to provide 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 keywords **Sender**, **Parent**, **Self** and **Offspring** cannot be abbreviated.*

### Matching of parameters

When a (possibly abbreviated) parameter name is entered and is to be matched with a non-abbreviated name, only names in the entity class of interest are considered. If a name is expected as parameter, only the names denoting active classes will be part of the search for the full name.

Signal names and timer names are in the same entity class; formal parameters of state machines and attributes are in the same entity class; every other type of name is in an entity class of its own.

Knowledge of previous parameters is used to narrow the search for a given parameter name.

### Qualifiers

Names can cause problems, if, for example, there are two active classes with the same name in two different ‘blocks’, or if the ‘system’ and an active class contain signal definitions with the same name.

In the first situation the class name will always be ambiguous and in the second case the name of the signal defined in the ‘system’ 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 must 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. 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.

### Instance path

Instance path is the UML representation of a resulting qualifier path originating from a Manifest relation of an object. Examples of instance paths are:

```
Match.p1
{Match.p1[2]}
```

The path must be the full path, starting from the top-level instance. If the object is specified with brackets ([ ]), braces ({}) must be used as in the example above.

### Signal and timer parameters

Parameters of signal instances and timer instances are entered 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 135: 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 (that is the computer's memory for the value is set to 0). By entering ')', the rest of the parameters are given "null" values.

**Example 136: Signal parameters in 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. A command is either executed completely or not at all.

## Console commands

### ? (Interactive Context Sensitive Help)

**Parameters**: (None)

When typing a '?' (question mark), a list of all possible allowed values, commands or types at the current position is displayed. After the list has been presented, you must type one of the available values in order to continue.

Typing '?' at the prompt level gives a list of all available commands, after which a command can be entered.

**Note**

*This command does not apply for commands that start with '!'*

**!U2::Debug add_message**

**Parameters**:

```
<sender path> <signal path> <connector name>
<receiver path> < parameter string>
```

Use this command to add messages to your messages list.

The `<sender path>` indicates the path to the sending agent and `<receiver path>` indicates the path to the receiving agent. An instance within an instance is separated with a period '.'.

The `<signal path>` indicates the path to the signal. An example of a signal path would be:

```
::<object>::<signal name>
```

**!U2::Debug assign**

**Parameters**:

```
<object reference> <value>
```

This command allows you to assign a new value to an object. Use the command !U2::Debug path2object to the locate the object reference.

**!U2::Debug break**

This command is equivalent to "Break" on page 499.

**!U2::Debug delete**

**Parameters**:

```
<object reference>
```

This command executes the delete command on a specified object. Use the command !U2::Debug path2object to the locate the object reference.

### !U2::Debug display

**Parameters**:

```
<object reference>
```

This command allows you to display the value of the object in the console. Use the command !U2::Debug path2object to the locate the object reference.

### !U2::Debug echo

**Parameters**:

```
<arguments>
```

This command displays all arguments in the Model Verifier tab in the Output window.

### !U2::Debug exec

**Parameters**:

```
<command>
```

This command will execute the parameter as a command.

**Example 137:**

```
!U2::Debug exec "Go-Forever"
```

### !U2::Debug go

This command is equivalent to “Go” on page 498.

### !U2::Debug new

**Parameters**:

```
<object reference>
```

This command executes the new command on a specified object. Use the command !U2::Debug path2object to the locate the object reference.

### !U2::Debug next transition

This command is equivalent to “Next Transition” on page 499.

### !U2::Debug open

**Parameters**:

```
<path name> <optional argument>
```

Use this command to load a scenario files (`.ttdscn`) or when you Load Model Verifier configurations (`.ttdcfg`). The optional argument is only valid for configuration files.

- If the argument `-merge` is added, the objects in the configuration file you want to open, merge with the objects in the current configuration.
- If the argument is omitted, the objects in the configuration file overwrite the corresponding objects in the current configuration.

**Example 138: Command with parameters**

```
!U2::Debug open c:/project/test.ttdcfg -merge
```

### !U2::Debug output

**Parameters**:

```
-from <sender path> -to <receiver path> [-via <connector
name>]: <signal name and parameters in parenthesis>
```

This command sends the specified message.

### !U2::Debug path2object

**Parameters**:

```
<instance path>
```

This command returns the object reference for the Instance path. The object reference is needed in order to execute other commands.

### !U2::Debug restart

This command is equivalent to “Restart” on page 499.

### !U2::Debug save

**Parameters**:

```
<path name> <optional arguments>
```

Use this command to save a scenario or when you Save Model Verifier configurations for future use. Scenario files must have the extension `.ttdscn`, while configuration files must have the extension `.ttdcfg`

The optional arguments only apply for configuration files and they decide which objects in the configuration that should be saved. If there are no arguments, all objects are saved. The available arguments are:

| Argument | Definition |
|---|---|
| `-messages` | Messages will be saved |
| `-breakpoints` | Breakpoints will be saved |
| `-gr_trace` | Execution tracking levels will be saved |
| `-msc_trace` | Sequence diagram trace levels will be saved |
| `-watches` | The root elements in watch windows will be saved |

Examples of parameters for this command are:

```
c:/project/test.ttdscn

c:/project/test.ttdcfg -messages -msc_trace
```

### !U2::Debug set_breakpoint

**Parameters**:

```
<U2 statement>
```

This command sets a breakpoint on the UML statement specified as a reference to the model.

### !U2::Debug set_msc_trace

**Parameters**:

```
<object reference> <integer trace level>
```

This command sets the Sequence diagram trace levels for the specified object. Use the command !U2::Debug path2object to the locate the object reference.

The values for `<integer trace level>` are:

0=Never

1=If source or target

2=Always

3=On block level

4=According to Parent

**Example 139: Setting the trace level**

```
!U2::Debug set_msc_trace [U2::Debug path2object {CoffeeMachine}] 3
```

### !U2::Debug set_replay

**Parameters**:

`<boolean>`

This command sets the Model Verifier in replay mode if the boolean expression is true.

### !U2::Debug set_tracking_level

**Parameters**:

`<object reference> <integer trace level>`

This command sets the Execution tracking levels for the specified object. Use the command !U2::Debug path2object to the locate the object reference.

The values for `<integer trace level>` are:

0=Never

1=When yielding control

2=Continuously

3=According to Parent

### !U2::Debug start_msc

**Parameters**:

`<Trace value>`

This command starts the logging of events in a sequence diagram on the fly. Stop the log by using the command !U2::Debug stop_msc. Valid trace levels are:

0 = only messages

1 = messages and states

2 = full trace, actions, messages and states

**!U2::Debug step in**

This command is equivalent to "Step Into" on page 499.

**!U2::Debug step local**

This command is equivalent to "Step Local" on page 499.

**!U2::Debug step out**

This command is equivalent to "Step Out" on page 499.

**!U2::Debug step over**

This command is equivalent to "Step Over" on page 499.

**!U2::Debug stop_msc**

This command stops the logging of events in a sequence diagram.

**Activity-Mode**

**Parameters**: (None)

This command sets a number of Model Verifier settings appropriate for activity simulation. When the Model Verifier operates in activity mode it does not show the next statement to execute. Instead the focus is on the execution of activities, and if the ADSim add-in is activated, the execution will be tracked in activity diagrams.

To leave the activity simulation mode, use the command Statemachine-Mode.

### Assign-Value

**Parameters**:

```
<object reference> <value>
```

This command is equivalent to "!U2::Debug assign" on page 511.

### ASN1-Value-Notation

**Parameters**: (None)

The value notation used in all outputs of values is set to ASN.1 value notation.

### Breakpoint-Output

**Parameters:**

```
<Signal name> <Receiver class name> <Receiver instance
number> <Sender class name> <Sender instance number>
<Counter> <Optional breakpoint commands>
```

A breakpoint is inserted and a breakpoint command is defined. The breakpoint condition defines sending one or several signals and is specified by the parameters.

The `<Counter>` parameter indicates how many times the breakpoint condition should be true before the execution breaks. Default value for this parameter is 1, which means that the execution stops each time the breakpoint condition is true.

The `<Optional breakpoint commands>` parameter can be used to execute one or more Model Verifier commands when the breakpoint is triggered. Commands should be separated by " ; ", that is space, semicolon, space.

### Breakpoint-Transition

**Parameters**:

```
<Class name> <Instance number> <State name> <Signal name>
<Sender class 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 execution stops immediately before the transition is started. The breakpoint condition matches one or sev-

eral 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 breakpoint is activated. Default value for this parameter is 1, which means that the execution stops each time the breakpoint condition is true.

The <Optional breakpoint commands> parameter can be used to give console commands that should be executed when the breakpoint is triggered. Console commands should be separated by " ; ", that is space, semicolon, space.

### Breakpoint-Variable

**Parameters**:

```
<Attribute name> <Optional breakpoint commands>
```

This command inserts a breakpoint on the specified attribute in the instance of active class given by the current scope. If the value of the attribute is changed, the execution breaks. The value is only checked between symbols and between assignment statements in actions.

The breakpoint is also triggered when the attribute no longer exists, that is the Pid containing the attribute is stopped or the operation containing the attribute has reached its end. In this case, the breakpoint is automatically removed.

The `<Optional breakpoint commands>` parameter can be used to execute one or more Model Verifier commands when the breakpoint is triggered. Commands should be separated by " ; ", that is space, semicolon, space.

### Call-env

**Parameters**: (None)

This command calls xInEnv for a Model Verifier generated with Simulation kind set to `With Environment`.

### Cd

**Parameters**:

```
<Directory>
```

This command changes 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. If only one log file is being used, the entry number parameter can be omitted.

### Command-Log-Off

**Parameters**: (None)

The command log facility is turned off. Compare with the command Command-Log-On for details.

### Command-Log-On

**Parameters**:

```
<Optional file name>
```

This command enables logging of all the commands that are typed in the Model Verifier console. The first time this command is used, a file name for the log file must be stated. 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 functionality is turned off. When activated, 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.

### Create

**Parameters**:

```
<object reference>
```

This command is equivalent to "!U2::Debug new" on page 512.

### 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 connector to "`env`" in the sequence diagrams trace. The default is "Off".

### Display-Array-With-Index

**Parameters**:

```
"On" | "Off"
```

When value is On and an array is displayed in execution traces, 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”.

**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**:

`<object reference>`

This command is equivalent to “!U2::Debug display” on page 512.

**Exit**

**Parameters**: (None)

This command is equivalent to “Stop Debugging” on page 499.

**Go**

**Parameters**: (None)

This command is equivalent to “Go” on page 498.

**Go-Forever**

**Parameters**: (None)

The execution will continue until a breakpoint becomes active or that you stop the execution manually. To stop the execution of transitions, press `<Return>` (and only this key).

If the UML model becomes completely idle (no possible transition and no active timer), the simulation waits for external stimulus.

### Include-File

**Parameters**:

`<File Name>`

This command provides the possibility to execute a sequence of console commands that are 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-Breakpoints

**Parameters**: (None)

This command lists all active breakpoints in the model. Each breakpoint is assigned en entry number.

### List-GR-Trace-Values

**Parameters**: (None)

This command lists the trace value that is being used for textual tracking.

### List-MSC-Log

**Parameters**: (None)

This command returns the current status of the sequence diagrams log (off / interactive / batch).

### List-Ready-Queue

Deprecated (well, not at 100%, as current procedure is not given in TT/D: let us say this is a bug). no included

### List-Signal-Log

**Parameters**: (None)

Print information about currently active signal logs. Each log is assigned an entry number.

### 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).

### Log-Off

**Parameters**: (None)

This command turns off the interaction log facility, compare with Log-On.

### Log-On

**Parameters**:

```
<Optional file name>
```

This command takes an optional file name as parameter and enables logging of all the interaction between you and the simulation program 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 if you enter the command without a file name, more information will be appended to the previous log file. Each time you enter the command with a file name, the old log file will be closed and a new file will be opened 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.

### Next-Local-Statement

**Parameters**: (None)

This command is equivalent to “Step Local” on page 499.

### Next-Statement

**Parameters**: (None)

This command is equivalent to "Step Into" on page 499.

### Next-Transition

**Parameters**: (None)

This command is equivalent to "Next Transition" on page 499.

### Next-Visible-Transition

**Parameters**: (None)

A number of transitions are executed up to and including the next transition with a trace value that is not disabled. For a timer output transition, it is the trace value for the corresponding instance that is considered.

This command should be used with care, as it might result in executing the simulation program forever if no transition with active trace is ever executed. To stop the execution of transitions, press `<Return>`.

### Now

**Parameters**: (None)

The current value of the simulation time is printed.

### 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 file with the name specified as parameter is created in the work directory containing the relevant information. The coverage file always reflects the situation from the start of the simulation.

**Note**

*The specified file is always overwritten, that is 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 active class. Secondly the file contains detailed information about how many times each symbol and each state - input combination is executed.

**Example 140: 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
```

The information should be interpreted in the following way:

- As **transitions**, the number of executed signal receipt symbols + guards on triggered transition + start symbols are counted.
- As **symbols**, the number of executed symbols (action, signal sending, decision, set, reset, call, stop, return, create, nextstate, input, guard on triggered transition, 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’ and for each active class. The relative numbers for the active classes are, of course, relative to the numbers for the ‘system’.
- **MaxQ** is the maximum input port queue length for any instance of the active class. Start up signals, used to implement create, and guard on triggered transition 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, an operation is defined within or after another operation.

**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.*

### 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 console will become active when the value of the time first becomes equal to the time value given as parameter.

Relative time values can be given using the '+' sign. Entering "+5.0" as parameter is interpreted as the time value NOW+5.0.

### Quit

**Parameters**: (None)

This command is equivalent to "Stop Debugging" on page 499.

### REF-Address-Notation

**Parameters**: (None)

REF values (pointers introduced using the Own and ORef templates) 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 (compare with command REF-Value-Notation) can be used.

**Note**

*This command only applies to values that are displayed in execution traces in the Output window.*

### REF-Value-Notation

**Parameters**: (None)

REF values (pointers introduced using the Own and ORef templates) 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 141:**

```
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 nth NEW in the printed value.

**Example 142:**

```
NEW( (. 1, NEW( (. 2, OLD 1 .) ) .) )
```

The Null value is printed as `Null`. On input both this syntax and the Address-Notation (compare with command REF-Address-Notation) can be used.

**Note**

*This command only applies to values that are displayed in execution traces in the Output window.*

### Remove-All-Breakpoints

**Parameters**: (None)

This command removes all inserted breakpoints.

### Remove-Breakpoint

**Parameters**: <entry number>

This command removes breakpoints in the model. The `List-Breakpoints` command lists the entry numbers for each inserted breakpoints.

### 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.

**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 on the system, Reset-Trace on the system is considered to be equal to setting the trace value to 0.

**Note**

*The user interface is not updated when this command is used.*

**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.

**Save-Breakpoints**

**Parameters**:

```
<File name>
```

This command saves all breakpoints in a file. It is written as a text file with commands.

### 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 entity. If no unit is specified the GR trace value is assigned to the whole model. The initial GR trace value of the system is 0, i.e. no execution tracking in the diagrams is enabled. The value of 1 enables execution tracking.

### Set-MSC-Trace

**Parameters**:

```
<Optional unit name> <Trace value>
```

This command is equivalent to "!U2::Debug set_msc_trace" on page 514.

### Set-Timer

**Parameters**:

```
<Timer name> <Timer parameters> <Time value>
```

The result of the command is exactly the same as if the instance of active class in the current scope had executed a set timer action. If the set action causes a timer signal to be removed and this signal is selected for the next transition, the instance will execute an implicit nextstate action.

### Set-Trace

**Parameters**:

```
<Optional unit name> <Trace value>
```

The Textual trace levels is assigned to the specified unit, or node. If no unit is specified the trace value is assigned to the whole model. The trace value specifies which type of information that will be displayed in the textual trace. The initial trace value for the model is 4, while it is undefined for all other units.

**Show-C-Line-Number**

**Parameters**: (None)

The command displays where in the C code the execution is suspended. The file name and the line is displayed in the Output window.

**Show-Next-Symbol**

**Parameters**: (None)

The symbol in turn to be executed will be selected in a state machine diagram.

**Show-Previous-Symbol**

**Parameters**: (None)

The last executed symbol will be selected in a state machine diagram.

**Show-Versions**

**Parameters**: (None)

The versions of the SDL to C Compiler and the run-time kernel that generated the currently executing program are presented.

**Signal-Log**

**Parameters**:

`<Unit name> <File Name>`

This commands starts logging of signals to a specified file. The `<Unit name>` parameter specifies which entity that will be logged. Only signals to and from this entity will be listed in the file. Use the command Close-Signal-Log when the log is complete.

To view the log use the command List-Signal-Log.

### Start-Batch-MSC-Log

**Parameters**:

`<Symbol level> <File name>`

This command starts the logging of events which can be translated into the corresponding sequence diagram events in a log file. Stop the log by using the command Stop-MSC-Log.

The results will be stored in a log file.

The symbol level parameter determines if states and actions should be included in the log. For a description of the possible symbol level values.

The file name parameter to this command can be any valid file name.

### Start-env

**Parameters**: (None)

This command initiates the possibility to call environment API functions from a Model Verifier generated with Simulation kind set to `With Environment`.

### Statemachine-Mode

**Parameters**: (None)

By default the Model Verifier operates in state machine mode, meaning that execution can be tracked in state machine (and text) diagrams. However, when simulating activities the focus is more on the execution of activities, and a special activity mode is then used instead (see the Activity-Mode command). Use the Statemachine-Mode command to leave the activity simulation mode.

### Stop-env

**Parameters**: (None)

This command stops the calling of environment API functions from a Model Verifier executable. This command is only available when a Model Verifier has been generated with Simulation kind set to `With Environment`.

**Stop-MSC-Log**

**Parameters**: (None)

This command stops the logging of sequence diagram events. In the case of a batch mode logging, the log file will be closed. Compare with the command Start-Batch-MSC-Log for more details.

Following this command, it is possible to log the rest of the session on a new file.

## Special console commands

In the following is a list of console commands that are for various reasons not fully documented. They are normally not necessary to use and should only in rare occasions be considered, typically on recommendations from Tau Support in combination with workarounds to complex simulation situations. Some of these commands can be considered deprecated as they are similar or identical to other menu and toolbar commands.

**%**

**Breakpoint-At**

**Define-At-Delay**

**Define-Continue-Mode**

**Define-Delay**

**Down**

**Examine-PId**

**Examine-Signal-Instance**

**Finish**

**List-Input-Port**

**List-MSC-Trace-Values**

**List-Process**

**Nextstate**

**Next-Symbol**

**Output-From-Env**

**Output-Internal**

**Output-None**

**Output-To**

**Output-Via**

**Performance-Simulation**

**Print-Paths**

**Rearrange-Input-Port**

**Rearrange-Ready-Queue**

**REF-Deref-Value-Notation**

**Remove-At**

**Remove-Delay**

**Remove-Signal-Instance**

**Reset-MSC-Trace**

**Restore-State**

**Save-Delay**

**Scope**

**Set-Scope**

**Show-Breakpoint**

**Stack**

**Start-Interactive-MSC-Log**

**Step-Statement**

**Step-Symbol**

**Stop**

**SymbolTable**

**Up**

**xSet**

# Replay Mode

In replay mode you can record:

- Execution steps
- User commands

## Execution steps

The following execution steps are recorded in a scenario:

- A **transition** is described by the path of the active class instance, the state from which it starts, and the signal it consumes. This is displayed as:

  ```
  active_class_instance_path: from state_path input
  signal_path
  ```

- A **time-out** is described by the path of the active class instance which sets the timer, and the path of the timer. This is displayed as

  ```
  active_class_instance_path: timer timer_path
  ```

## User commands

User commands recorded in a scenario are the commands which modify the state of the application:

- **Output** is described by the information that is available in the Messages window. It is displayed as:

  ```
  output from sender_path [via connector name] to
  destination_path : signal_path [ (parameters values) ]
  ```

- **New** is described by the path of the parent of the created object. This is displayed as:

  ```
  new object_path
  ```

- **Delete** is described by the path of the deleted object, This is displayed as:

  ```
  delete object_path
  ```

- **Rearrange** is described by the path of the parent object, and source and destination positions. This is displayed as:

  ```
  collection_path: source_index -> destination_index
  ```

- **Assignment** is described by the path of the assigned object, and the new value expression. This is displayed as:

```
assigned_object_path = new_value
```

# Dynamic Errors

Error messages are presented as described in Example 143 on page 535.

**Example 143: Dynamic Error Printout**

```
************************ ERROR ************************
Error in SDL Decision: Value is 12:

Entering decision error state
TRANSITION
  Process         : Ctrl:1
  State           : Idle
  Input           : Coin
  Symbol          : #SDTREF(U2,"u2:C:\Program
Files\Telelogic\TAU_4.2\examples\CM\
CM.u2#7t6K1VwL19VLrn8XRELUlHzI|pos(1,18)")
TRACE BACK
  Process         : Ctrl
  System          : CoffeeMachine
*******************************************************
```

## Action on dynamic errors

After a dynamic error, the execution of the simulation program continues until the current symbol is ended. The following actions will be taken:

- If the error is a signal sending error the signal will not be sent.
- If the error is a decision error the instance of active class will immediately be placed in a *decision error state*. The input port will not be affected when the decision error state is entered. All signals sent to an instance of active class in a decision error state will be saved in the input port.
- If the error occurred during the creation of static instances of active classes, that is the initial number of instances is greater than 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 is found in a range condition check during an assignment, the attribute to the left of the assignment operator will be assigned the computed value, although it is out of bounds.
- If the error is 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 `choice`, an error message is given and the operation is performed anyhow.
- If a Null pointer (Own and ORef) is de-referenced, a 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.

# *UML Import and Export*

The chapters in this section describe Tau's capabilities for importing and exporting data in external formats from and to a UML model. This includes features for information exchange with other modeling tools.

**See also**

Adding Importers to learn how to add custom importers to Tau.

# 14

# *.NET Assembly Importer*

The .NET Assembly importer is a tool for importing a .NET assembly or a COM component into a UML model. After the assembly or component has been imported into the model its contents can be visualized and it can be accessed from the UML model.

In this document we use the term **component** both for .NET assemblies and COM components since they are handled in almost the same way by the importer.

The .NET Assembly Importer is available on Windows platforms only.

# Operation Principles

The main application area of the .NET Assembly Importer is to create a UML representation of a component. This representation serve mainly two different purposes:

- Visualization. By importing a component its contents can be visualized in diagrams and Model View.
- Access. The imported definitions from the component can be accessed and used in the UML model.

When importing a component only the definitions it exposes will be imported. The implementation of the component is not imported to the model. All imported definitions from a component are put in one or sometimes many UML packages. In addition a file artifact will be created that represents the imported component in the model.

## Import a Component

To import a component into UML:

- Select the **Model** item in Model View.
- Open the Import Wizard (**File** menu, **Import...** command).
- Select **Import .NET assembly or COM component** in the dialog.

The wizard then displays the following dialog allowing you to specify which components to import:

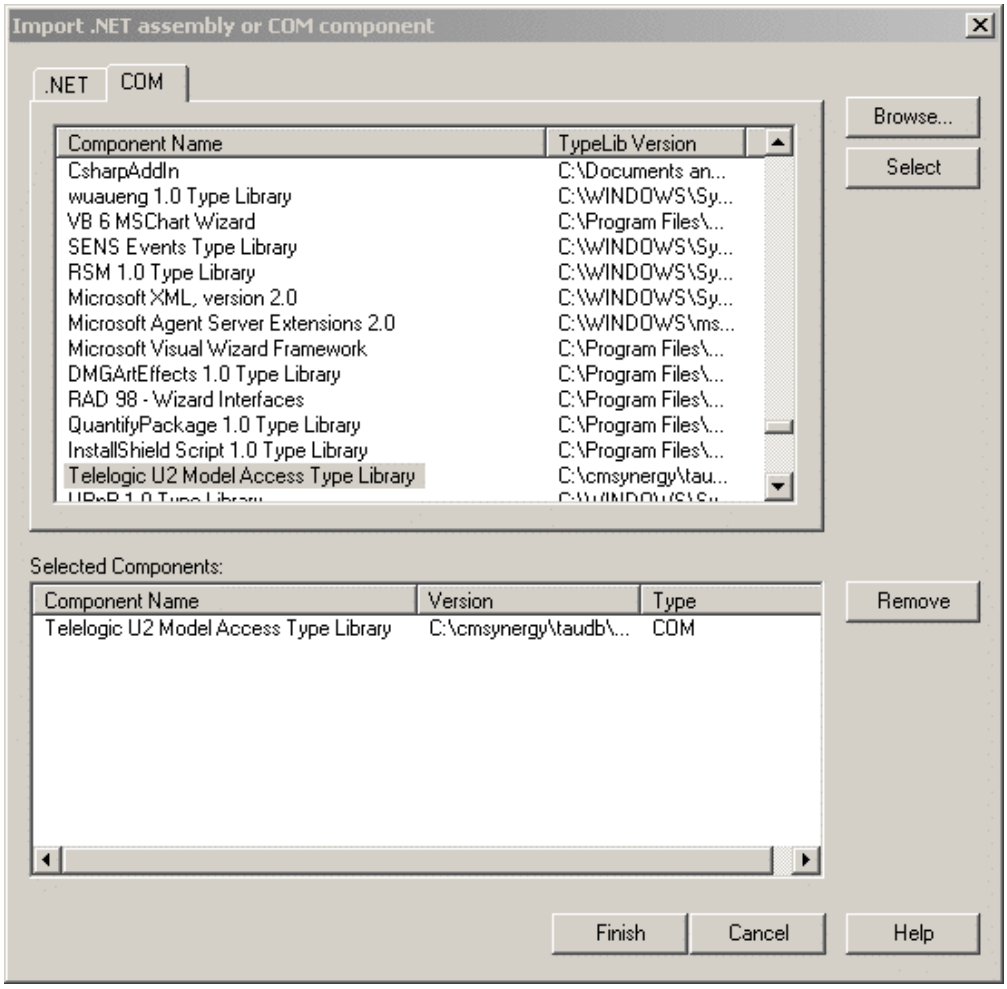



*Figure 168: The Import .NET assembly or COM component wizard*

The .NET tab of the dialog lists all .NET assemblies that are present in the Global Assembly Cache (GAC) on the computer. For each assembly its version number is also shown.

The COM tab of the dialog lists all COM components that are registered in the system registry on the computer. For each COM component the file that contains its type library is also shown.

In case the component you want to import is not registered in the GAC, nor in the system registry, it won't be present in any of the lists. In that case you can use the **Browse...** button to locate the component in the file system.

You may select any number of .NET assemblies or COM components to be imported. All components that are listed in the bottom list will be imported when the **Finish** button is pressed.

### Reimport a Component

Normally a component is only imported once into the model. However, there are situations when you might want to repeat the import at a later point in time for the same component. For example, if changes have been made in the component a repeated import can be used in order to update the model with these changes.

To update the model by reimporting a component select the command **Import component** that is available in the context menu on the file artifact that represents the component in the model.

# Translation Rules

This section describes the translation rules that are used when importing a component into the UML model. The terminology used here is oriented towards .NET assemblies rather than COM components. The reason is that a COM component that is imported will first be converted to a .NET assembly, using the standard type library importer tool that is part of the .NET framework. This interop assembly is then translated to UML.

### Assembly and Namespace

Each namespace is translated to a UML package with the same name as the namespace.

The assembly itself also defines a top-level scope for the types it contains. Hence the assembly will also be translated to a UML package with the same name as the assembly.

### Class

A class is translated to a UML class with the same name as the class.

If the class inherits from another class, or implements interfaces, the UML class will contain corresponding generalizations.

Fields of the class are translated to corresponding attributes of the UML class.

## Interface

An interface is translated to a UML interface with the same name as the interface.

If the interface inherits from other interfaces, the UML interface will contain corresponding generalizations.

## Method

A method of a class or interface is translated to a UML operation contained in the corresponding UML class or interface. Parameters of the method are translated to parameters of the UML operation. The names of both the imported method and parameters are the same as the originals.

## Enumeration

An enumeration is translated to a UML datatype with the same name as the enumeration. The literals of the enumeration are translated to corresponding literals of the UML datatype.

# 15

# *C/C++ Import*

This section is a reference and guide to the C/C++ import functionality, which takes a set of C/C++ header files, preprocesses them, and translates the declarations in these headers into corresponding UML representations.

The C/C++ importer can also be instructed to import statements that are present in inline implementations of imported functions, making it a useful tool when migrating C/C++ header files to UML.

This document describes the rules for this translation process, and the operation principles for the C/C++ import.

# Operation Principles

The main application area of the C/C++ import is to provide access to external C/C++ code from UML applications developed with the UML tool set. This is achieved by parsing a set of C/C++ header files, checking their syntax and semantics, and then translating the C/C++ definitions to their corresponding UML definitions according to a fixed set of C/C++ to UML translation rules.

Another application area is to produce a graphical documentation of legacy code, aiming at helping you understanding its design, structure, dependencies and inheritance tree.

Yet another application area is to use C/C++ import as a means for migrating an existing C/C++ application to UML, with the intention to continue its development in UML and to generate new C/C++ code with the Tau code generators. Such a migration does not necessarily have to be done for every part of an application. A common scenario is to only generate header files from the imported UML model, while still keeping and maintaining existing implementation files in C/C++. The C/C++ importer facilitates this migration scenario through an option for also importing action code (statements) that are present in inline implementations of header files. There is also an option for letting the C/C++ importer create a set of header file artifacts, and a build artifact, that are appropriate for regenerating the header files using the C++ Application Generator.

### Target UML package

Each import of C/C++ will result in a number of UML elements that are stored in the package that is designated as target. Even though assigning any package as target is possible, for example the package containing your UML model under development, there may be advantages to assign an empty package to start with, for instance to gain more visibility and understanding of what has actually been imported.

### Input

You may specify any set of input C or C++ header files as input to an importing scheme. The files that are specified as input are preprocessed according to your preferences, before the actual translation is done.

### Preprocessor

The C/C++ import is designed to take advantage of the conditional compiling and other directives provided by a C/C++ preprocessor, in order to allow a flexible importing scheme, supporting the fundamental idea about how the code is intended to be compiled. The C/C++ mode of the Import Wizard is designed in a way that allows you to specify any preprocessor, and any preprocessor options. Some Preprocessor restrictions may apply.

**Note**

*Importing a C/C++ header file consists of two steps, preprocessing and import. During preprocessing the header file is preprocessed. It is possible to specify your own preprocessor, and any preprocessor options.*
*During import to UML all macros and other preprocessor constructs are expanded in the imported (preprocessed) file. Original macro names and definitions are not saved and are not imported to UML.*

### Syntax and semantic checks

After the C/C++ code is preprocessed, the necessary syntactic and semantic checks are done to verify the completeness of the imported header files before translation to UML. These checks encompass the subset of the C and C++ languages that is supported by the C/C++ import.

The imported C/C++ code must always be fully syntactically correct. If syntax errors are detected these are reported, and the import will not proceed.

During import, a limited subset of semantic checks are performed. However, the C/C++ Importer is not a replacement for a C/C++ compiler, and it is usually possible to import a program even if it contains several semantic errors. If you are uncertain about the correctness of the code that is imported, it is recommended to use a C/C++ compiler to verify the correctness before doing the import.

### C/C++ to UML translation rules

The translation of each C/C++ construct or topic that is possible to import is described in a subsection of its own. See General Translation Rules and the following sections.

Most of the language constructs are illustrated with examples for how imported C/C++ definitions are translated to UML.

**Note**

*The purpose of the examples is to illustrate a particular translation rule only. Therefore details in the translated UML not directly related to that translation rule may be omitted for the sake of brevity. Also, keep in mind that exactly how the translated UML will look like often depends on which options that have been used in the import.*

### Created UML definitions

The generated UML definitions are put in the UML package that is designated as "target" for the import. After the package is included in a project, you can then refer to these UML definitions from the UML model, which allows full syntactic and semantic analysis.

You may have multiple packages, each of them functioning as target container for a subset of all definitions. This allows for instance to separate C and C++ definitions, and also to separate code of various origins that has been developed using different flavors of C and C++. For each such import, you may have different settings for preprocessor, language (C or C++), and the C++ dialect that is used.

### Trace back to source headers

The importing scheme keeps track of the location of entities in the source header files so that you can easily look up any imported entity in your C/C++ programming environment or any text editor. Navigation from Tau is done by means of the **Go to source** command that is available in the context menu of an imported entity.

**Example 144: Navigate to source headers from an imported entity**

Assume the following C++ file test.h is imported to UML:

```
class C {};
```

After the import of this file you may right-click on the class C in the import package and select **Go to source**. A submenu will list all occurrences of class C in the imported header files, in this case "test.h: 1".

### Compiler and language support

The translation rules are designed to support both C and C++ target compilers, using any supported C/C++ dialect. To a large extent, the translation rules are independent of whether a C or C++ target compiler is used. However, when C/C++ Import executes in "C mode", a few translation rules are slightly modified.

These modifications are described in the section "Translation rules for C compilers" on page 617.

## Import C/C++

To import C/C++ files into UML:

- Select the **Model** item in Model View.
- Open the Import Wizard (**File** menu, **Import...** command).
- Select **Import C/C++** in the dialog.

Header files and options are specified in the `cppImportSpecification` stereotype, applied to a package. Imported definitions will be placed into this package stereotyped by `cppImportSpecification`. The Import wizard will automatically apply this stereotype, and store all settings made in the wizard using tagged values. You may later use the Properties Editor for inspecting or changing these settings, or add more advanced settings that are not available from the Import wizard. See Specify import options for more information.

### Specify import input

The first page of the C/C++ import wizard lets you specify the input files. Simply browse and select the files to import.

**Note**

*Under Windows you must choose to either select a set of C/C++ files to import, or to select a single Visual Studio project file (.vcproj). In the latter case you must also select a configuration defined in that project file. The preprocessor settings of the selected configuration will be used in the import.*

### Specify import settings

The second page of the C/C++ import wizard lets you set some common options for the import. You can specify the following:

- ***C/C++ dialect.*** The default dialect (Standard C/C++) conforms to the ISO/IEC 14882 standard of the C++ language, and the ANSI/ISO 9899-1989 standard of the C language. Other dialects that are supported are GNU, Borland and Microsoft. For more information see C/C++ dialect.
- ***Action code import.*** This option controls how the importer handles function bodies that are present in the imported files. By default such bodies are not imported to the UML model. If you want to import them you can either import them to UML actions, or you can import them as informal actions. The option to import action code is typically used when the purpose of the import is to migrate a legacy C/C++ application to UML, and later regenerate it using the C++ Application Generator. However, it may of course also be used if the purpose of the import is to visualize a C/C++ program in UML. See Action code strategy for more information.
- ***Source files contain only C.*** By default the importer assumes that the imported header files contain C/C++. If you are importing plain C header files (i.e. no use of C++) then you should set this option. The importer then operates in “C mode”. For more information see C only.
- ***Import only exported definitions.*** Some compilers support the `__declspec(dllexport)` construct as a means for specifying those definitions that are exported from a dynamic link library or shared object. By using this option it is possible to only import such definitions. This can be useful when only the exported interface of a library shall be imported to UML. For more information see Import only exported definitions.
- ***Generate GUIDs based on scoped name.*** By default imported entities will get random GUIDs. If you intend to do repeated imports it is more appropriate to use the option to generate GUIDs that are based on the scoped names of the imported definitions. See GUID name option for more information.
- ***Generate artifacts.*** If this option is enabled the importer will generate file artifacts that represent the imported files. It will also generate a build artifact for the C++ Application Generator, which will manifest the file artifacts. The purpose of these artifacts is to facilitate the regeneration of the imported files using the C++ Application Generator. See Generate artifacts for more information.

- ***Import now.*** Turning this option off before clicking **Finish** in the **Import Wizard** disables the actual import and allows you to specify the import options before the C/C++ import is launched (usually the first time the C/C++ import is called). See Specify import options for how to specify these import options. When appropriate import options have been set, use the **Import C/C++** command (available in the context menu of the import package) to perform the import.

  Even if you leave the "Import now" option enabled in the wizard, you will need to use the **Import C/C++** command if you want to repeat the same import operation at a later point in time (for example because changes have been made in the imported files). Note that such repeated imports require some special attention; see Repeated Import Considerations for more information.

### Specify u2 file for package ImportedDefinitions

The third page of the C/C++ import wizard allows you to specify a model (`.u2`) file where to save the result package of the import. When a package is saved in a model (`.u2`) file any imported header file paths become relative to this model file. The flag "**Use absolute paths for input header files**" should be set if the paths should not be relative to the model file.

### Import output

The result of the import will be placed in a new package located in the context of your current Model View selection, named "ImportedDefinitions". You may want to move the generated definitions to a "source" package, tidy up the diagrams and add comments.

### Specify import options

The C/C++ import options are edited by opening the Properties Editor using the `cppImportSpecification` filter (stereotype).

1. Open the Properties dialog for the imported package. In the Filter drop-down menu select cppImportSpecification.
2. Adjust the available options to adequate values.

### Specify import input in C/C++ import options

You may use a wildcard notation in the **input header files** field in cppImportSpecification filter (stereotype). Thus you can write for example `e:\test\*.h` or `e:\test\*` in this field and matching file names will be marked for import.

## Manual C/C++ Import

This step is normally not necessary to perform. The necessary Add-Ins are loaded when the **Import Wizard** is used.

1. From the **Tools** menu select Customize.
2. Click the add-ins tab in the dialog to list the currently available add-in modules.
3. In the list of available add-ins, select the **CppImport** add-in and then close the dialog.

A UML profile package named **TTDCppImport** is now loaded and appears in the Library node in the Model View. This package contains the cppImportSpecification stereotype and its attributes with the settings for the C/C++ import.

You may then apply this stereotype to a package, and set appropriate import options by editing the tagged values of that stereotype. The import operation is performed by right-clicking on the package in the Model View, and selecting the **Import C/C++** command.

# Repeated Import Considerations

Normally a C/C++ import is an operation that only is performed once for a set of C/C++ files. However, there are situations when you might want to repeat the import at a later point in time for the same set of files. For example, if changes have been made in the input files a repeated import can be used in order to update the model with these changes.

**Important!**

*When a repeated import is performed from the context of some import package, all existing imported UML entities in that package will first be deleted before the package is populated with new imported entities. Any information added to these imported entities in the UML model will hence be lost. It is not possible to perform an "Undo" of a repeated C/C++ import, so be careful.*

## GUID name option

UML entities that result from a translation like the C/C++ import are by default given randomly generated GUIDs.

Importing C/C++ definitions repeatedly implies that references to the entities (typically from presentation elements) become unresolved the next time an import is made. To cope with this, the option GUID algorithm can be set to `Based on scoped name`, in order to have each GUID generated from the (fully qualified) name in the original C/C++ definition.

**Important!**

*When using a GUID algorithm based on scoped names it is very important that the imported header files are semantically correct. If the files are incorrect, so that for example a C++ scope contains two definitions with the same name, then two UML entities will get the same GUID. After saving the model, it can therefore not be loaded again due to a GUID conflict. Because of this, always save the import package in a separate .u2 file if you want to do repeated imports. Should a GUID conflict arise you can then just remove the import package .u2 file from your project and still load the remaining parts of your model.*

For some entities the "scoped name" is computed according to special rules, described next. The purpose of these rules is to guarantee the uniqueness of generated GUIDs. These rules are applicable to Generalization, Return parameter, Formal parameter and Operation.

### Generalization

For a generalization, the GUID becomes “X-inherits-Y”, where

- X is the name of the subtype (qualified relative the package into which the import is made)
- Y is the name of the supertype.

### Return parameter

For a return parameter, the GUID becomes “X-return-parameter”, where X is the qualified signature of the operation to which the return parameter belongs.

### Formal parameter

For a formal parameter, the GUID becomes “X-Y”, where

- X is the qualified signature of the operation to which the parameter belongs.
- Y is “`fparN`”, where ‘N’ denotes the ordinal number of the parameter.

### Attribute

For an attribute, the GUID becomes the qualified name of the attribute prefixed with “`attr--`”.

### Operation

For an operation, the GUID becomes the qualified signature of the operation.

# General Translation Rules

### External

By default imported definitions are marked as external in UML. Thereby it is indicated that these definitions correspond to C/C++ definitions that are external to the UML model.

There is an option “Set “External” attributes for imported definitions” in the cppImportSpecification stereotype that can be turned off in order to not mark imported definitions as external. This can for example be useful if the C/C++ import is done with the purpose of migrating legacy code to UML. With this option turned off the external property will only be set if the C/C++ definition is explicitly declared to be `extern`. This is done since for some kinds of definitions (for example constants) the translation rules from UML to C++ are different for external and non-external definitions.

In the translation rule examples in the following sections this option is assumed to be set. That is, the UML definitions are normally not marked as external in the examples for brevity reasons.

# Names

A C/C++ identifier is given the same name in UML.

When a C/C++ name is a UML keyword, it will be quoted with apostrophes in order to create a valid UML name.

**Example 145: Translation of names**

C/C++:

```
void OpenFile();
double signal; // UML keyword "signal"
typedef int part; // UML keyword "part"
part port; // UML keywords "part" and "port"
```

UML:

```
public void OpenFile();
public double 'signal';
public syntype 'part' = int;
public 'part' 'port';
```

# Fundamental Types

A fundamental C/C++ type is mapped to a UML type with the same name.

UML representations of all fundamental C/C++ types are available in the TTDCppPredefined profile package, which is loaded with the CppTypes add-in.

During C/C++ import it is assumed that this package is available, and references to the types in that package will be generated. Should the package not be available at the time when the import takes place, a warning message is issued to inform that a large number of unbound references as the result of the translation can be expected.

## Translation from C/C++ fundamental types to UML

The table below shows the mapping between fundamental C/C++ types and UML types. In most cases, a type in the TTDCppPredefined profile package is a syntype of a predefined UML type.

| C/C++ Fundamental Type | UML Type | Predefined UML Type |
|---|---|---|
| `signed int,`<br>`int` | `int` | `Integer` |
| `unsigned int,`<br>`unsigned` | `'unsigned int'` | `Integer` |
| `signed long int,`<br>`signed long,`<br>`long int,`<br>`long` | `'long int'` | `Integer` |
| `unsigned long int,`<br>`unsigned long` | `'unsigned long int'` | `Integer` |
| `signed short int,`<br>`signed short,`<br>`short int,`<br>`short` | `'short int'` | `Integer` |
| `unsigned short int,`<br>`unsigned short` | `'unsigned short int'` | `Integer` |

| C/C++ Fundamental Type | UML Type | Predefined UML Type |
|---|---|---|
| `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'` | `Character` |
| `wchar_t` | `'wchar_t'` | `Character` |
| `float` | `float` | `Real` |
| `double, long double` | `double` | `Real` |
| `bool` | `bool` | `Boolean` |
| `void` | `N/A` | `N/A` |

**Note**

*The special* `void` *type is not represented explicitly in UML. Instead this type is translated by omitting input and result arguments to operations as described in the section about translation of Function.*

# Pointer, Array and Reference Type

The pointer (`*`) and array (`[ ]`) type specifiers of C/C++, and the reference (`&`) type specifier of C++, make it possible to create the following:

- Pointer type specifier,
- Array type specifier,
- Reference type specifier,
- No type specifier,

and combinations of these.

## Pointer type specifier

A pointer type specifier is translated to a template instantiation of the `CPtr` template of the TTDCppPredefined profile package. The `CPtr` template has operations corresponding to the operations that can be performed on a C/C++ pointer.

**Example 146: Translation of pointers**

C/C++:

```
typedef int* p_int;
extern void* p_userdata;
typedef char** pp_char;
```

UML:

```
public syntype p_int = CPtr<int>;
public 'void*' extern p_userdata;
public syntype pp_char = CPtr<'char*'>;
```

### Untyped pointers

Untyped pointers (`void*`) are translated to a special UML type in the TTDCppPredefined profile package, called `'void*'`. It is defined as follows:

```
syntype 'void*' = CPtr<Any>;
```

Operations for CPtr<> like `SetValue` and `GetValue` should not be used with `'void*'`, because `Any` means nothing and you can not directly define values of the type `Any`.

The `'void*'` type can only be used as a conversion buffer between C++ `void*` and UML values. In UML `'void*'` can be converted to `CPtr<>` types and UML class references and back without any special cast. For all other UML types a conversion between `'void*'` and a UML type can be fulfilled only by an explicit call of the `cast<>` operation.

**Note**

*If you want to generate code, you must always insert explicit cast operators for `'void*'` conversion! You will not get any error messages when checking your UML model if cast is not used, but the generated code will not compile.*

**Example 147: Translation and usage of C++ void* type**

C/C++:

```
typedef void* voidstar;
typedef struct Str {
  voidstar  ptr;
} Str;
```

Imported UML:

```
public syntype voidstar = 'void*';
public <<struct>> class Str {
  public voidstar ptr;
}
```

`'void*'` usage in UML:

```
part Str x;  CPtr<int> pi;
pi = new CPtr<int>();
pi.SetValue(10);
x.ptr = cast<'void*'>( pi );     // 'void*' = CPtr<int>
pi = cast<CPtr<int> >( x.ptr ); // CPtr<int> = 'void*'

class C {}
CPtr<C> pc;
C c = new C();
x.ptr = cast<voidstar>( c );     // 'void*' = class
reference
pc = cast<CPtr<C> >( x.ptr );    // CPtr<C> = 'void*'
c = pc;                          // class reference =
CPtr<C>
```

Example 147 on page 559 illustrates the following rules of `'void*'` usage in UML:

- There are no memory allocation and deallocation functions for `'void*'` in UML. One way to initialize `'void*'` in UML is to create an object of a class (`new C()`) or initialize a `CPtr<>` type (`new CPtr<int>()` ) and then assign it to `'void*'` with use of `cast`.
- If a`'void*'` variable in UML should point to some UML object with value semantics (for example, `int`), `CPtr<type>` should be used to create a pointer.
- Note the UML syntax for actual template arguments that are template instantiations: `cast<CPtr<int> >`. The syntax requires a blank after the first closing bracket. For increased readability you may define syntypes of the `CPtr<>` types.

- `cast<CPtr<Any> >` can **not** be used! Only `cast <'void*'>` can be used!
- `cast` operation can use direct type name ( `cast<'void*'>` ) or any of its synonyms as defined with syntypes ( `cast<voidstar>` ).
- Casting '`void*`' to a class reference will not work in the generated C code - this is a restriction on the C Code Generator - so if such a conversion is needed, '`void*`' should be explicitly cast to `CPtr<class>`, then `CPtr<class>` can be implicitly cast to class reference.

'`void*`' can be initialized by calling the UML `new` operator. It can also be initialized by a `GetAddress` call.

**Example 148: void* initialization in UML**

C/C++:

```
typedef void* voidstar;
```

Imported UML:

```
public syntype voidstar = 'void*';
```

`void*` initialization in UML:

```
'void*' pv1, pv2;  CPtr<int> pi;  int i;
pi = new CPtr<int>();
pi.SetValue(10);
pv1 = cast<'void*'>( pi );       // 'void*' = CPtr<int>

i = 12;
pi = CPtr<int>::GetAddress( i );
pv2 = cast< voidstar >( pi );    // 'void*' = CPtr<int>
```

UML`'void*'` values can be passed to a C++ function taking `void*`. C++ `void*` values returned by C++ functions can be stored in UML`'void*'` values.

**Example 149: Calling C++ functions from UML**

C/C++:

```
typedef void* MyVoidStar;
MyVoidStar init( void );
void finit( MyVoidStar );
```

Imported UML:

```
public syntype MyVoidStar = 'void*';
public MyVoidStar init();
public void finit( MyVoidStar );
```

Calling C++ functions from UML:

```
CPtr<int> pi;
pi = cast< CPtr<int> > ( init() );
finit( cast<MyVoidStar>( pi ) );
```

---

### Pointer to char

Pointers to char (`char*, wchar_t*`) are also mapped to a special UML type from the TTDCppPredefined profile package, called '`char*`' and '`wchar_t*`'. These types are defined in the following way:

```
<<External="true">> datatype 'TCHAR*' : CPtr<TCHAR> {
    public 'TCHAR*'( Charstring str);
    public <<External="true">> Charstring ToString();
}
<<External="true">> syntype TCHAR = Character;
syntype 'char*' = 'TCHAR*';
syntype 'wchar_t*' = 'TCHAR*';
```

The UML type 'TCHAR*' inherits all operations defined for the `CPtr` template, and adds two new operations - a constructor from the UML type `Charstring` and a conversion operation from 'TCHAR*' to `Charstring`. These operations are intended for conversion between UML character strings and C++ char* strings.

There is an option “Import char* to CPtr<char>” in the cppImportSpecification stereotype that can be turned on if you prefer to treat pointers to char as pointers in general. That could be of interest if the conversion facilities to and from UML Charstrings are not needed.

## Array type specifier

An array type specifier is translated to a template instantiation of the `CArray` template of the TTDCppPredefined profile package.

The reason why a special `CArray` template is used instead of using the ordinary Multiplicity specification of UML is that `"[]"` is a type specifier in C/C++, while in UML it is a specifier on a structural feature. If multiplicity

had been used, it would not have been possible to translate `typedef`, since no structural feature is present in its translation, as shown in Example 150 on page 562.

**Example 150: Translation of arrays**

C/C++:

```
extern char c_arr1[20];
typedef int array_of_ints[1024];
extern char c_arr2[];
```

UML:

```
public CArray<char, 20> extern c_arr1;
public syntype array_of_ints = CArray<int, 1024>;
public 'char*' extern c_arr2;
```

### Unbounded array

Array specifiers that do not specify the size of the array (also referred to as "unbounded arrays") are translated in the same way as a pointer specifier. For an example see Example 150 on page 562.

**See also**

Pointer type specifier

## Reference type specifier

Reference type specifiers do not need to be translated, since they are implicit in UML, that is the default is the reference semantics for a UML type.

**Example 151: Translation of references**

C/C++:

```
extern int i;
/* i is initialized elsewhere */
extern int& r;
/* r is initialized to i, elsewhere (int& r = i;) */
```

UML:

```
public int extern i;
```

```
public int extern r;
```

References could also appear as specifiers for formal function arguments, and return types. The translation of the reference type specifier in that context is described in "Arguments and return type" on page 569.

References used on return types of a function gets the `<<CppReference>>` stereotype applied on the corresponding operation return parameter. See Example 160 on page 571 for an example.

## No type specifier

- If a type, other than a predefined C/C++ type, has no type specifier at all, then UML structural features that are typed by the corresponding UML type become parts.
- If the UML type is used for typing entities that are not structural features (for example a syntype), the lack of type specifiers does not lead to any special translation.
- In the case of a syntype, structural features typed by it will become parts, and the rule is applied recursively throughout the "syntype chain". The reason for this rule is that the default is the reference semantics in UML.

**Example 152: Translation of types without type specifiers**

C/C++:

```
class MyClass {};
MyClass value_var; // Value semantics
extern MyClass& ref_var; // Reference semantics
```

UML:

```
public class MyClass {};
public part MyClass value_var;
public MyClass extern ref_var;
```

### Predefined type

If the type is a predefined type, the default is the value semantics (as in C/C++). In that case, structural features typed by the type will be translated as usual.

# Enumerated Types

An enumerated type is translated to a datatype with literals corresponding to the `enum` literals.

In case the enumerated type has no literals, it can be treated as an integer in C/C++, and is consequently translated to a syntype of `int`.

**Example 153: Translation of enumerated types**

C/C++:

```
enum {} v;
enum E2 {}; // Enum without literals
enum E3 {a, b = 10, c = b + 5};
```

UML:

```
public syntype incomplete_v = int;
public incomplete_v v;
public syntype E2 = int;
public enum E3 { a, b = 10, c = [[b + 5]]}
```

For more information about the translation of constant expressions, that may be specified for C/C++ literals, see Constant expression.

**See also**

"Implicit conversions from int to enum" on page 623

# Typedef

A `typedef` is translated to a UML syntype.

**Example 154: Translation of a type definition in C**

C/C++:

```
typedef int MyInt;
typedef struct r {
  int a;
} r; // Typedef name is the same as the tag name!
typedef struct q {
  bool m_bShall;
} *q; // Typedef name is the same as the tag name,
      // and there is a type specifier.
```

```
typedef struct s {
  MyInt a;
}; // Omitted typedef name - legal but rare!
typedef void myvoid;
typedef myvoid myvoid2;
myvoid f(myvoid2);
```

UML:

```
public syntype MyInt = int;
public <<struct>> class r {
  public int a;
}
public <<struct>> class q {
  public bool m_bShall;
}
public <<struct>> class s {
  public MyInt a;
}
public void f();
```

The translation of `q` above will yield a warning (“Conflicting typedef name”) during import and the typedef will not be translated. For more information see Typedef declaration of tagged types.

Note also that a typedef of void is not translated, but usage of such a typedef name is translated as if void had been used instead. See Typedef with void type for more information.

---

## Typedef declaration of tagged types

For `typedef` declarations of a tagged type, where the `typedef` name is the same as the name of the tag, and where the `typedef` type is not a forward declaration without definition, the following rule applies:

If the `typedef` type of such a `typedef` has type specifiers, it does not define a new type name, and hence no syntype is generated. The reason for not generating such a syntype is that there would otherwise be a naming conflict if it is translated according to the normal rules for Pointer, Array and Reference Type.

## Typedef without name

A `typedef` declaration where the `typedef` name has been omitted, does not define a new type name, and hence no syntype is generated.

## Typedef with void type

A `typedef` where the `typedef` type is void, or a `typedef` of void does not define a new type name, and hence no syntype is generated, but the `typedef` name is remembered.

References to the `typedef` name will then be translated in the same way as void would have been translated in that context.

# Function

### General, function prototype

A function prototype is translated to a UML operation.This rule is valid for both member and non-member functions.

**Example 155: Translation of functions in general**

C/C++:

```
char myfunc1(char);
int myfunc2(void);
void myfunc3();
void myfunc4(int);
bool myfunc5(void* p1, double p2);
```

UML:

```
public char myfunc1(char);
public int myfunc2();
public void myfunc3();
public void myfunc4(int);
public bool myfunc5 ('void*' p1, double p2);
```

## Non-member function

Non-member functions will result in operations in the package into which the import is made, while member functions will be put as operations of a class or choice.

## Member function

Although the translation rules as such are identical as for non-member functions, a C++ member function may be “richer in features” in its declaration. The translation rules for such member-specific features of a function are described in the relevant subsections of the section describing the translation of Class, Struct and Union.

## Formal arguments

A formal argument of the function is translated to a corresponding formal parameter of the UML operation, unless it is of void type in which case it will not be translated.

## Return type

The return type of the function is translated to a return type parameter of the UML operation.

## Function declaration without prototype

A function declaration without a prototype (old-style) is translated as an ordinary function with a prototype, provided that the arguments of the function are properly declared.

Function declarations without a prototype are supported by the C/C++ import, but only if declarations of all function arguments are present.

**Example 156: Translation of function without prototype**

C/C++:

```
float average(x,y,z)
float x,y;
char z;
{
  return 1.1;
}
```

UML:

```
public float average(float x, float y, char z);
```

## Overloaded functions

A set of overloaded functions are translated to a corresponding set of overloaded UML operations.

**Example 157: Translation of overloaded functions**

C/C++:

```
int f0();
int f0(double);
int f1(int);
int f1(const int);
```

UML:

```
public int f0();
public int f0(double);
public int f1(int);
```

Overloading on const is not supported in UML, but it is allowed in C++. The reason is that const is a part of a type in C++ and a const type is distinguished from the same type without const. In UML const is not a property of the type and name resolution does not take it into account while resolving calls of overloaded operations.

Overloading by using const as a specifier on a member function can also lead to that not all overloaded versions of the function can be imported to UML. See the example below.

**Example 158: Overloading on const**

The following C++ declaration will not be fully imported to UML:

C/C++:

```
class X {
   int func( int x );
   int func( int x ) const;
};
```

Output:

```
Ignored conflicting declaration 'func'
```

UML:

```
public class X {
  private <<IsQuery = "true">> int func( int x );
```

```
}
```

**See also**

"Ambiguities between overloaded functions" on page 572

## Arguments and return type

In UML by default function arguments are passed by reference. The exception is datatypes which follow value semantics. In C++ function arguments have value semantics by default.

Due to these difference, the rules in the table below are applied. In the examples, a type with the name 'D' is used for C++ types which are translated to UML datatypes (for example, simple types, pointer, array, enum). Types named 'C' denote all other possible C++ types.

| Rule/example | C++ | UML |
|---|---|---|
| 1) C++ argument of type D is translated to UML argument without direction or part specifiers | `void F(D);` | `void F(D);` |
| 2) C++ argument of type C is translated to UML part argument | `void F(C);` | `void F (part C);` |
| 3) C++ argument of type D passed by reference (&) is translated to UML inout argument | `void F(D&);` | `void F (inout D);` |

| Rule/example | C++ | UML |
|---|---|---|
| 4) C++ argument of type C passed by reference (&) is translated to UML part argument with inout direction | `void F(C&);` | `void F (inout part C);` |
| 5) C++ pointer to D type passed by reference (&) is translated to UML inout argument of type CPtr <D> | `void F(D*&);` | `void F (inout CPtr<D>);` |
| 6) C++ pointer to C type passed by reference (&) is translated to UML inout argument | `void F(C*&);` | `void F(inout C);` |

The import for C++ reference arguments is aligned with the following UML to C++ translation rules:

- A UML reference is translated to a C++ pointer
- An inout parameter is translated to a C++ reference parameter
- Part arguments are translated to common C++ arguments (part is ignored, only the name of the type is printed)
- A datatype is not a UML reference so it is mapped in the same way as part arguments

For more details see Type of typed definitions in the C++ Application Generator documentation.

**Example 159: Translation of formal arguments**

C/C++:

```
class C {
    int x;
};
typedef C* pC;

void F1( int x );
void F2( int& x );
void F3( int*& x );
void F4( int**& x );

void F5( C x );
void F6( C& x );
void F7( C*& x );
```

```
void F8( pC x );
void F9( pC& x );
void F10( pC*& x );
```

UML:

```
public class C {
  private int x;
}
syntype pC = CPtr<C>;

void F1( int x );
void F2( inout int x );
void F3( inout CPtr<int> x );
void F4( inout CPtr<CPtr<int> > x );

void F5( part C x );
void F6( inout part C x );
void F7( inout C x );

void F8( pC x );
void F9( inout pC x );
void F10( inout CPtr<pC> x );
```

Constant arguments are translated to read-only UML operation parameters.

**Example 160: Translation of function arguments and return value**

C/C++:

```
int f1 (int p1, int& p2, const int& p3, const int* p4,
int *const p5);
int& f2();
const int& f3();
class MyClass {};
void f4(MyClass& p1, MyClass p3, MyClass* p2);
```

UML:

```
public int f1(int p1, inout int p2, const inout int p3,
const CPtr<int> p4, CPtr<int> p5);
public (<<CppReference>> int) f2();
public (<<CppReference>> int) f3();
public class MyClass {}
public void f4 (inout part MyClass p1, part MyClass p3,
MyClass p2);
```

Note that a C++ function that returns a pointer to a constant type, for example `const char*`, will in UML be translated to an operation that returns the type without const specifier. This is normally not a problem at UML level, but might become an issue when generating C or C++ code that uses that function. See Pointer to constant for more information.

## Default argument

A function argument with a default value specified is translated to a UML operation parameter with a default value. If the default value is a constant expression it is translated as specified in Constant expression.

**Example 161: Translation of functions with default arguments**

C/C++:

```
int func(int a, int b = 5, int c = 7);
int func(int a); // Ambiguous function!
```

UML:

```
public int func(int a, int b = 5, int c = 7);
public int func(int a);
```

Note that a call of `func` with just one actual argument will be ambiguous both in C++ and UML. See Ambiguities between overloaded functions for more information.

---

## Ambiguities between overloaded functions

In C++, ambiguities between overloaded functions is allowed, provided that the functions are never called with a set of actual arguments that make the call unresolvable.

The same is true for UML. The second version of `func` in Example 161 on page 572 is hence acceptable as a definition in UML, but it cannot be called.

## Unspecified argument

Functions with unspecified arguments (also referred to as Ellipsis function) are not supported. If such a function is encountered by the C/C++ Importer the ellipsis arguments (...) will be ignored.

A workaround is to define a set of wrapper functions in a C/C++ header that is imported to UML. These wrapper functions specify the versions of the ellipsis function that should be used in the UML model.

**Example 162: Translation of ellipsis functions**

C/C++:

```
int printf(const char *, ...);

/* Wrapper functions */
int printf_int(const char* s, int a) { return printf(s,
a); }
int printf_str(const char* s, const char* a) { return
printf(s, a); }
```

UML:

```
public int printf(const 'char*');
public int printf_int(const 'char*' s, int a);
public int printf_str(const 'char*' s, const 'char*' a);
```

## Inline function

A function that is declared to be inline is translated as an ordinary function.

The inline keyword on functions could be seen as a directive to the C++ compiler, which only affects the way that calls to these functions are generated.

**Note**

*The "inline" property will be kept using the inline stereotype from the TTD-CppPredefined profile.*

**Example 163: Translation of inline functions**

C/C++:

```
inline int fac(int n){return (n == 1) ? 1 : n * fac(n - 1);};
```

UML:

```
public <<inline>> int fac ( int n);
```

# Function pointer

Function pointers are mapped to UML interfaces with the «operationReference» stereotype. The name of the interface will be the type definition name. If a function pointer type is used in inline code (without a typedef), then the name of the interface will be on the form "fpointer_<formal parameter type names>_returns_<return parameter type name>". The resulting interface will only contain one operation named "call".

Calling a function through a function pointer is in UML done with

```
i = myOp.call( 10 );
```

where myOp is an attribute typed by the generated function pointer interface.

**Example 164: Translation of function pointers**

C/C++:

```
typedef int (*PFunc) ( int );
void Func ( bool (*pointer) ( bool ) );
int (* qq) ( int );
int (* Func1( bool ) ) ( int );
```

UML:

```
public <<operationReference>> interface PFunc {
  int call( int);
}
public void Func( fpointer_bool_returns_bool pointer );
<<operationReference>> interface fpointer_bool_returns_bool {
  bool call( bool);
}
public fpointer_int_returns_int extern qq;
public fpointer_int_returns_int Func1( bool );
<<operationReference>> interface fpointer_int_returns_int {
  int call( int);
}
```

## Function type

An alternative way of declaring function pointers in C/C++ makes use of function types. A pointer declarator can be added when using a function type in order to declare a function pointer.

In UML function types are not supported, and the C/C++ Importer will print a warning if it encounters a function type declaration being used. However, the importer can still handle usage of function types with a pointer declarator correctly, and it will treat such definitions in the same way as ordinary function pointer declarations.

**Example 165: Translation of function pointers using function types**

C/C++:

```
typedef void (foo)(int);
foo* pfn;
foo fn; // Yields warning during import
typedef foo foo2;
foo2 fn2; // Yields warning during import
```

UML:

```
public <<operationReference>> interface foo {
  void call(int);
}
public foo extern pfn;
public syntype foo2 = foo;
```

As can be seen in this example, variables typed by a function type is not translated to UML (`fn` and `fn2`). However, if a function type is used with a pointer declarator (`pfn`) it will be translated according to the normal rules for function pointers.

---

## Function body

By default the C/C++ importer does not import the body (statements) of functions that are present in the imported header files. However, there is an option “Action code strategy” in the cppImportSpecification stereotype that can be turned on in order to translate function bodies to UML. This option is intended to be used when the purpose of the C/C++ import is to migrate legacy code to UML.

The body of a C/C++ function is translated to an operation body for the UML operation that is the translation of the function. This operation body will contain a list of actions corresponding to the C/C++ statements of the function body. The translation rules for different kinds of statements are described below.

Note that the “Action code strategy” option also allows function bodies to be imported using informal UML. In that case the entire body of the function will be copied verbatim into an informal UML action.

**Example 166: Translation of function bodies**

C/C++:

```
int m() {
  int a = 3;
  return a;
}
```

UML (with “Action code strategy” set to import using UML actions):

```
int m() {
  int a = 3;
  return a;
}
```

UML (with “Action code strategy” set to import using informal UML):

```
int m() {
[[
  int a = 3;
  return a;
]]
}
```

## Labelled statement

A statement that has a label is translated to a UML action with a corresponding label attached. A UML action may have atmost one label attached. This means that if the statement has multiple labels, the C/C++ importer will insert empty actions before the result action, in order to preserve all labels in the translation.

**Example 167: Translation of labelled statements**

C/C++:

```
A: foo();
B:
EXIT: return;
```

UML:

```
A: foo();
```

```
B: ;
EXIT: return;
```

### Declaration statement

A declaration statement is translated to a UML definition action. Just like in C++, UML definitions may be declared anywhere in a list of actions.

The translation rules for the different kinds of C/C++ actions are described in their own chapters. See for example Enumerated Types, Typedef, Function, Variable, Constant and Class, Struct and Union.

**Example 168: Translation of declaration statements**

C/C++:

```
typedef unsigned int UINT;
UINT a = 5;
```

UML:

```
syntype UINT = 'unsigned int';
UINT a = 5;
```

### Expression statement

An expression statement is translated to a UML expression action. For more information about the translation of the expression within an expression statement see Expression.

If the expression of an expression statement cannot be represented as a corresponding UML expression, the expression action will be translated to an informal UML action ([[...]];), which preserves the original C/C++ expression untranslated.

The expression of an expression statement is optional. If it is omitted the expression statement is translated to an “empty” UML action (a single semicolon).

**Example 169: Translation of expression statements**

C/C++:

```
int a = 5 + 6 - foo();
```

```
bool b = true && !false;
a--;
a = sizeof(int);
a += 5;
;
```

UML:

```
int a = 5 + 6 - foo();
bool b = true && ! false;
a --;
a = [[sizeof(int)]];
[[a += 5]];
;
```

### Break statement

A break statement is translated to a UML break action. See Example 170 on page 578 for an example.

### Continue statement

A continue statement is translated to a UML continue action. See Example 170 on page 578 for an example.

### For statement

A for-statement is translated to a UML for-action, which is a special kind of loop action.

**Example 170: Translation of for statements**

C/C++:

```
for(int i = 0; i < 10; i++) {
  if (i == 5)
    break;
  if (i == 4)
    continue;
}
```

UML:

```
for ( int i = 0; i < 10; i ++) {
    if (i == 5)
        break;
    if (i == 4)
        continue;
```

```
}
```

### Do statement

A do-statement is translated to a UML do-action, which is a special kind of loop action.

**Example 171: Translation of do statements**

C/C++:

```
do {
  compute(x);
} while (x >= 0);
```

UML:

```
do
{
    compute(x);
}
while (x >= 0);
```

### Goto statement

A goto statement is translated to a UML join action.

**Example 172: Translation of goto statements**

C/C++:

```
if (!valid())
     goto ERROR;

  return 0;

  ERROR: return -1;
```

UML:

```
if (! valid())
    goto ERROR;
return 0;
ERROR : return - 1;
```

See also Labelled statement for the translation of labelled statements.

### If statement

An if-statement is translated to a UML if-action. Else branches are also directly translated to corresponding Else branches in UML.

**Example 173: Translation of if statements**

C/C++:

```
x = getX();
if (x == y)
  equal();
else if (x < y)
  less();
else
{
  greater();
}
```

UML:

```
x = getX();
if (x == y)
    equal();
else
    if (x < y)
        less();
    else
        {
            greater();
        }
```

### Switch statement

A switch statement is translated to a UML decision action. Case branches, and the optional default branch, are also translated to corresponding branches in UML.

**Example 174: Translation of switch statements**

C/C++:

```
switch (e) {
case 1 :
{
  i = 1;
  break;
}
case 2  :
  break;
```

```
 default : {i = 3;}
}
```

UML:

```
switch (e) {
    case 1 :
        {
            i = 1;
            break;
        }
    case 2 :
        break;
    default :
        {
            i = 3;
        }
}
```

### While statement

A while-statement is translated to a UML while-action, which is a special kind of loop action.

**Example 175: Translation of while statements**

C/C++:

```
while (true) {
  if (done())
    break;
}
```

UML:

```
while (true) {
    if (done())
        break;
}
```

### Return statement

A return statement is translated to a UML return action. See Example 172 on page 579 for an example.

### Try statement

A try statement is translated to a UML try action. Catch clauses are also translated to corresponding UML catch clauses.

Note that a "re-throw" of an exception in a catch clause must be done in UML by referring to the caught exception parameter. Hence a `catch(...)` clause in C++ is translated to a UML catch clause with a named exception parameter typed by `Any`. The name of this exception parameter is by default `Exception`, but if the catch clause contains a local definition with that name, or a reference to a non-local definition with that name, a suffix is appended to make the name of the exception parameter unique within the scope of the catch clause.

**Example 176: Translation of try statements**

C/C++:

```
try {
  test();
}
catch (char* error_msg)
{
  printf("%s", error_msg);
}
catch (...)
{
  throw;
}

try {
  test2();
}
catch (...) {
  bool Exception = true;
  throw;
}
```

UML:

```
try {
  test();
}
catch ('char*' error_msg)
{
  printf([["%s"]], error_msg);
}
catch (Any Exception)
{
  throw Exception;
```

```
}

try {
  test2();
}
catch(Any Exception1)
{
  bool Exception = true;
  throw Exception1;
}
```

### Compound statement

A compound statement is translated to a UML compound action.

**Example 177: Translation of compund statements**

C/C++:

```
int a = 5;
{
  int a = 6;
  a++;
}
a--;
```

UML:

```
int a = 5;
{
    int a = 6;
    a ++;
}
a --;
```

# Scope Unit

- Namespace
- Class, struct and union
- Template classes

## Namespace

A namespace is translated to a UML package.

**Note**

*Global declarations are placed in the package into which the import is made. That package will be marked by the <<globalNamespace>> stereotype of the TTDCppPredefined profile.*

**Example 178: Translation of definitions in the global namespace**

C/C++:

```
int i;
void op(unsigned int);
```

UML (in the import package):

```
public int i;
public void op('unsigned int');
```

## Class, struct and union

Refer to the main topic Class, Struct and Union.

**Example 179: Translation of nested scope units**

C/C++:

```
class C {
public:
  int ci;
  class CC {
  public:
    int op();
  };
};
```

UML:

```
public class C {
  public int ci;
  public class CC {
    public int op();
  }
}
```

## Template classes

C++ class templates are mapped to UML class templates.

**Example 180: Translation of template nested definitions**

C/C++:

```
template <class C> class String {
  struct Srep {
    C* s;
    int sz;
    int n;
  };
  Srep *rep;
public:
  String();
  String(const C*);
  String(const String&);
  C read(int i) const;
};
```

UML:

```
template <type C > public class String {
  private <<struct>> class Srep {
    public C s;
    public int sz;
    public int n;
  }
  private Srep rep;
  public String();
  public String( const C );
  public String( const inout part String<C> );
  public <<IsQuery="true">> part C read( int i );
}
```

# Variable

A variable is translated to a UML attribute. This rule is valid for both member and non-member variables.

**Example 181: Translation of variables**

C/C++:

```
int ivar, jvar;
class X {
  int j;
public:
  int Get() { return j;};
} xvar;
```

UML:

```
public int ivar;
public int jvar;
public class X {
    private int j;
    public int Get ();
}
public part X xvar;
```

---

### Non-member variable

Non-member variables will result in attributes in the package into which the import is made.

### Member variable

Member variables will become attributes of a `class` or `choice`.

Although the translation rules as such are identical as for a Non-member variable, a C++ member variable may have more "features" in its declaration. The translation rules for such member-specific features of a variable are described in the relevant sections of the section "Class, Struct and Union" on page 590.

## Constant

A constant is translated to a UML attribute which is read-only (i.e. changeability set to "frozen"). This rule is valid for both member and non-member constants.

Non-member constants will result in attributes in the package into which the import is made, while member constants will be put as attributes of a `class` or `choice`.

Although the translation rules as such are identical, a C++ member constant may have more "features" in its declaration. The translation rules for such member-specific features of a constant are described in the section "Member constants" on page 598.

If the constant has a constant expression specified, the corresponding UML attribute will have a default value that is the translation of that expression according to the rules described in Constant expression.

**Example 182: Translation of constants**

C/C++:

```
class MyClass;
const double pi = 3.14159;
const MyClass m(7, 'x');
extern const int extconst; // Defined elsewhere.
```

UML:

```
public class MyClass {}
public const double pi = 3.14159;
public const part MyClass m with (7, 'x');
public const int extern extconst;
```

## Constants as preprocessor macros

It is not uncommon, especially in older C APIs, to have Preprocessor macros represent constants. Such constants will **not** be translated to UML since the preprocessor will remove them before the translation starts. In order to access such constants from the UML model, you have to use inline target code ([[...]]).

**See also**

"Macro" on page 615

# Expression

There are two kinds of expressions in C++; constant and non-constant expressions. When importing declarations only (that is no function bodies) only Constant expressions are translated. When the C/C++ importer is used for migrating legacy code to UML also non-constant expressions may be encountered and will then be translated.

The translation rules used for translating C/C++ expressions into UML expressions are the reverse of the rules for generating C++ from UML expressions using the C++ Application Generator. See "Expression" on page 1575 in Chapter 52, *C++ Application Generator Reference* for more details.

C/C++ expressions that have no corresponding representation in UML are translated to informal expressions ([[...]]).

## Binary and unary expressions

A binary C/C++ expression references an operator taking two operands, while a unary expression references an operator taking just one operand. Some operators of the C/C++ language have no representation in UML. Binary and unary expressions that refer to such operators are translated to informal UML expressions ([[...]]). The table below lists the operators that do have a UML representation. Binary and unary expressions using these operators can therefore be translated to UML.

| C/C++ operator | | Kind |
|---|---|---|
| ! | Logical not | Unary |
| != | Inequality | Binary |
| && | Logical and | Binary |
| * | Multiplicity | Binary |
| + | Addition | Binary |
| + | Plus | Unary |
| - | Subtraction | Binary |
| - | Negation | Unary |
| / | Division | Binary |
| < | Less than | Binary |
| <= | Less than or equal to | Binary |
| = | Assignment | Binary |
| == | Equality | Binary |
| > | Greater than | Binary |
| >= | Greater than or equal to | Binary |
| [] | Array subscript | Binary |
| << | Left shift | Binary |
| >> | Right shift | Binary |
| \|\| | Logical or | Binary |
| ++ | Prefix increment | Unary |
| ++ | Postfix increment | Unary |

| C/C++ operator | | Kind |
|---|---|---|
| -- | Prefix decrement | Unary |
| -- | Postfix decrement | Unary |
| % | Modulus | Binary |
| new | New expression | Unary |

## Constant expression

Constant expressions may be encountered at a number of places in a C/C++ header, for example as constant initialization for variables, or as size specifiers of array type specifiers bitfields.

Constant expressions are sometimes evaluated during the translation to UML. This happens when the result of an expression has a semantic impact also at UML level, for example, a constant expression representing the size of an array.

When evaluating a constant expression there are two kinds of expressions which are not fully supported, since a proper evaluation of these requires knowledge about the compilation environment:

- `cast` **expressions**

  The expression of the `cast` will be evaluated, but no modification will be done of the value (due to type modifications etc.). If a `cast` expression is encountered a warning is issued that the casting is ignored.

- `sizeof` **expressions**

  The evaluation of a `sizeof` expression depends on properties of the platform where the C/C++ code is compiled, and can thus not be handled by the C/C++ import. If a `sizeof` expression is encountered a warning is issued and it is assumed that the expression evaluates to an informal expression.

**Example 183: Translation of constant expressions**

C/C++:

```
enum e { a, b, c = 10 };
const int i = (2+c)*b;
struct s{
  int f1 : (2+c)*b; // Evaluating
};
```

```
const int uncmn = (int) (bool) 4;
typedef int intarr[sizeof(int)+1];
// Assume that sizeof(int)+1 is an informal expression
```

UML:

```
public enum e {
  a,
  b,
  c = 10
}
const int i = [[(2 + c) * b]];
public <<struct>> class s {
  public <<bitfield('size' = 12.)>> int extern f1;
}
const int uncmn = [[(int) (bool) 4]];
public syntype intarr = CArray<int, ([[sizeof(int)+1]])>;
```

Messages:

```
Unable to evaluate sizeof expression. It will be
imported as an informal expression.

Can not translate C++ expression, importing it to
informal expression [[ ... ]]
```

**See also**

"Expression evaluation" on page 622

# Class, Struct and Union

- A class or a struct is translated to a UML class.
- A union is translated to a UML choice.
- Structs are marked with a <<`struct`>> stereotype.

**Example 184: Translation of class, struct and union**

C/C++:

```
class C {};
struct S {};
union U {};
```

UML:

```
public class C {}
public <<struct>> class S {}
public choice U {}
```

## Class, struct, or union without tag

If a class, struct, or union has no tag, it is an incomplete type declaration.

**See also**

"Incomplete Type Declaration" on page 606

## Anonymous union

An anonymous union is translated by making its members become attributes of the UML class or choice that is the translation of the enclosing C/C++ scope unit. The reason for this translation rule is that, contrary to an ordinary union, an anonymous union is no scope unit in C/C++.

**Example 185: Translation of anonymous unions**

C/C++:

```
struct S {
  int i;
  union {
    int j;
    int k;
  };
};
```

UML:

```
public <<struct>> class S {
  public int i;
  public int j;
  public int k;
}
```

**Note**

*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.*

**See also**

"Incomplete Type Declaration" on page 606

## Constructor

A constructor for a class is translated to a UML constructor, that is an operation in the corresponding UML class having the same name as the class.

There are two different kinds of constructors in C++: user-defined constructors, which are explicitly declared and implemented, and implicit constructors, which are implicitly declared and are auto-generated by the C++ compiler (provided that they are not already explicitly declared).

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 parameter-less 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.

There are no auto-generated constructors in UML. Default and copy constructor are not explicitly inserted to imported C++ classes. The reasons are:

- In UML, object assignment (c1 = c2) works because all classes have an implicit conversion to datatype Any, and this type has an assignment operator defined
- auto-generated C++ constructors are explicitly added to the generated code
- The C++ Application Generator produces C++ where auto-generated constructors are present

**Example 186: Translation of constructors**

The example below imports a C++ class with three user-defined constructors and one implicit copy constructor.

C/C++:

```
class C {
public:
  C();
  C(int i);
  C(char c);
};
```

UML:

```
public class C {
  public C();
  public C(int i);
  public C(char c);
```

```
}
```

Constructors may be overloaded just as functions can. The rules for overloaded functions thus also apply on constructors.

---

**See also**

"Overloaded functions" on page 568

## Destructor

A destructor for a class is translated to a UML destructor, that is an operation in the corresponding UML class having the same name as the class prefixed with a tilde character ('~')

**Example 187: Translation of destructors**

C/C++:

```
class D {
public:
  ~D();
};
```

UML:

```
public class D {
  public ~D();
}
```

---

## Member

Member variables of a C++ class are translated to attributes in the UML class, that is the translation of that class. Member functions are translated to operation in the same class.

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 (that is the class).

**Example 188: Translation of class members**

C/C++:

```
class C {
  public:
  int mv1; // Member variable
  void mf1(long long p1); // Member function
  enum e {a,b,c}; // "Member" type declaration
};
```

UML:

```
public class C {
  public int mv1;
  public void mf1 ('long long int' p1);
  public enum e {a, b, c}
}
```

## Member access specifier

Members with `public` (respectively `private` and `protected`) access specifier are translated to members with a "`public`" (respectively "`private`", "`protected`") visibility kind.

The default behavior (when visibility is omitted) corresponds to the C++ rules:

- for structs and unions, members are imported as `public` if visibility is omitted
- class members are imported as `private` if visibility is not specified

**Example 189: Translation of members with different access specifiers**

C/C++:

```
class C {
private:
  int i;
protected:
  int j;
public:
  int k;
  int GetI();
  int GetJ();
  int Calc (int x, int y);
};
```

UML:

```
public class C {
  private int i;
  protected int j;
  public int k;
  public int GetI();
  public int GetJ();
  public int Calc(int x, int y);
}
```

### Virtual member functions

A virtual member function is translated to a UML operation with a “virtual” virtuality kind.

**Example 190: Translation of virtual members functions**

C/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()
};
```

UML:

```
public class CPen {
  public virtual void Draw();
  public double GetRep();
}
public class CPenD : CPen {
  public virtual void Draw();
}
```

### Pure virtual member functions

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 has an impact on how the containing class, which is an abstract class, is translated.

**Example 191: Translation of "pure virtual" class to abstract class**

C/C++:

```
class C {
public:
  virtual int f(int) = 0; // pure virtual member
function
  C() {};
};
class D : public C {
};
```

UML:

```
abstract public class C {
    public int f ( int );
    public C () ;
}
public class D : C {
}
```

C++ classes that contain only pure virtual member functions, can be translated to UML interfaces. This is controlled through the Properties Editor for the cppImportSpecification, by selection of the "Import C++ pure virtual classes to UML interfaces" check box.

**Example 192: Translation of pure virtual class to interface**

C/C++:

```
class Shape {
  public:
    virtual void rotate(int) = 0;
    virtual void draw() = 0;
    virtual bool isclosed() = 0;
};

class Box : Shape {
};
```

UML:

```
public interface Shape {
  void rotate( int );
  void draw();
  bool isclosed();
}
public class Box : Shape {
```

```
}
```

An interface is a “stronger” concept than an abstract class in UML. One difference is that for interfaces the UML checker will print warning messages if one of the interface functions is not implemented in the inheriting (realizing) class. This is not the case for abstract classes, because the implementation can then be provided further down in the inheritance hierarchy.

**Example 193: Warning messages for functions that are not implemented**

```
Class Box: Warning: TSC0124: Operation 'rotate' in
interface 'Shape' was not realized by class 'Box'.
Class Box: Warning: TSC0124: Operation 'draw' in
interface 'Shape' was not realized by class 'Box'.
Class Box: Warning: TSC0124: Operation 'isclosed' in
interface 'Shape' was not realized by class 'Box'.
```

## Static members

A static member is translated to a static UML attribute or operation (that is with “Classifier” owner scope)

**Example 194: Translation of static members**

C/C++:

```
class C {
public:
  static int k;
  static void InitI(int);
};
```

UML:

```
public class C {
    public static int k;
    public static void InitI ( int ) ;
}
```

## Constant members

A constant member is translated as an ordinary member with changeability set to “frozen”.

**Example 195: Translation of constant members**

C/C++:

```
class C {
public:
  const int cm; // constant member
  C(int k) : cm(k) {};
  void Do(double);
  void Undo(double) const; //constant member function
};
```

UML:

```
public class C {
  const int cm;
  public C ( int k ) ;
  public void Do ( double );
  public <<IsQuery="true">> void Undo ( double );
}
```

## Member constants

A member constant is translated as an ordinary member with changeability set to “frozen”, and with a default value.

**Example 196: Translation of member constants**

C/C++:

```
class X {
public:
  static const int i = 99; // member constant
};
const int X::i; // definition of i
```

UML:

```
public class X {
  public static const int i = 99;
}
```

## Mutable member variables

A mutable member variable is translated as an ordinary Member variable.

### Bitfield member variables

A C++ struct, union, or class may have member variables that are bitfields.

Bitfield member variables are mapped to UML member variables with the `<<bitfield>>` stereotype applied (from the TTDCppPredefined profile package). The size is specified as a tagged value. The bitfield size can be edited via the Properties Editor.

**Example 197: Translation of bitfield member variables**

C/C++:

```
struct mybitfields {
  unsigned a : 4;
  unsigned b : 5;
  unsigned c : 7;
};
```

UML:

```
public <<struct>> class mybitfields {
  public <<bitfield('size' = 4.)>> 'unsigned int' a;
  public <<bitfield('size' = 5.)>> 'unsigned int' b;
  public <<bitfield('size' = 7.)>> 'unsigned int' c;
}
```

### Class inside Union

A class defined inside a union is imported into the union owner scope.

This rule governs that a C++ union is imported to a UML choice, but it is not allowed to define classes inside choices in UML.

**Example 198: Translation of classes inside unions**

**C/C++**:

```
typedef struct S1 {
  union U1 {
    struct S2 {
    } vS2;
  } vU1;
};
```

**UML**:

```
public <<struct>> class S1 {
  public <<struct>> class S2 {
```

```
  }
  public <<IsUnion = "true">> choice U1 {
    public part S2 vS2;
  }
  public part U1 vU1;
}
```

## Friend

Friendship between a class C and another declaration D affects what members of C that the implementation of D may access. In UML this is represented by a dependency with the <<friend>> stereotype applied (from the TTDCppPredefined profile package).

**Example 199: Translation of friend declarations**

**C/C++**:

```
class X {};
class Y {
  friend class X;
};
```

**UML**:

```
public class X { }
public class Y <<friend>> dependency to X { }
```

**Note**

*Since the C++ friend concept does not exist in UML the UML Checker will not consider <<friend>> stereotypes when checking the visibility of accessed definitions. Thus, to utilize a friend declaration from behavior code you must use inline C++ code.*

## Inheritance

Inheritance relationships are translated to UML generalizations.

**Example 200: 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();
};
```

UML:

```
public class A {
    public int am;
    public A ( char ) ;
}
public class B : A {
    public char bm;
    public virtual void calc ();
    public void 'set' ();
}
public class C : B {
    public int am;
    public double cm;
    public void calc ();
    public void 'set' ();
}
```

---

In C++ inheritance of unspecified visibility between classes means private visibility. This information is kept using the `<<inheritanceVisibility>>` stereotype of the TTDCppPredefined profile. The reason for this is that by default, public inheritance visibility is used by C++ Application Generator for UML generalization. See Inheritance access specifier for more information.

### Multiple inheritance

The translation rule for C++ inheritance is also used when a class inherits from more than one base class.

**Example 201: Translation of multiple inheritance** ———————————

C/C++:

```
class A {
  public:
  int m;
};
  class B {
  public:
  int m;
  int n;
};

class C: public A, public B {
};
```

UML:

```
public class A {
    public int m;
}
public class B {
    public int m ;
    public int n;
}
public class C : A, B {
}
```

---

### Virtual inheritance

Virtual inheritance is translated in the same way as ordinary inheritance.

A virtual inheritance will be translated by setting the stereotype `<<virtualInheritance>>` from the TTDCppPredefined profile.

### Inheritance access specifier

C++ supports access specifiers (public, protected, private) on inheritance as a means to reduce the accessibility of inherited members. In UML this is not possible (i.e. all inheritance is “public”).

A protected or private access specifier will therefore be translated by applying the `<<inheritanceVisibility>>` stereotype from the TTDCppPredefined profile. Use the Properties Editor on the UML generalization to edit this information.

## Forward declarations

A forward declaration is not translated to UML. This rule is valid for all forward declarations for which there are definitions later on in the header file. This is the most common use 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, an extra type to represent the missing definition is generated.

**Example 202: Translation of forward declarations**

C/C++:

```
class A;
class B {
  public:
    static A& g() { return *pA; }
  private:
    static A* pA;
};
```

UML:

```
public class A {
}
public class B {
  public static (<<CppReference>> part A) g();
  private static A pA;
}
```

Here class A is forward declared, and has no definition in the imported file. Hence the C/C++ Importer will create an empty class for representing it.

## Generation of class and package diagrams

The C/C++ Importer creates class and package diagrams if there are imported definitions to visualize in such diagrams. A class diagram named **C++ Imported Types** is then generated and contains symbols for imported classes, interfaces and datatypes (not syntypes and choices though) including lines representing generalizations and associations.

The C/C++ Importer will also generate a package diagram named **C++ Imported Packages** for a package containing other (nested) packages.

**Example 203: Creating class diagram** ——————————————————————

C/C++:

```
namespace N{
  class A  { int z; };
  class B : A { long y; };
  class C : A { double x; };
}
class D { long h; };
class E { D d1; };
class F { D* d2; };
```

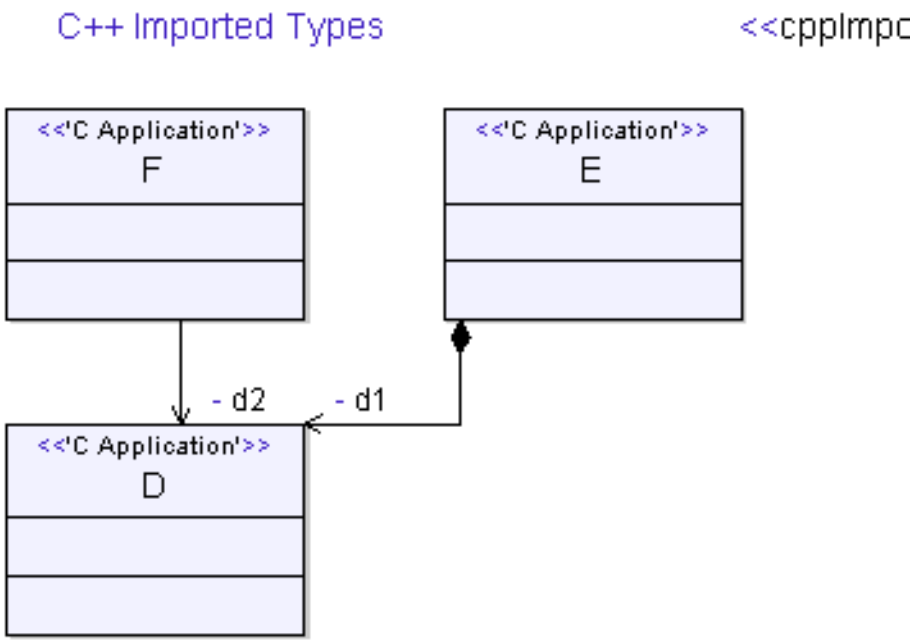



*Figure 169: Class diagram "C++ Imported Types" in package "N"*

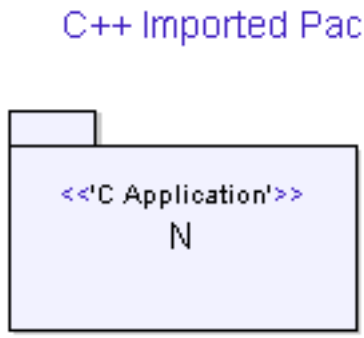



*Figure 170: Package diagram "C++ Imported Packages" in import package*

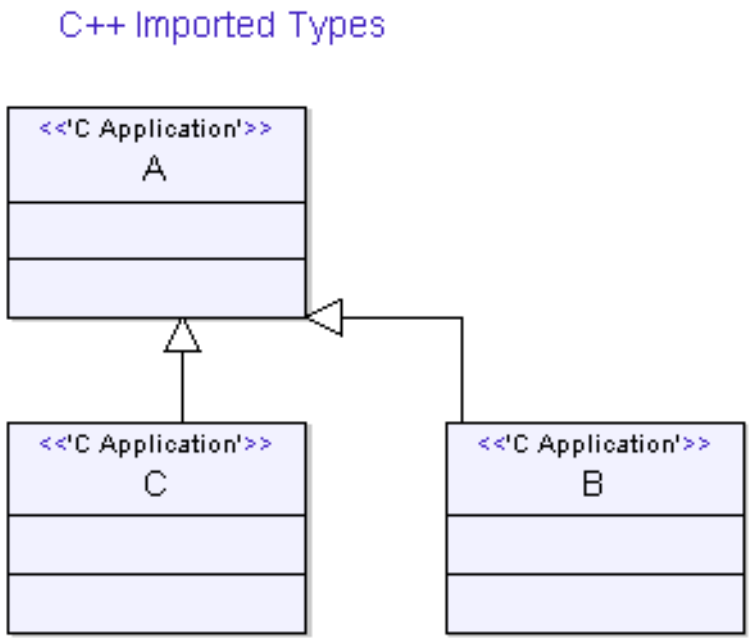



*Figure 171: Class diagram "C++ Imported Types" in package "N"*

# Incomplete Type Declaration

C/C++ allows declarations of incomplete classes, structs, unions and enumerations. These types are declared as incomplete by not giving them a tag. Incomplete types are therefore also sometimes called tag-less types.

Incomplete types can be used in

- Data declarations (for example variables, constants etc.)
- Type declarations (for example `typedef`)
- "Pointless" declarations (for example without declaring neither data nor type). Incomplete types in "pointless" declarations will not be translated to UML, and warnings are issued that the declarations were ignored.

Incomplete types are imported in the same way as named types of the same kind. For type declarations, no syntype will be generated unless more than one type name is defined with the same type declaration.

**Example 204: Incomplete types in type definitions**

C/C++

```
typedef enum { aa, bb, cc } tEnum1;

typedef class {
public:
  int i;
  int j;
  enum tEnum2 { xx, yy, zz } k;
  void init(int ii, int jj, tEnum1 kk);
} CIncomplete;
```

UML

```
public enum tEnum1 {
    aa,
    bb,
    cc
}
public class CIncomplete {
    public int i;
    public int j;
    public enum tEnum2 {
        xx,
        yy,
        zz
    }
    public tEnum2 k;
    public void init( int ii, int jj, tEnum1 kk);
```

```
}
```

Incomplete types that are used in data or type declarations are given the name of the last declared variable or type, prefixed with "`incomplete_`".

**Example 205: Translation of incomplete types**

C/C++:

```
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 - "pointless" declaration
struct {
  int i;
}; // Missing variable name - "pointless" declaration
```

UML:

```
public <<struct>> class S {
  public int i;
  public <<struct>> class 'incomplete_ss3@1' {
    public int j;
  }
  public part 'incomplete_ss3@1' ss1;
  public 'incomplete_ss3@1' ss2;
  public CArray<'incomplete_ss3@1', 2> ss3;
}
public incomplete_ss3 enum  {a, b, c}
public syntype ss1 = incomplete_ss3;
public syntype ss2 = CPtr<incomplete_ss3>;
public syntype ss3 = CArray<incomplete_ss3, 2>;
```

## Note

*Incomplete classes, structures and unions define scope units although they are incomplete. The translation rules in chapter "Scope Unit" on page 583 apply as usual.*

# Overloaded Operator

Overloaded operators are imported to UML operator definitions.

**Example 206: Overloaded operators**

C/C++:

```
class MyInt {
public:
   int x;
   MyInt operator+ ( const MyInt& i );
};
```

UML:

```
public class MyInt {
    public int x;
    public part MyInt '\+'( const inout part MyInt i );
}
```

**See also**

"Overloaded conversion operators" on page 620

# Template

- Class template
- Function template

**See also**

"Pointer to constant" on page 624

## Class template

C++ class templates are translated to UML class templates. C++ template parameters are translated to UML template parameters. Value template parameters are translated to UML const value template parameters, all other template parameters are mapped to UML type template parameters.

**Example 207: Translation of class templates**

C/C++:

```
template <class T, int i> class Buffer {
  T v[i];
  int size;
};
```

UML:

```
template <type T, const int i> public class Buffer {
  private CArray<T, ([[i]])> v;
  private int 'size';
}
```

Template class members are mapped in the same way as members of an ordinary class. If the template name is referenced from the body of a template class, actual template parameters are attached. Otherwise the corresponding UML reference will not bind to the template.

**Example 208: Translation of class templates**

C/C++:

```
template <class C> class String {
public:
  String();
  String(const C*);
  String(const String&); // reference to template name
};
```

UML:

```
template <type C> public class String {
  public String();
  public String( const C );
  public String( const inout part String<C> ); // actual
parameter C attached
}
```

A C++ template instantiation is mapped to a UML template instantiation. Actual values for value template parameters will be translated as constant expressions; see Constant expression.

**Example 209: Translation of template instantiation**

C/C++:

```
template <class T, int i> class Buffer {
  T v[i];
```

```
  int sz;
};

Buffer<char, 127> cbuf;
```

UML:

```
template <type T, const int i> public class Buffer {
  private CArray<T, ([[i]])> v;
  private int sz;
}

public part Buffer<char, 127> cbuf;
```

**Example 210: Translation of template instantiation**

C/C++:

```
template <class C> class String {
  C*rep;
public:
  String();
  String(const C*);
  String(const String&);
  C read(int i) const;
};

String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;

class Jchar {
};
String<Jchar> js;
typedef String<char> CString;
```

UML:

```
template <type C> public class String {
  private C rep;
  public String();
  public String( const C );
  public String( const inout part String<C> );
  public <<IsQuery="true">> part C read( int i);
}

public part String<char> cs;
public part String<'unsigned char'> us;
public part String<wchar_t> ws;

public class Jchar {
```

```
}
public part String<Jchar> js;
public syntype CString = String<char>;
```

## Function template

C++ function templates are translated to UML function templates. C++ function template parameters are translated to UML function template parameters in the same way as for template classes (see Class template).

**Example 211: Translation of template functions**

C/C++:

```
template <class T> void sort(int *);
```

UML:

```
template <type T > public void sort( CPtr<int> );
```

## Default template arguments

C++ template arguments with default values specified are translated to UML template parameters with default values.

**Example 212: Default template arguments**

C/C++:

```
template <class T, class U = char> class C {
public:
  T t;
  T f();
  C(U chr);
};

C<int> var1;
C<int, char> var2;
C<int, bool> var3;
```

UML:

```
template <type T, type U = char> public class C {
  public part T t;
  public T f();
  public C( part U chr );
```

```
}

public part C<int> var1;
public part C<int, char> var2;
public part C<int, bool> var3;
```

# Exception

The specification that a function may throw certain exception types are translated into a corresponding specification of the UML operation that is the translation of the function.

If a function is specified to throw no exceptions at all (`throw()`) this is represented in UML by the application of the `<<noException>>` stereotype of the TTDCppPredefined profile.

**Example 213: Translation of exception specification on functions**

C/C++:

```
double foo() throw(char, int);
void bar() throw();
```

U2:

```
public double foo() throw char, int;
public <<noException>> void bar();
```

# Miscellaneous

- Language constructs
- Compiler-specific language constructs
- Non-language constructs
- Translation rules for C compilers

# Language constructs

## Volatile

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 UML translation.

## Linkage

The linkage of a C/C++ definition is normally not visible in the UML translation. However, if the definition have extern linkage explicitly specified this will be represented in UML by means of the ‘external’ property on the imported definition.

**Example 214: Translation of definitions with different linkage**

C/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;
  };
}
static void foo();
```

UML:

```
public int extern a;
public <<struct>> class S {
  public int x;
}
public void foo();
```

## Using directive

A C++ using directive (“using namespace”) is translated to an <<access>> dependency. The supplier of the dependency is the package that corresponds to the referenced namespace.

**Example 215: Translation of using directives**

C/C++:

```
namespace X {
  class C {};
}

using namespace X;

C var;
```

UML:

```
package ImportedDefinitions <<access>> dependency to X {
    package X {
        public class C { }
    }
    public part C extern var;
}
```

## Using declaration

A C++ using declaration (“using”) is translated to an <<access>> dependency. The supplier of the dependency is the UML definition that corresponds to the referenced C++ definition.

**Example 216: Translation of using declarations**

C/C++:

```
namespace K
{
  class A {};
}

using K::A;
A aInst;
```

UML:

```
<<access>> dependency to K::A;

package K {
    public class A { }
}
public part A extern aInst;
```

## Compiler-specific language constructs

Most C/C++ compilers add their own specific additions to the C/C++ language they support. The C/C++ Importer allows the presence of such constructs in the input files if the appropriate language dialect is used (see C/C++ dialect for more information). However, usually these language extensions have no impact on the UML translation.

Exceptions to this rule are listed below.

### __declspec

Use of the `__declspec` keyword supported by the Microsoft and GNU compilers is translated using the `<<__declspec>>` stereotype of the TTDCpp-Predefined profile.

**Example 217: Translation of the __declspec keyword**

C/C++:

```
void __declspec(dllexport) foo();
```

UML:

```
void <<__declspec(. modifier = __declspecModifier(. kind
= "dllexport".) .)>> foo();
```

## Non-language constructs

### Macro

Macros are preprocessed and expanded to the values specified by you in the C/C++ files and also using the values defined in the preprocessor Options. The resulting C/C++ code is then translated to UML. Refer to the sections about translation rules for respective language constructs.

**See also**

"Preprocessor" on page 622

### Referencing predefined types in C/C++

It is possible to use special type references in imported C/C++ headers which will be mapped to the predefined UML types. These type references are on the form `SDL_<name>`, where <name> is the name of a predefined UML type. For example, `SDL_Integer` is mapped to the UML Integer type.

The prefix `SDL` is used since this feature mainly is designed to be used with the C code generator which uses this prefix on its C implementations of the predefined types. If the imported header file includes the definition of these types from the C code generator library headers, the option Translation of depending declarations can be set to **Off** in order to prevent new UML types to be generated for these `SDL_` prefixed types.

The table below lists these special type references that can be recognized by the C/C++ Importer:

| C/C++ type reference | UML type |
|---|---|
| `SDL_Boolean, SDL_boolean` | `Boolean` |
| `SDL_Integer, SDL_integer` | `Integer` |
| `SDL_Real, SDL_real` | `Real` |
| `SDL_Natural, SDL_natural` | `Natural` |
| `SDL_Time, SDL_time` | `Time` |
| `SDL_Duration, SDL_duration` | `Duration` |
| `SDL_PId` | `Pid` |
| `SDL_Character, SDL_character` | `Character` |
| `SDL_Charstring, SDL_charstring` | `Charstring` |
| `SDL_IA5String` | `IA5String` |
| `SDL_NumericString` | `NumericString` |
| `SDL_VisibleString` | `VisibleString` |
| `SDL_PrintableString` | `PrintableString` |
| `SDL_Bit` | `Bit` |
| `SDL_Bit_String` | `BitString` |
| `SDL_Octet` | `Octet` |

| C/C++ type reference | UML type |
|---|---|
| SDL_Octet_String | OctetString |
| SDL_Object_Identifier | ObjectIdentifier |
| SDL_Null | Any |

When any of these names are recognized during the translation, an information message is printed.

**Example 218: Using C names of predefined UML types**

C/C++:

```
SDL_Integer func();
```

UML:

```
public Integer func();
```

Message:

```
Information sdltypes.h(8): Recognized C++ name
'SDL_Integer' of predefined SDL type. Imported to
'Integer'
```

## Translation rules for C compilers

### Language

The Language attribute in C Application stereotype will set to C instead of C++.

# STL support

A special library package "std" with UML versions of C++ Standard Template Library definitions is available. These definitions will be referenced from the UML model when a C++ header using STL definitions is imported to UML. To use these definitions, there is thus no need to import standard STL headers to UML.

The `std` package supports:

- STL containers:

  `vector, list, set, multiset, map, multimap, pair, deque, queue, priority_queue, stack, string`

- STL algorithms:

  `for_each, find, find_if, adjacent_find, count, count_if, search, search_n, find_end, find_first_of, iter_swap, swap_ranges, transform, replace, replace_if, replace_copy, replace_copy_if, generate, generate_n, remove, remove_if, remove_copy, remove_copy_if, unique, unique_copy, reverse, reverse_copy, rotate, rotate_copy, random_shuffle, partition, stable_partition, sort, stable_sort, partial_sort, partial_sort_copy, nth_element, lower_bound, upper_bound, equal_range, binary_search, merge, inplace_merge, includes, set_union, set_intersection, set_difference, set_symmetric_difference, push_heap, pop_heap, make_heap, sort_heap, max_element, min_element, next_permutation, prev_permutation`

- STL Input/Output facilities; examples of these are:

  `basic streams and file input/output streams including cin, cout, cerr, <<, >>`

If STL support is required in UML, the `std` package should be added to the project by activating the add-in `CppStdLibrary`.

**Note**

*The STL support is currently limited to work with the C++ Application Generator. STL can not be used with the C Code Generator. In general, any imported C++ code that uses templates can not be used with the C Code Generator.*

# C/C++ Import and Build Types

- C Code Generator
- C++ Application Generator

## C Code Generator

The following code generation attributes in the C Application stereotype will be set when importing:

- Language is set to C (when importing in C mode) or C++ (otherwise).

- C name is used to declare a reference name for `struct`, `union` and `enum` when importing in C mode.

## C++ Application Generator

For the C++ Application Generator, stereotypes from the TTDCppPredefined profile are used. You can find information about these stereotypes in the Package TTDCppPredefined documentation.

# Known Restrictions

## C++ language restrictions

### Overloaded conversion operators

Overloaded conversion operators are not supported. The following warning message is issued:

```
Ignoring conversion operator. Conversion operators are
not yet supported.
```

### Overloading on const

Overloading on const is not supported. Such overloaded definition is ignored during import with a warning message:

```
Ignored conflicting declaration
```

### Ellipsis function

Ellipsis function arguments are ignored. The following warning message is issued:

```
Cannot translate ellipsis function 'MyFunc'.
```

However, the function itself is imported (including the arguments preceding the ellipsis). See Unspecified argument for a technique that can be used to call different versions of an ellipsis function from UML.

### Function pointers

There are some limitations when importing function pointers:

1. Formal parameters that have a reference type are not considered.

**Example 219: Formal parameters with a reference type**

C++:

```
typedef int (*pf)(int& );
int f1(int& i);
const pf pf1 = f1;
```

UML:

```
public <<operationReference>> interface pf {int
call(inout int);}
public int f1(inout int i);
public const pf pf1 = operation f1(int);
```

2. Formal parameters that have a constant type are not considered.

**Example 220: Formal parameters with a constant type**

C++:

```
typedef int (*pf)(const int);
int f1(const int i);
const pf pf1 = f1;
```

UML:

```
public <<operationReference>> interface pf {
     int call(const int);}
public int f1(const int i);
public  const pf pf1 = operation f1(int);
```

3. Formal parameters that have a class type transmitted by value are not considered.

**Example 221: Formal parameters with a class type transmitted by value**

C++:

```
class X {};
typedef int (*pf)(X);
int f1(X i);
const pf pf1 = f1;
```

UML:

```
public class X {}
public <<operationReference>> interface pf {
     int call( part X);
}
public int f1( part X i);
public const pf pf1 = operation f1(X);
```

4. Function types are not supported (although pointers to function types are). See Function type for more information.

### Exceptions

All constant types that are declared as exception types will be imported without a `const` qualifier and a warning message will be produced to inform you of this:

```
Constant type is used as exception. This exception will
be imported without const
```

**Example 222: Constant as exception**

C++:

```
void foo () throw (const int);
```

UML:

```
public void foo() throw int;
```

### Preprocessor

Macro names and macro definitions as well as other preprocessor constructs are not translated to UML.

### Expression evaluation

Most of the expressions are not evaluated during import, they are translated to an informal UML expression. Expression are evaluated when the result of an expression has semantic impact also at UML level, for example, constant expression representing the size of an array.

There are two restrictions for evaluated expressions.

- `sizeof` expressions are not evaluated during import, instead they are imported as informal expressions:

```
struct afx_float {
  char floatBits[ sizeof(float) ];
};

Warning: Unable to evaluate sizeof expression. It will be
imported as informal expression.

public <<struct>> class afx_float {
  public CArray<char, ([[sizeof(float)]])> floatBits;
}
```

- `cast` expressions are ignored during expression evaluation, so it will not influence on modification of the value due to type modifications at UML level.

Furthermore, some of the operators in C/C++ have no representation in UML. Binary or unary expressions using these operators are therefore translated to informal expressions. Examples of such operators include '+=', '-=', ',' etc. See Binary and unary expressions for more information.

### Implicit conversions from int to enum

Implicit conversions from int to enum or from enum to int are not supported.

**Example 223: Implicit and explicit cast**

```
public enum cars {  volvo, saab, audi, vw, bmw  }
...
int a;
a = volvo; // ERROR: not allowed
...
int b;
cars c;
b = cast<int>(volvo);
c = cast<cars>(1);
```

### Using template parameter as qualifier

When a template parameter is used as qualifier, the semantic analysis may report errors on imported definitions. This is due to that it is not allowed to use template parameter as a qualifier in templates.

**Example 224: Error from imported syntype**

C++:

```
template <class X> class Y {
  typedef X::pointer iterator;
  iterator End;
};
```

Imported UML:

```
template <type X > public class Y {
  private syntype iterator = X::pointer;
  private iterator End;
}
```

Imported syntype will produce error messages:

```
Syntype iterator: Error: TNR0047:
Failed to find definition of pointer (while looking for Type).
Syntype iterator: Error: TNR0034:
Failed to find Type X::pointer of Syntype.
```

### Pointer to constant

There is no representation of a C++ const pointer in UML. C++ pointer to constant is not supported in UML.

Consider, for example, the C++ type `const char*`. It will be imported to `const 'char*'` type in UML, which is a constant, i.e. it should be assigned a value only once during initialization. However, C++ `const char*` is a pointer to a constant, which means that it is a variable pointing to a constant and it can change its value:

```
const char* cc;
cc = "constant string 1";
cc = "constant string 2";
```

More limitations apply with the C code generation - UML constants can not be initialized by a function call. This will cause an error message when trying to use an imported function that returns a pointer to a constant string:

**Example 225: Using C++ pointers to constant in UML**

C/C++:

```
const char* func1( const int a );
```

Imported UML:

```
public 'char*' func1( const int a );
```

Storing external function result:

If const 'char*' variable is used to store func1 return value in UML, then as a constant, it should be initialized upon definition:

```
const 'char*' cc = func1( 1 );
```

and the following error message is printed by C code generator:

```
ERROR TIL2084: Procedure call not allowed where constant required
```

If 'char*' variable is used:

```
'char*' cc = func1( 1 );
```

then error message is printed by the C compiler:

```
const01.c(511) : error C2440: '=' : cannot convert from 'const char
*' to 'char *'
```

---

It is not possible to use C++ pointers to constant in UML, in particular, C++ constant strings that are returned by some functions.

There are two possible alternatives:

1. Change the C++ header files to use a typedef instead of directly using const char*. Then it is possible to use this external type to store C++ constant strings.

**Example 226: Storing C++ constant strings in UML**

C/C++:

```
typedef const char* cstr;
cstr func1( const int a );
```

Imported UML:

```
public syntype cstr = 'char*';
public cstr func1(const int a);
```

Storing external function result:

```
cstr cc = func1( 1 );
```

---

2. Leave the C++ header files as they are and instead use a UML cast (`cast<'char*'>`) operation to call the external function. The external function can be called as:

```
'char*' cc = cast<'char*'>( func1(1) );
```

**Note**

*When using assigned value it is important that the string pointed to by the variable is not changed. The external C++ code most likely relies on the value being a constant.*

# Usability restrictions

### Importing a header with standard includes

When an imported header file contains other included header files, and the option Do not import definitions from included header files is switched off, all definitions from included files will be imported. When the tree of include files is big, especially for standard includes, the imported model becomes large which may affect performance. Individual definitions can be imported using Selective import.

# 16

# *DOORS Import*

For information on how to import requirements and links created in DOORS into Tau, see "Importing requirements" on page 1726 in Chapter 59, *Working together with DOORS*.

# 17

# *SDL Import*

This chapter describes the support in Tau for import of SDL specifications and the resulting transformation to UML models.

# Operation Principles

The SDL import is based on a file containing an SDL specification and transforms it to a corresponding UML model using a set of predefined transformation rules ("SDL to UML Transformation Rules" on page 636). The resulting model is then automatically loaded and the user can proceed with the remaining work in order to achieve the intended results towards a complete UML model.

## Import an SDL system

### SDL import for SDL Suite

The SDL System must be complete and semantically correct to ensure a successful import. This can be confirmed by running a Full Analyze on your system before exporting.

CIF information can be added to SDL PR files by the SDL Suite CIF exporter, which should be invoked using the **Generate/Convert GR to Telelogic Tau G2 CIF...** menu choice in SDL Suite (SDL Suite 4.6.1 and later). Refer to Telelogic Tau SDL Suite documentation for more information about this feature.

**Note**

*State machine diagrams from process diagrams will only be generated if import is done from CIF. The layout of elements on the diagram will be very similar to layout in the original SDL model.*

To get a convenient format of SDL-PR with CIF comments for SDL import from SDL Suite 4.6.0 and earlier, the command **Generate/Convert GR to CIF...** should be used. The following settings/options should be applied

- **CIF generation** should be chosen for `*.sdt` file
- **Generate one CIF file** should be **on**
- **Include CIF comments** boxes should be **on**
- **Include graphical SDT references** should be **off**.
- **Desired CIF file name:** should be placed in the target directory of your system

**Note**

*For the models imported from SDL without CIF information, state chart diagrams can be created by drag-and-drop state machine implementation onto state machine diagram. In this case autolayout is applied to the created diagram elements (see Diagram auto layout for more informtion about autolayout).*

If the SDL system utilizes the CPP2SDL utility or if external C/C++ definitions exist in the system. Definitions tagged 'EXTERNAL' will not be represented in the imported UML model. If these definitions are used within the model then name-resolution errors will occur after import of the SDL system. To avoid this import the external C/C++ code before proceeding and apply the noScope stereotype to the resulting package. Before code generation this stereotype will have to be replaced with a dependency between the required packages or errors will be issued.

In order to activate SDL import, a workspace with a project must be open.

- Select **Model** item in Model View.
- Open the Import Wizard (**File** menu, **Import...** command).
- Select **Import SDL** in the dialog window and press **OK**.
- Specify the file to import in the dialog window that appears.
- **SDL Dialect:** should be SDL Suite.
- **SDL/PR or SDL/CIF** should be set according to your desired results.

The following should be the result when the second dialog closes:

- A package **ImportedSDLDefinition** is created in the model
- A stereotype **sdlImportSpecification** is applied to the package.
- The SDL file that the import is based on is stored as a value in the stereotype instance for the package.
- The import operation is performed, and the result is added to the created package.

### Importing from Japanese edition of SDL Suite

**Note**

*This is applicable for the Japanese edition of SDL Suite only and can be disregarded from when importing from the standard (English) version of SDL Suite.*

SDL Suite Japanese edition, which is supported on Solaris and Windows systems, uses “native” encoding when storing Japanese texts (such as character strings and comments). Native in this case means SHIFT-JIS on Windows and EUC-JP on Solaris.

However, for SDL import it is assumed that texts to be encoded are using UTF-8 and therefore files created by SDL Suite need to be converted prior to import. As a user convenience, this conversion is done prior to import, by calling the `iconv` utility, a standard program for converting various encodings to UTF-8 under Solaris/Linux/Cygwin. (`iconv` is provided with the Tau installation).

The encodings that supported by iconv can be listed with `iconv -l` Any of these values could be assigned to the environment variable `TauImportedSDLEncoding` in order to specify which encoding is used for the input SDL files. The useful values in this case are:

- **SHIFT-JIS** if SDL files were created using SDL Suite Japanese edition on Windows
- **EUC-JP** if SDL files were created using SDL Suite Japanese edition on Solaris
- **UTF-8** if files have already been converted to UTF-8

The value of the environment variable is passed verbatim to the `iconv` utility to specify the encoding of the source SDL files. If the variable is not present or has no value, then it is assumed that files are encoded using UTF-8.

### SDL import for ObjectGeode

The ObjectGeode SDL import interface is in essence identical to the Import from SDL Suite interface. The difference being the dialect.

- **SDL Dialect:** should be ObjectGeode.
- **SDL/PR or SDL/CIF** should be set according to your desired results.
- Files has to be saved as **SDL for Export** in ObjectGeode to be possible to import.

**Note**

*SDL-PR with CIF information is the default, native storage format for ObjectGeode and CIF comments are printed when the SDL file is saved.*

## Supported SDL

The SDL import supports SDL-96, either expressed as plain SDL-PR (textual syntax) or as SDL-PR enriched with CIF comments (Common Interchange Format).

**Note**

*For more information about SDL and CIF, refer to the Z.100 and Z.105 recommendations from ITU-T. For more information about how SDL and CIF are supported in Tau SDL Suite and ObjectGeode, refer to the user documentation for these tools.*

### SDL-PR

The SDL import supports plain SDL-PR specifications. Such specifications are imported to UML models, and the semantics of the original SDL specification has been preserved as much as possible.

**Example 227: SDL-PR text specifications**

```
system MySystem;
  signal ok;
  channel c from b to env with ok; endchannel;
  block b;
    procedure out_ok; returns Boolean;
      start;
        output ok;
        return true;
    endprocedure;
    signalroute sg from p to env with ok;
    connect c and sg;
    process p(1,1); signalset;
      dcl v Boolean;
      start;
        task v := "not"(call out_ok);
        stop;
    endprocess;
  endblock;
endsystem;
```

### CIF

The SDL import also supports import of graphical information from SDL specifications represented using the CIF (Common Interchange Format for SDL) comments. Importing CIF preserves as much as possible of the graph-

ical layout such as positioning and size of symbols, lines and text attributes, so that the generated UML models and presentation elements look familiar to the user.

**Example 228: SDL specification enriched with graphical information**

```
/* CIF SystemDiagram */
/* CIF Page 1 (1900,2300) */
/* CIF Frame (150,150),(1600,2000) */
/* CIF PackageReference (175,25),(200,100) */
/* CIF Specific SDT Version 1.0 */
/* CIF Specific SDT OriginalFileName
'C:\Telelogic\SDL_TTCN_Suite4.4\sdt\examples\demongame\D
emonGame.ssy' */
/* CIF Specific SDT Page 1 Scale 100 AutoNumbered */
system DemonGame;
/* CIF CurrentPage 1 */
/* CIF Text (600,250),(200,100) */
SIGNAL
Newgame, Probe, Result, Endgame,
Win, Lose, Score(Integer), Bump;
/* CIF End Text */
/* CIF Channel (150,475),(600,475) */
/* CIF TextPosition (525,425) */
/* CIF TextPosition (275,500) SignalList1 */
/* CIF Arrow1Position (262,475) */
channel C1 from env to GameBlock with Newgame, Probe,
Result, Endgame;
endchannel C1;
```

# Supported tools and versions

The SDL import supports the following tools and versions.

- Import from Tau SDL Suite version 4.4 and later
- Import from ObjectGeode version 4.2

## Import from SDL Suite

The level of SDL support, regarding the precise SDL version and proprietary extensions is the same as the support provided by the SDL Analyzer in SDL Suite, operating in case sensitive mode.

Refer to the SDL Suite user documentation for more information on how to convert an SDL system to be case sensitive: User's Guide, CPP2SDL Migration Guide, Migration Guidelines, Update to case-sensitive SDL.

### Import from ObjectGeode

For ObjectGeode, the level of SDL support, regarding the precise SDL version and proprietary extensions is the same as the SDL support in ObjectGeode SDL_API.

Refer to the user documentation for ObjectGeode for more information.

## Activate SDL import

The SDL import is called from the Tau graphical user interface. The SDL Import functionality is enabled through specific Add-Ins for the supported tools. The appropriate add-in will normally be activated by the first use of the Import wizard.

It is possible to activate it manually; from the **Tools** menu select Customize and then select the appropriate add-in.

**Note**

*To be able to activate the SDL import, a workspace must first be opened and a project must be created, otherwise the add-in can not be activated.*

Both SDL Suite and ObjectGeode SDL import are available through Add-Ins. There is one add-in designated for each of these tools. Make sure to select the add-in that corresponds to the tool you want to import data from.

- Select the **SDL96Import** addin in order to enable the Tau SDL Suite SDL Import.
- Select the **OGSDLImport** addin for ObjectGeode SDL Import.

# SDL to UML Transformation Rules

## Structure and scopes

### Top-level definitions

The following top-level definitions from SDL specification are imported:

- Package definition
- System definition
- Type based system definition

A top-level system is imported to a package containing an active class. All signals, signal lists, types, synonyms in the system scope and interfaces defined for remote definitions are inserted into the package. The system itself with all other definitions is mapped to an active class inside a package. Full qualifiers, if generated, start from the target package where all imported definitions are inserted.

**Example 229: System definition**

SDL

```
system MySystem;
  signal ok;
  ...
  output ok to p;
  output ok to system MySystem/block b1 p;
  ...
endsystem;
```

UML

```
package MySystem {
  active class MySystem {
    ...
    ^ p.ImportedSDLDefinitions::MySystem::ok();
    ^
ImportedSDLDefinitions::MySystem::MySystem::b1_T::p.ok()
;
    ...
  }
  signal ok;
}
```

As a support for the command line version of SDL import, a build artifact pointing to the active class is also generated. Full qualifiers, if generated, start from the global scope. The build artifact is generated only if SDL import is operated from a command line.

**Example 230: Importing system definition from the command line**

UML

```
package MySystem {
  active class MySystem {
    ...
    ^ p.::MySystem::ok();
    ^ ::MySystem::MySystem::b1_T::p.ok();
    ...
  }
  signal ok;
}
artifact Build <<manifest>> dependency to
MySystem::MySystem { }
```

An SDL package is imported to a UML package that contains imported entities from the corresponding SDL package.

**Example 231: Package definition**

**SDL**

```
package MyPackage;
  ...
endpackage;
```

**UML**

```
package MyPackage {
  ...
}
```

A package use clause is transformed to a dependency with the «access» stereotype. Use clauses from the system definition is applied to the created package with the active class.

**Example 232: Package use**

**SDL**

```
package MyPackage;
```

```
  ...
endpackage;

use MyPackage;
system MySystem;
  ...
endsystem;
```

**UML**

```
package MyPackage {
  ...
}

package MySystem <<access>> dependency to MyPackage {
  active class MySystem
   MyPackage {
  ...
  }
}
```

---

A type based system definition is imported to an active class which inherits the specified parent system type.

**Example 233: Type based system definition** ——————————

**SDL**

```
system MySystem : MySystemType;
```

**UML**

```
active class MySystem : MySystemType {
}
```

---

### System type, block type, process type

System types, block types and process types are imported into active classes that contain imported definitions, which correspond to the scoped definitions from the imported SDL entity.

**Example 234: Block type and Process type** ——————————

**SDL**

```
block type bt1;
  process type p1;
    ...
```

```
  endprocess type;
  ...
endblock type;
```

**UML**

```
active public class bt1 {
  active public class p1 {
     ...
  }
  ...
}
```

---

Process type inheritance is mapped to class generalization and state machine generalization.

**Example 235: Process type inheritance** ——————————————————

**SDL**

```
process type ptype; fpar i Integer;
  start virtual;
    stop;
endprocess type;

process type psuper inherits ptype; fpar k Integer;
  start redefined;
    stop;
endprocess type;
```

**UML**

```
active public class ptype {
  virtual statemachine ptype( Integer fpar_i) {
    virtual start {
      ptype::i = fpar_i;
      stop;
    }
  }
  Integer i;
}

active public class psuper : ptype {
  virtual statemachine psuper( Integer fpar_k) :
ptype(Integer) {
     redefined start {
       ptype::i = fpar_i;
       psuper::k = fpar_k;
       stop;
     }
  }
  Integer k;
```

```
}
```

### Service type

Service type is not supported.

### Block instance, process instance

Block instances and process instances are transformed to attributes of an active class that corresponds to the governing block type or process type. Aggregation of imported attributes is set to “composite”:

**Example 236: Block instance and Process instance**

**SDL**

```
block type bt1;
  ...
endblock type;
block b1 : bt1;
```

**UML**

```
active public class bt1 {
...
}
part bt1 b1;
```

The initial number of process instances is mapped to the initial number of attribute instances in UML.

**Example 237: Initial number of process instances**

**SDL**

```
process type p1;
  ...
endprocess type;
process pi(10):p1;
process pii(1):p1;
```

**UML**

```
active public class p1 {
  ...
}
part p1 pi / 10;
```

```
part p1 pii / 1;
```

If the maximum number of process instances is specified, it is imported to attribute Multiplicity. The multiplicity is equal to range `0..max`, where `max` is the maximum number of instances.

**Example 238: Maximum number of process instances**

**SDL**

```
process type p1;
  ...
endprocess type;
process pi(10,20):p1;
process pii(0,11):p1;
```

**UML**

```
active public class p1 {
  ...
}
part p1 [0..20] pi / 10;
part p1 [0..11] pii / 0;
```

### Service type instance

Service type instance is not supported.

### Block and process

Blocks and processes are imported as attributes of active classes, similar to the block instance and process instance import, but active classes are created from imported blocks and processes and get the same name with the suffix "_T" (T standing for Type). Parent type is specified as inline type in the imported model.

- For blocks, the initial number of attribute instances is set to 1.
- For processes, the initial and maximum number of process instances is imported according to the table below.

| SDL process | UML attribute | Condition |
|---|---|---|
| `process p(N, MAX)` | `[0..MAX] p / N` | |
| `process p(1); process p();` | `p / 1` | There are NO `create p;` statements |
| `process p(1); process p();` | `[0..*] p / 1` | There is at least one `create p;` statement |
| `process p(N);` | `[0..*] p / N` | `N == 0` or `N > 1` |

**Example 239: Block and process**

**SDL**

```
block b;
  process p(1,1);
    ...
  endprocess;
  ...
endblock;
```

**UML**

```
part active class b_T {
  part active class p_T {
    ...
  } [0..1] p / 1;
  ...
} b / 1;
```

### Block substructure

Block substructure is not imported to UML. All definitions from substructure are imported to the active class corresponding to the substructure owner, as shown in Example 240 on page 642.

**Example 240: Block substructure**

**SDL**

```
block type bt;
  substructure;
    block bs;
      ...
    endblock;
```

```
    ...
  endsubstructure;
endblock type;
```

**UML**

```
active public class bt {
  part active class bs_T {
    ...
  } bs / 1;
  ...
}
```

### Channel substructure

Channel substructure can not be correctly imported to a UML model and is disregarded from. Any substructure elements are imported to the same scope as the channel is defined in.

### Procedure

Procedures without return value are imported as UML operations into the owning class. Return type is set to void. All internal definitions of the SDL procedure are also imported.

**Example 241: Procedure**

**SDL**

```
procedure out_ok;
  start;
    output ok;
    return;
endprocedure;
```

**UML**

```
void out_ok() statemachine {
  start {
    ^ ok();
    return;
  }
}
```

Procedure formal parameters are imported correspondingly. If the type of parameter is mapped to UML class, composite aggregation kind is set for this parameter.

**Example 242: Procedure formal parameters**

**SDL**

```
newtype S struct
  x Integer;
endnewtype;
procedure p; fpar  a Integer, b S;
  start;
    task x := a;
    task y := b;
    return;
  endprocedure p;
```

**UML**

```
class S {
  public Integer x;
}
void p(in Integer a, in part S b)
statemachine {
  start {
    x = a;
    y = b;
    return ;
  }
}
```

All operation parameters are imported to UML Parameters. If you open Properties for a parameter, it has a “Direction” feature that can be one of the following: “In”, “Out”, “In/Out” or “Return”.

Procedure return is imported from operation formal parameter list as an attribute with “Return” direction.

An external procedure is imported to a UML external operation (compare with “External definitions” on page 691).

**Example 243: External procedure**

**SDL**

```
procedure prd1;
  fpar a Integer, in/out b Integer;
  returns Integer;
external;
```

**UML**

```
extern Integer prd1( in Integer a, inout Integer b );
```

For specific SDL procedures in SDL it is allowed to use the last IN/OUT parameter as procedure return parameter and pass a value to it. When this kind of procedure calls are present in SDL model, the definition of the procedure is modified. There will be one additional IN parameter for procedure result is added to the end of procedure formal list. Procedure calls are also updated so the last actual parameter is used as a variable where the operation result is saved. For the normal calls, default value is passed for the generated last IN parameter.

**Example 244: Procedure return as last IN/OUT parameter**

**SDL**

```
procedure p1; returns res Integer;
    start;
      task res := res + 1;
      return res;
endprocedure;
task x := call p1(); /* normal */
task call p1(x);     /* specific */
```

**UML**

```
Integer p1( in Integer res ) statemachine {
  start {
    res = res + 1;
    return res;
  }
}
x = p1(0); /* normal, 0 - default value for Integer */
x = p1(x); /* specific */
```

The opposite situation is also possible. The last IN/OUT parameter can be used as procedure return, for example, it is allowed to write:

```
task x := call proc(7); instead of call proc(7, x);
```

When there are such procedure calls in the imported SDL model, the procedure call is modified.

**Example 245: Last IN-OUT parameter and no return**

**SDL**

```
procedure proc1;
```

```
fpar in a Integer, in/out b Integer;
start;
  task b := a+10;
  return;
endprocedure;

procedure proc2;
fpar in a Integer; returns Integer;
start;
  return a+10;
endprocedure;

procedure proc3;
fpar in a Integer; returns b Integer;
start;
  task b := a+10;
  return;
endprocedure;

call proc1(7, var1);
call proc2(7, var1);
call proc3(7, var1);

task var2 := call proc1(7);
task var1 := call proc2(7);
task var1 := call proc3(7);
```

**UML**

```
void proc1(in Integer a, inout Integer b) statemachine {
  start {
    b = a + 10;
    return ;
  }
}
Integer proc2(in Integer a) statemachine {
  start {
    return a + 10;
  }
}
Integer proc3(in Integer a) statemachine {
  Integer b;
  start {
    b = a + 10;
    return b;
  }
}

proc1(7, var1);
var1 = proc2(7);
var1 = proc3(7);

proc1(7, var2);
var1 = proc2(7);
var1 = proc3(7);
```

For the procedure call examples where procedure result is used as last in/out parameter (`proc2`, `proc3`) warning messages will be printed.

---

Transformations of the return result and last IN/OUT parameter are not applied to external procedures.

### Virtual and redefined procedures

Virtual procedures can be redefined in the inheriting entity. If redefined procedure does not have explicit ATLEAST constraint or explicit inheritance, generalization to the corresponding virtual procedure (with full qualifier) is generated to UML:

**Example 246: Redefined procedure**

**SDL**

```
process type pt1;
  virtual procedure pr2; fpar in xxx Integer; returns
Natural;
  endprocedure pr2;
endprocess type;

process type pt2 inherits pt1;
  redefined procedure pr2;
  endprocedure pr2;
endprocess type;
```

**UML**

```
active public class pt1 {
  virtual Natural pr2(in Integer xxx, in Natural result)
  statemachine {
  }
}

active public class pt2 : pt1 {
  redefined void pr2() : virtprd::pt1::pr2
  statemachine {
  }
}
```

---

### Remote procedure

Remote procedure calls (RPCs) are handled through ports. An RPC is similar to signal sending to another process that implements a remote procedure (outgoing request for execution of remote procedure).

For remote procedures, interfaces are created. They have the same name as the procedure but a postfix "_I" is added. In a class that realizes an exported procedure, a port is added that realizes this interface. In a class where a procedure is called, a port is added that requires this interface. Generated ports are unique for each remote procedure and are named
`rpc_port_<procedure_name>`

**Example 247: Remote procedure**

**SDL**

```
remote procedure setv;
  fpar in Integer;

process P1(1,1);
  exported procedure setv; fpar in i Integer;
  endprocedure;
endprocess;

process P2 (1, 1);
  imported procedure setv; fpar in Integer;
    ...
    call setv(10);
    ...
endprocess;
```

**UML**

```
interface setv_I {
  void setv( in Integer);
}

part active class P1_T {
  public void setv( in Integer i) statemachine {
    ...
  }
  port rpc_port setv in with setv_I;
} [0..1] P1 / 1;

part active class P2_T {
  ...
  setv(10);
  port rpc_port setv out with setv_I;
} [0..1] P2 / 1;
```

“Exported as p procedure my_x” - this construction means that a procedure is referenced by name 'my_x' inside its visibility scope, but it is renamed during export and referenced by another name from where it is imported. Since a UML operation cannot have two different names, exported-as name is always used for the procedure name in the imported model. Information message is printed.

**Example 248: Exported as remote procedures**

**SDL**

```
remote procedure p; returns Integer;
process p(1,1); signalset;
  exported as p procedure xx; returns Integer;
    ...
  endprocedure;
  ...
  task x := call xx;
  ...
endprocess;
```

**UML**

```
interface p_I {
  Integer p();
}
part active class p_T {
  public Integer p() comment "procedure xx" statemachine
{
    ...
  }
  ...
  x = p();
  ...
} [0..1] p / 1;
```

Messages

```
Information: TSI0200: Importing SDL started
Information: TSI0206: Procedure 'xx' has been imported
under the "exported as" name 'p'
Information: TSI0202: Importing SDL completed
```

# Communication

### Signal, signallist

SDL signals are imported to UML signals. Signals formal parameters are preserved during import. If signal parameter is mapped to UML class, it is imported with composite aggregation.

SDL signallist is imported to UML signallist:

**Example 249: Signal lists and signals**

**SDL**

```
newtype MyStruct struct
  x Integer;
endnewtype;
signal ok, s(Integer, MyStruct);
signallist sl = ok, s;
```

**UML**

```
class S {
  public Integer x;
}
signal ok;
signal s(Integer, part MyStruct);
signallist sl = ok, s;
```

Signal refinement is not imported.

Signal inheritance is preserved during import:

**Example 250: Signal inheritance**

**SDL**

```
signal s(Integer);
signal ss inherits s (Natural);
```

**UML**

```
signal s(Integer);
signal ss(Natural) : s;
```

### Gates

SDL gates are transformed to ports. Signals from gate constraint are imported and inserted to “realized” signal set for incoming signals (in with...), or to “required” signal set for outgoing signals (out with...):

**Example 251: Block with gate**

**SDL**

```
block type bt2;
  gate g in from bt1 with ok;
  gate g2 out with ok;
  process type p2;
    gate gp in with ok; out with ok;
    ...
  endprocess type;
  ...
endblock type;
```

**UML**

```
active public class bt2 {
  port g in with ok;
  port g2 out with ok;
  active public class p2 {
    port gp in with ok out with ok;
    ...
  }
  ...
}
```

### Channels and signal routes

Channels and signal routes are imported as UML connectors. In UML, connectors always end with gates. For each channel or signal route in SDL, gates are generated to UML model. Implicitly generated gates have the same

name as corresponding channel or signal route. There are no name conflicts, because gates and connectors with the same name are generated to different scopes. Connector end points are computed according to the following:

1. If a link is connected to an instance and a gate is specified for connector endpoint, it will be imported as connector end point. The gate itself will be imported to the instance scope.

**Example 252: Signals on a gate**

**SDL**

```
block b;
  signalroute sr from p via g to env with ok;
  ...
  process type pt;
    gate g out with ok;
    ...
  endprocess type;
  process p(1,1):pt;
endblock;
```

**UML**

```
part active class b_T {
  connector sr from p.g to c with ok;
  ...
  active public class pt {
    port g out with ok;
    ...
  }
  part pt [0..1] p / 1;

} b / 1;
```

2. If a channel is connected to the environment (to the border of the diagram), and a gate is specified, it will be imported as connector end point. The gate itself will be imported to the class of instance that contains the link being imported. Signals on the gate will also be imported.
3. If a link is connected to an instance but connected gate is not specified, an implicit gate will be inserted into the instance parent class and it will be given the name of the imported channel or signal route. Signals carried on the link will be added to the created port.

**Example 253: Implicit gate**

**SDL**

```
block GameBlock;
  ...
  signalroute R5 from Main to Game with GameOver;
  process Main;
    ...
  endprocess;

  process Game;
    ...
  endprocess;

endblock;
```

**UML**

```
part active class GameBlock_T {
  ...
  connector R5 from Main.R5 to Game.R5 with GameOver;
  part active class Main_T {
    port R5 out with GameOver;
    ...
  } Main / 1;
  part active class Game_T {
    port R5 in with GameOver;
    ...
  } Game / 1;
} GameBlock / 1;
```

4. If a channel is connected to the environment (to the border of the diagram) and this is not the outermost scope but the link is connected to some other link in the surrounding scope, an implicit gate will be inserted into the instance parent class and it will be given the name of the external link from surrounding entity to which the imported link is connected. Signals carried on the link will be added to the created port.
5. If a channel is connected to the environment (to the border of the diagram) and this is the outermost scope, an implicit gate will be inserted into the instance parent class and it will be given the name “Env”. Signals carried on the link will be added to the created port:

**Example 254: Implicit gate in parent class**

**SDL**

```
system DemonGame;
  channel C1 from env to GameBlock with Newgame, Probe,
Result, Endgame;
  endchannel C1;
  ...
  block GameBlock;
    signalroute R2 from env to Game with Probe, Result;
```

```
      connect C1 and R2;
      ...
      ...
    endblock;
  endsystem;
```

**UML**

```
  package DemonGame {
    active class DemonGame {
      ...
      connector C1 from Env to GameBlock.C1 with Newgame,
  Probe, Result, Endgame;
      port Env in with Newgame, Probe, Result, Endgame;
      part active class GameBlock_T {
        connector R2 from C1 to Game.R2 with Probe,
  Result;
        port C1 in with Probe, Result, Newgame, Endgame;
        ...
      } GameBlock / 1;
    }
```

---

Delaying property for connectors is not transformed as UML do not have delayed property on connectors.

### Connection

Connection itself is not imported. It only influences on the number of gates and the names of gates that are implicitly created for link endpoints ("Channels and signal routes" on page 651).

### Implicit communication gates and links

In SDL some of communication links can be created implicitly. In UML there are no such rules, so all implicit links should be constructed during import. This is done with implicit ports in the SDL import, the connectors themselves are added by the UML semantic analyzer.

For all SDL implicit signal routes and channels, ports with names "`io_port`" are added to the imported model. These ports represent end points for implicit connectors:

**Example 255: Implicit signal routes**

**SDL**

```
  system MySystem;
```

```
  signal ok;
  channel c from b1 to env with ok;
  endchannel;
  block b1;
    process p(1,1); signalset;
      start;
        output ok;
        stop;
    endprocess;
  endblock;
endsystem;
```

**UML**

```
package MySystem {
  active class MySystem {
    connector c from b1.c to Env with ok;
    part active class b1_T {
      part active class p_T {
        statemachine p_T {
          start {
            ^ ok();
            stop;
          }
        }
        port io_port out with ok;
      } [0..1] p / 1;
      port c out with ok;
    } b1 / 1;
    port Env out with ok;
  }
  signal ok;
}
```

---

# Behavior

## State machine

An SDL process is imported to a UML state machine named after the resulting class.

**Example 256: Process to UML state machine** ——————————————————

**SDL**

```
process p(1,1); signalset;
  start;
    output ok;
    stop;
endprocess;
```

**UML**

```
part active class p_T {
  statemachine p_T {
    start {
      ^ ok();
      stop;
    }
  }
} [0..1] p / 1;
```

Procedure is directly mapped to UML procedure with state machine body:

**Example 257: Procedure**

**SDL**

```
block b;
  procedure out_ok; returns Boolean;
    start;
      output ok;
      return true;
  endprocedure;
  ...
endblock;
```

**UML**

```
part active class b_T {
  Boolean out_ok(in Boolean result )
statemachine {
    start {
      ^ ok();
      return true;
    }
  }
} b / 1;
```

SDL textual procedure definitions are mapped to UML procedure with textual body:

**Example 258: Procedure in SDL-PR**

**SDL**

```
block b;
  procedure plus1; fpar x Integer; returns Integer;
  start;
    task x := x + 1;
    join L;
```

```
    connection L : return x;
  endprocedure;
  ...
endblock;
```

**UML**

```
part active class b_T {
  Integer plus1(in Integer, in Integer result ) {
    x = x + 1;
    goto L;
    L : return x;
  }
  ...
} b / 1;
```

---

SDL state machine is directly mapped to UML state machine.

**Example 259: State machine**

**SDL**

```
process Main;
  dcl GameP Pid;
  start;
    nextstate Game_Off;

  state Game_Off;
      input Newgame;
      create Game;
      task GameP := offspring;
      nextstate Game_On;
  endstate;

  state Game_On;
      input Endgame;
      output GameOver;
      task GameP := Null;
      nextstate Game_Off;
  endstate;
endprocess Main;
```

**UML**

```
part active class Main_T {
  Pid GameP;
  statemachine Main_T {
    start {
      nextstate Game_Off;
    }
    state Game_Off;
    state Game_On;
    for state Game_Off;
```

```
      input Newgame() {
        Game = new Game_T();
        GameP = offspring;
        nextstate Game_On;
      }
    for state Game_On;
      input Endgame() {
        ^ GameOver();
        GameP = NULL;
        nextstate Game_Off;
      }
  }
} Main / 1;
```

**Example 260: State machine**

## SDL

```
procedure ReadKeys; fpar in NumberKeys Natural, in/out
KeyData KeyArrayType; returns ReadResultType;
  dcl KeyIndex Natural:=1,
      Key Character;
  start;
    set(KeyTimer);
    nextstate WaitKeyStroke;
  state WaitKeyStroke;
    input KeyStroke(Key);
    task KeyData(KeyIndex) := Key;
    decision KeyIndex >= NumberKeys;
      (false):
        set(KeyTimer);
        task KeyIndex:=KeyIndex+1;
        nextstate WaitKeyStroke;

      (true):
        reset(KeyTimer);
        return Successful;
    enddecision;

    input KeyTimer;
    return TimedOut;
  endstate;
endprocedure ReadKeys;
```

## UML

```
ReadResultType ReadKeys( in Natural NumberKeys, inout
KeyArrayType KeyData) statemachine {
  Natural KeyIndex = 1;
  Character Key;
  start {
    set KeyTimer();
    nextstate WaitKeyStroke;
```

```
    }
    state WaitKeyStroke;
    for state WaitKeyStroke;
      input KeyStroke(Key) {
        KeyData[KeyIndex] = Key;
        switch (KeyIndex >= NumberKeys){
          case ==false : {
             set KeyTimer();
             KeyIndex = KeyIndex + 1;
             nextstate WaitKeyStroke;
          }
          case ==true : {
             reset KeyTimer();
             return Successful;
          }
        }
        input KeyTimer() {
        return TimedOut;
      }
  }
```

### Procedure call

Procedure call is mapped to similar UML operation call. One difference between those calls is the handling of omitted parameters. In SDL it is allowed to omit any IN parameters from procedure call, but in UML it is allowed to omit only the last parameter(s) from actual parameter list and only if default values are defined in procedure prototype. Implicit default values are not inserted to imported procedure prototype; instead, default values are inserted directly to the procedure call.

The following implicit default values are used:

| SDL type | UML default value |
|---|---|
| Integer, Duration, Octet | 0 |
| Boolean | false |
| Character | '0' |
| Charstring | “” |
| Real, Time | 0.0 |
| Pid | NULL |
| Bit | ‘\0’ |

| SDL type | UML default value |
|---|---|
| `Bit_string` | `''B` |
| `Octet_string` | `'00'H` |
| User defined datatype with literals | First literal in datatype definition |

For all other types a variable is generated to the surrounding scope and it is passed to the procedure call. The name of the variable is `default_<number>` and it is not initialized.

**Example 261: Procedure call with omitted parameters**

**SDL**

```
newtype S struct
  x Integer;
endnewtype;
procedure p; fpar  a Integer, b S;
endprocedure p;
...
dcl Svar S;
call p();
call p(1,);
call p(, Svar);
```

**UML**

```
class S {
  public Integer x;
}
void p(in Integer a, in part S b) statemachine {
}
...
part S Svar;
{
  part S default_0001;
  p(0, default_0001);
}
{
  part S default_0001;
  p(1, default_0001);
}
{
  p(0, Svar);
}
```

### Create statement

The mapping for create statements depends on the type of the created process. If maximum number of process instances is 2 or more then this is a multi-instance process and creating is mapped to the appending of the new instance of an active class to the set of class instances.

**Example 262: Create of multi-instance process**

**SDL**

```
process MyProcess(1,5);
endprocess MyProcess;

create MyProcess;
```

**UML**

```
part active class MyProcess_T {
} [0..5] MyProcess / 1;

MyProcess.append(new MyProcess_T());
```

If a process can not have more than one instance, create statement is mapped to the direct call of operator “new”:

**Example 263: Create of single-instance process**

**SDL**

```
process Game;
endprocess Game;

create Game;
```

**UML**

```
part active class Game_T {
} Game / 1;

Game = new Game_T();
```

When maximum number of process instances is an external constant, this process is considered to be multi-instance and create statement is mapped to append call:

**Example 264: Create of a process with external instance number**

**SDL**

```
synonym Max Integer = external 'C';
process type pt;
  create p;
endprocess type;
process p(1, Max) : pt;
```

**UML**

```
const Integer extern Max;
active public class pt {
  p.append(new pt());
}
part pt [0..Max] p / 1;
```

For each omitted actual parameter in SDL create statement, corresponding default value is inserted into the constructor call.

**Example 265: Create statements**

**SDL**

```
process pr(1,6);fpar a,b Integer; signalset s,ss;
    …
    create pr();
    create pr(,);
    create pr(2);
    create pr(2,);
    create pr(,2);
    …
  endprocess;
```

**UML**

```
part active class pr_T {
    …
    statemachine pr_T( Integer fpar_a = 0,
       Integer fpar_b) {
      …
      pr.append(new pr_T(0, 0));
      pr.append(new pr_T(0, 0));
      pr.append(new pr_T(2, 0));
      pr.append(new pr_T(2, 0));
      pr.append(new pr_T(0, 2));
    }
    …
  } [1..6] pr / 1;
```

### Output statement

Output statement is mapped directly to UML signal sending statement. Omitted parameters in signal output are treated in the same way as omitted parameter in procedure call:

**Example 266: Signal output**

**SDL**

```
signal ss( Integer, Integer );
  ...
  output ss(5);
  output ss(6,);
  output ss();
  output ss(,7);
```

**UML**

```
signal ss(Integer, Integer);
^ ss(5, 0);
^ ss(6, 0);
^ ss(0, 0);
^ ss(0, 7);
```

Output via-list is imported to the corresponding UML via list. If the same entity is referenced several times in via list, it is imported only once.

Output via non-local ports or links is not supported in UML, and all missing output and input ports are implicitly generated to UML model. Output via SDL channel is not imported to UML. Qualifiers on via elements are also ignored.

**Example 267: Output via**

**SDL**

```
system OutputVia;
  signal ok;
  channel c1 from t to env with ok;endchannel;
  block t;
    signalroute sr1 from p1 to env with ok;
    connect c1 and sr1;
    process p1(1,1);
      start;
      output ok via c1,c1,c1,sr1,sr1;
      output ok via <<system OutputVia/block t>> sr1;
      stop;
    endprocess p1;
```

```
  endblock t;
endsystem;
```

**UML**

```
package OutputVia {
  active class OutputVia {
    part active class t_T {
        part active class p1_T {
            statemachine p1_T {
                start {
                    ^ ok() via sr1;
                    ^ ok() via sr1;
                    stop;
                }
            }
            port sr1 out with ok;
        } [0..1] p1 / 1;
        connector sr1 from p1.sr1 to c1 with ok;
        port c1 out with ok;
    } t / 1;
    connector c1 from t.c1 to Env with ok;
    port Env out with ok;
  }
  signal ok;
}
```

For output to Pid-expression, full qualifier is added to all output items. Qualifiers are needed since output item is resolved in the context of to-expression, but for Pid-expression there is no context.

**Example 268: Output to Pid-expression**

**SDL**

```
system OutputTo;
  signal ok;
  syntype MyPid = Pid endsyntype;
  block t;
    signal pong;
    ...
    process p2(1,1);
      dcl s MyPid;
      ...
      task s := sender;
      output pong to s;
      ...
      output pong to sender;
      ...
    endprocess p2;
  endblock t;
endsystem;
```

**UML**

```
package OutputTo {
  active class OutputTo {
    part active class t_T {
      signal pong;
      ...
      part active class p2_T {
        MyPid s = NULL;
        ...
        s = sender;
        ^
s.ImportedSDLDefinitions::OutputTo::OutputTo::t_T::pong(
);
        ...
        ^
sender.ImportedSDLDefinitions::OutputTo::OutputTo::t_T::
pong();
      } [0..1] p2 / 1;
    } t / 1;
  }
  signal ok;
  syntype MyPid = Pid;
}
```

## Input statement

An input statement is mapped directly to a UML signal receipt statement. For each omitted parameter in a signal input a variable is generated to the surrounding scope and passed to the input expression. The name of the variable is `default_<number>` and it will not be initialized.

**Example 269: Signal input**

**SDL**

```
signal s(Natural),ss(Integer,Integer);
block b;
  process p(1,6); signalset s,ss;
    ...
    state A;
      input s(), ss();
      nextstate A1;

    state A1;
      input ss(v);
      nextstate A2;

    state A2;
      input ss(,v);
```

```
        nextstate A3;

      state A3;
        input ss(v,);
        stop;
      endstate;
    endprocess;
  endblock;
```

**UML**

```
signal s( Natural);
signal ss( Integer, Integer);
part active class b_T {
  part active class p_T {
    private Integer default_0006;
    private Integer default_0005;
    private Integer default_0004;
    private Integer default_0003;
    private Integer default_0002;
    private Natural default_0001;
    statemachine p_T() {
      state A; state A1; state A2; state A3;
      for state A;
        input s(default_0001),
              ss(default_0002, default_0003) {
          nextstate A1;
      }
      for state A1;
        input ss(v, default_0004) {
          nextstate A2;
      }
      for state A2;
        input ss(default_0005, v) {
          nextstate A3;
      }
      for state A3;
        input ss(v, default_0006) {
          stop;
      }
    }
  } [0..6] p / 1;
} b / 1;
```

---

### Informal statement

In SDL, some actions can be specified informally, inside informal statements. These statements are mapped to UML comments attached to empty statements. Informal text is inserted into the comment string. Surrounding quotes are removed:

**Example 270: Informal statement**

**SDL**

```
call p(1);
  decision 'Decide?';
  ('my decision') :
    task 'do something';
    task 'do it once again';
    output ok;
    stop;
  else :
    task 'do something else';
    output ok;
  stop;
enddecision;
```

**UML**

```
p(1);
  switch ("Decide?") {
  case =="my decision" :
  {
    comment "do something";
    comment "do it once again";
    ^ ok();
    stop;
  }
  default :
  {
    comment "do something else";
    ^ ok();
    stop;
  }
  }
```

**Note**

*When C code is generated from these informal statements, nested comments will be generated that will cause fail in the build process. This can be avoided by placing informal text inside an action symbol.*

# Code generation directives

Tau SDL Suite has support for proprietary extensions using special comments - also know as directives to the code generator. These directives can contain C code that should be inserted in the generated C code.

### #CODE directive

C code may be included in SDL tasks by using the #CODE directive. This directive has the syntax `/*#CODE C code */` and it can be placed near SDL task statements (as defined in the SDL Suite documentation). This directive is mapped to a UML informal action. C code inside the directive is inserted into the UML informal action without any changes:

**Example 271: #CODE directive**

**SDL**

```
start;
  task
/*#CODE
testvalue := testvalue + 1;
*/
  testvalue := testvalue+2
/*#CODE
testvalue = testvalue+3;
*/
  ,
/*#CODE
testvalue = testvalue+4;
*/
  testvalue := testvalue+5
/*#CODE
testvalue = testvalue+6;
*/
  ;
/*#CODE
#(testvalue) = #(testvalue)+7;
*/
  task 'stop the kernel' /*#CODE  SDL_Halt(); */;
```

**UML**

```
start {
  [[ testvalue = testvalue+1; ]]
  [[ testvalue = testvalue+7; ]]
  testvalue = testvalue + 2;
  [[ testvalue = testvalue+3; ]]
  [[ testvalue = testvalue+4; ]]
  testvalue = testvalue + 5;
  [[ testvalue = testvalue+6; ]]
  comment "stop the kernel";
  [[ SDL_Halt(); ]]
}
```

### #ADT directive

Tau SDL Suite C Code generator offers a possibility to include implementations written in C of the operator and literal functions. They are introduced with #ADT directives that are recognized when placed immediately before the reserved word ENDNEWTYPE (or ENDSYNTYPE).

These directives are also imported and stored in UML model in ADT stereotypes (defined in the **CApplication** customization module) that contains a text string attribute with the body of ADT directive.

If an #ADT directive is used to implement operators from a newtype defined with generator transformations, it will not be correctly imported. Operators for this datatype are imported to the same scope as where the datatype is defined and the datatype itself is imported to a UML syntype (compare with "Newtype with generator transformation, adding operator" on page 684). The corresponding ADT stereotype will be attached incorrectly and disregarded during code generation.

**Example 272: Data type implementation for newtype**

**SDL**

```
process Central;
  NEWTYPE CardBaseType
    Array(CardBaseIndexType, CardRecordType);
    OPERATORS
      Full: CardBaseType -> Boolean;
      Register: CardBaseType, CardType, CardRecordType -
> CardBaseType;
      Validate: CardBaseType, CardRecordType ->
ResultType;
/*#ADT(B)
#BODY
...
*/
  ENDNEWTYPE CardBaseType;
  …
endprocess Central;
```

**UML**

```
part active <<ADT(.bodyText = "#ADT(B)\n#BODY …",
generateBodyText = true.)>> class Central_T {
  static <<External="true">> Boolean Full( CardBaseType)
comment "Implemented in #ADT #BODY-section";
  static <<External="true">> CardBaseType Register(
CardBaseType, CardType, part CardRecordType) comment
"Implemented in #ADT #BODY-section";
  static <<External="true">> ResultType Validate(
```

```
CardBaseType, part CardRecordType) comment "Implemented
in #ADT #BODY-section";
  syntype CardBaseType = Array<CardBaseIndexType,
Value<CardRecordType> >;
  ...
}
```

### #INCLUDE directive

Tau SDL Suite directive `#INCLUDE 'file'` makes the SDL Analyzer include the contents of `'file'` to the place of the directive. Included definitions are imported to UML with the exception of definitions included from any of the predefined files:

```
BasicC++Types.pr
BasicCTypes.pr
C++Pointer.pr
CPointer.pr
CharConvert.pr
```

**Example 273: Include of predefined files**

**SDL**

```
system Mysystem;
  /*#INCLUDE 'BasicC++Types.pr'*/
  /*#INCLUDE 'C++Pointer.pr'*/
  /*#INCLUDE 'CharConvert.pr'*/
  newtype Ptr Ref(Integer) endnewtype;
  ...
endsystem;
```

**UML**

```
active class Mysystem {
  syntype Ptr = CPtr<Integer>;
  ...
}
```

### Restrictions for code directives

Extended form of `#CODE` directive with `#TYPE`, `#HEADING` and `#BODY` sections is not supported.

Only directives inside comments are imported in the described way, other cases, for example, `x + #CODE('a')`, is not supported.

# Data types

## Simple data types

SDL predefined simple data types and operations are mapped to the corresponding UML basic data types and operations according to the following tables. Predefined data types and constants mapping:

| SDL | UML |
|---|---|
| boolean | Boolean |
| true | true |
| false | false |
| character | Character |
| charstring | Charstring |
| ia5string | IA5string |
| numericstring | NumericString |
| duration | Duration |
| time | Time |
| bit | Bit |
| octet | Octet |
| string | String |
| powerset | PowerSet |
| bag | Bag |
| pid | Pid |
| null | NULL |
| integer | Integer |
| float | float |
| visiblestring | VisibleString |
| printablestring | PrintableString |
| real | Real |
| natural | Natural |
| plus_infinity | PLUS_INFINITY |

| SDL | UML |
|---|---|
| minus_infinity | MINUS_INFINITY |
| array | Array |
| oref | ORef |
| bit_string | BitString |
| octet_string | OctetString |
| own | Own |
| object_identifier | ObjectIdentifier |
| any_type | AnyType |

### Predefined operations

| SDL | UML |
|---|---|
| /= | != |
| // | + |
| = | == |
| not | not |
| and | and |
| or | or |
| xor | xor |
| mod | mod |
| rem | rem |
| in | in |
| num | num |
| chr | chr |
| mkstring | mkstring |
| length | length |
| first | first |
| last | last |
| substring | substring |

| SDL | UML |
|---|---|
| append | append |
| bitstr (that returns bit_string) | String2BitString |
| bitstr (that returns octet) | String2Octet |
| bitstr (that returns octet_string) | bitstr |
| hexstr (that returns bit_string) | HexString2BitString |
| hexstr (that returns octet) | HexString2Octet |
| hexstr (that returns octet_string) | hexstr |
| shiftl | shiftl |
| shiftr | shiftr |
| i2o | I2O |
| o2i | O2I |
| incl | incl |
| take | take |
| makebag | makebag |
| fix | fix |

In SDL there are two forms of `incl` and `del` operations for SDL `Bag` and `Powerset`:

```
incl : Itemsort,  Powerset  -> Powerset;
incl : Itemsort,  in/out Powerset;
```

There is only one form of `incl` and `del` operations in UML. The short form without a return value is not supported in UML, so all references to this are transformed so the second (`in/out`) parameter is used on the left-hand side of an assignment expression.

**Example 274: Short form for Powerset operation**

**SDL**

```
newtype Power
  Powerset (Integer)
endnewtype;
task incl(1, pow1);
```

**UML**

```
syntype Power = PowerSet<Integer>;
pow1 = Power::incl(1, pow1);
```

### Predefined C++ types

Simple data type "unsigned_char" is a part of SDL package representing C++ types. This type is defined as a synonym of SDL type "Octet". It is mapped directly to UML type "Octet":

**Example 275: Mapping for SDL** unsigned_char

**SDL**

```
newtype MyType Ref(unsigned_char) endnewtype;
syntype Myunsigned = unsigned_char endsyntype;
dcl x unsigned_char;
dcl y Myunsigned;
dcl p MyType;
dcl i Integer;
task x := p*>;
task y := x;
task i := O2I(y);
task x := I2O(i);
```

**UML**

```
syntype MyType = CPtr<Octet>;
syntype Myunsigned = Octet;
Octet x;
Myunsigned y;
MyType p;
Integer i;
x = p.GetValue();
y = x;
i = O2I(y);
x = I2O(i);
```

A special SDL type representing pointer to void "ptr_void" is mapped to UML type 'void*'. A special SDL type representing C string "ptr_char" is mapped to UML type 'char*'.

In the SDL Suite there are special operators that are generated to the imported enum types. These operators are called EnumToInt and IntToEnum and they are used for transformations between integer and enumerated values. EnumToInt and IntToEnum are imported to the call of cast<> operator in UML:

**Example 276: EnumToInt and IntToEnum**

**SDL**

```
dcl i Integer;
dcl e MyEnum := a;
task i := EnumToInt( e );
task e := IntToEnum( i );
```

**UML**

```
Integer i;
MyEnum e = a;
i = cast<Integer>(e);
e = cast<MyEnum>(i);
```

Special import rules are applied when casting types from `ptr_void` to `Ref<T>`, and from `ptr_char` to `Charstring`:

- `cast(Charstring)` to `ptr_char`: this is omitted because there is implicit conversion from `Charstring` to `'char*'` in UML
- `cast(ptr_char)` to `Charstring` is imported to a call of the operator `'char*'.ToString()`;
- `cast(Ref<T>)` to `ptr_void` is imported to `cast<'void*'>( CPtr<T> )`
- `cast<ptr_void>` to `Ref<T>` is imported to UML cast from `'void*'` to `CPtr<T>`

**Example 277: Special import rules for SDL cast**

**SDL**

```
newtype Ptr Ref(Integer) endnewtype;
dcl Invar  Integer;
dcl Outvar Ptr;
dcl Vs     ptr_void;
dcl pc     ptr_char;
dcl str    Charstring;
task Outvar := &Invar,
     Vs := cast(Outvar),
     Outvar := cast(Vs);
task str := 'My String',
     pc := cast(str),
     str := cast(pc);
```

**UML**

```
syntype Ptr = CPtr<Integer>;
```

```
Integer Invar;
Ptr Outvar;
'void*' Vs;
'char*' pc;
Charstring str;
Outvar = GetAddress<Integer>(Invar);
Vs = cast<'void*'>(Outvar);
Outvar = cast<Ptr>(Vs);
str = "My String";
pc = str;
str = pc.ToString();
```

The following table summarizes special mapping rules for data types and operators used to support external C++ types in SDL:

| **SDL** | **UML** |
|---|---|
| **Data types** | |
| unsigned_char | Octet |
| ptr_void | 'void*' |
| ptr_char | 'char*' |
| **Operators** | |
| EnumToInt | cast<> |
| IntToEnum | cast<> |

### Conflicts with UML Predefined types

SDL user-defined type can conflict with the UML predefined type after import. This simply means that imported user-defined type will have the same name as UML type from one of the predefined packages `TTDCppPredefined`, `TTDRTTypes` and `Predefined`.

There is an option **Import user-defined types that conflict with UML predefined types** that controls the handling of conflicting types during SDL import. By default, this option is set to false and conflicting types are not imported. An information message will alert you on this situation:

```
Information: TSI0209: SDL user-defined type conflicts
with UML predefined type 'short int' from package
'TTDCppPredefined'. SDL type is NOT imported.
```

If the option is true, conflicting types are imported and a warning is generated:

```
Warning : TSI0210: SDL user-defined type conflicts with
UML predefined type 'short int' from package
'TTDCppPredefined'. Consider revising imported type.
```

### Structured data types

SDL struct type is mapped to UML class with public fields.

**Example 278: Struct**

**SDL**

```
newtype n struct
  f1 Integer;
  f2 Boolean;
endnewtype;
```

**UML**

```
class n {
  public Integer f1;
  public Boolean f2;
}
```

**Note**

*Comparison operation for structured data types: v1 == v2, where v1 and v2 are values of a struct - this operation is comparing references, but not nested values of structured type.*

Optional fields are mapped to public fields with `[0..1]` Multiplicity. Fields with one type defined with field name list in SDL are mapped to separate field declarations on different lines in UML. Inline field initialization is mapped to similar initialization in UML.

**Example 279: Struct with optional fields**

**SDL**

```
newtype str struct
  a, b Integer;
  c    Boolean optional;
  d    str2 optional;
  e    Charstring := 'telelogic';
  f    arr3 := (. 11 .);
endnewtype;
```

**UML**

```
class str {
  public Integer a;
  public Integer b;
  public Boolean [0..1] c;
  public part str2 [0..1] d;
  public Charstring e = "telelogic";
  public arr3 f = arr3 (.11.);
}
```

---

SDL `choice` is mapped to UML `choice` with public fields.

**Example 280: SDL choice**

**SDL**

```
newtype c choice
  f1 Integer;
  f2 Boolean;
endnewtype;
```

**UML**

```
choice c {
  public Integer f1;
  public Boolean f2;
}
```

---

The general UML form of choice creation with initialization of a specific field is the following

```
<choice name> (. <field name> = <expression> .)
```

**Example 281: SDL choice with initialization**

**SDL**

```
newtype choice_type choice
  a Integer;
  b Boolean;
endnewtype;

dcl an_int choice_type:= a : 1;
/* Initialization of field a by value 1 */
```

**UML**

```
choice choice_type
  {
    public Integer a;
    public Boolean b;
  }

choice_type an_int = choice_type (. a = 1 .);
```

Other SDL data types are mapped to UML datatypes and all operators are also mapped. For such data types basic operation prototypes are also inserted (except data types that contain only literals).

**Example 282: Operators**

**SDL**

```
newtype dummy
  operators
    op: Charstring -> Charstring;
  operator op; fpar i Charstring; returns Charstring;
    start;
      return call p( i // 'def' );
  endoperator;
endnewtype;
```

**UML**

```
datatype dummy {
  static <<IsQuery="true">> Charstring op( Charstring i)
{
    return p(i + "def");
  }
  extern public static dummy '\='( dummy, dummy);
  extern public static Boolean '=='( dummy, dummy);
  extern public static Boolean '!='( dummy, dummy);
}
```

### Default values in SDL newtype and syntype

Default values in SDL newtype and syntype are supported in the SDL import. If default value is specified for a type, it is used to initialize variables and fields of this type and it is passed to procedure calls, signals, etc. if the actual value is omitted.

**Example 283: Default values in SDL newtype and syntype**

## SDL

```
newtype lights
  literals red, yellow, green
  default  yellow
endnewtype;

syntype __lights = lights
  default  green
endsyntype;

newtype X struct
  x Integer;
  y lights;
endnewtype;

signal ok(lights);

procedure out_ok; fpar ii Integer, ss __lights; returns
Boolean;
  start;
    output ok;
    return true;
endprocedure;

dcl v Boolean;
dcl s1 lights;
dcl s2 lights := red;
dcl s3 __lights;
dcl s4 __lights := red;

input ok(s1);
task v := "not"(call out_ok(1));
```

## UML

```
enum lights { red, yellow, green }
syntype __lights = lights;
class X {
  public Integer x;
  public lights y = yellow;
}
signal ok( lights );
Boolean out_ok( in Integer ii, in __lights ss, in
Boolean result )
statemachine {
  start {
    ^ ok(yellow);
    return true;
  }
}

lights s1 = yellow;
```

```
lights s2 = red;
__lights s3 = green;
__lights s4 = red;
input ok(s1) {
v = not out_ok(1, green, false);
```

### Checking presence

Several implicit `Present` operators are available for SDL types. They allow to check presence of choice and optional structure fields.

Implicit boolean operator `<field name>present(<variable>)` applied to optional structure fields is imported to an expression of the form:

```
<variable>.<field name> != NULL
```

**Example 284: Checking presence for optional struct field**

**SDL**

```
newtype seq struct
  a Integer;
  r Integer;
  z Integer optional;
endnewtype

dcl  v seq;
dcl  b Boolean;
task b := zPresent( v );
```

**UML**

```
class seq {
  public Integer a;
  public Integer r;
  public Integer [0..1] z;
}

part seq v;
Boolean b;
b = (v.z != NULL);
```

The implicit boolean operator `<field name>present(<variable>)` applied to a choice field is transformed to an operator call on the form

```
<variable>.IsPresent("<field name>")
```

**Example 285: Checking presence for choice field**

**SDL**

```
newtype cho choice
  x Integer;
  y Boolean;
endnewtype;

dcl  myc cho;
dcl  b Boolean;
task b := yPresent( myc );
```

**UML**

```
choice cho {
  public Integer x;
  public Boolean y;
}

part cho myc;
Boolean b;
b = myc.IsPresent("y");
```

The implicit `<ChoiceName>present` type with literals equal to choice field names is mapped to a UML explicit enumerated type `<ChoiceName>_enum`. All references to `<ChoiceName>present` are renamed to `<ChoiceName>_enum`.

**Example 286: Enumerated type for handling present field in choice type**

**SDL**

```
newtype MyChoice choice
  bfield Boolean;
  ifield Integer;
endnewtype;
```

**UML**

```
choice MyChoice {
  public Boolean bfield;
  public Integer ifield;
}
enum MyChoice_enum {
  bfield,
  ifield
}
```

The implicit choice field `present` is typed by `<ChoiceName>present`. It holds the name of the active choice field. By checking the `present` field it is possible to find out which field is active.

"=" and "/=" boolean expressions referencing implicit choice field "`present`" are mapped to corresponding checks for `<variable>.IsPresent("<field name>")`.

**Example 287: Checking implicit field present**

**SDL**

```
newtype cho choice
  x Integer;
  y Boolean;
endnewtype;

dcl  myc cho;
dcl  b Boolean;
task b := ( myc!present /= y );
task b := ( myc!present = y );
```

**UML**

```
choice cho {
  public Integer x;
  public Boolean y;
}

part cho myc;
Boolean b;
b = (not myc.IsPresent("y"));
b = (myc.IsPresent("y"));
```

The last transformation is applied only if there is no user defined explicit field called "`present`" in the choice type.

---

All other references to implicit choice field `present` are mapped to a conditional expression checking presence for all choice fields.

**Example 288: Reference to implicit choice field "present"**

**SDL**

```
newtype MyChoice choice
  bfi Boolean;
  ifi Integer;
endnewtype;
```

```
dcl ch MyChoice;
dcl v  MyChoicepresent;
task ch!ifi := 10;
task v := ch!present;
task v := bfi;
```

**UML**

```
choice MyChoice {
  public Boolean bfi;
  public Integer ifi;
}
enum MyChoice_enum {
  bfi,
  ifi
}
MyChoice ch;
MyChoice_enum v;
ch.ifi = 10;
v = ch.IsPresent("bfi") ? bfi : ifi;
v = MyChoice_enum::bfi;
```

---

### Generators

Predefined SDL generators are imported. They are mapped to corresponding template instantiations. All template function calls (like `mkstring`) are prefixed by type name qualifier.

**Example 289: Newtype with generator**

**SDL**

```
newtype t
  String( Integer, Empty )
endnewtype;

task v2 := mkstring( 1 );
```

**UML**

```
syntype t = String<Integer>;
v2 = t::mkstring(1);
```

---

### Newtype with generator transformation, adding operator

Operators from newtypes with generator transformations are mapped to the same scope where the datatype is defined.

**Example 290: Newtype with generator transformation**

**SDL**

```
newtype t4
  array( Index, Integer)
  operators
    op4 : Integer -> Integer;
endnewtype;
```

**UML**

```
public static <<IsQuery="true">> Integer op4( Integer) ;
syntype t4 = Array<Index, Integer> ;
```

If one of the operators from such newtype is implemented using the #ADT directive, the corresponding stereotype will be attached to the wrong node in the imported model, and disregarded during code generation.

### Newtype with literals

SDL newtype containing only literals is mapped to UML enum type. User-defined literals are prefixed with type qualifier in all places they are used.

**Example 291: Newtype with literals**

**SDL**

```
newtype SomeType
  literals Some1, Some2, Some3
    default Some2
endnewtype;

newtype SomeType2
  inherits SomeType
endnewtype;

dcl var0 SomeType;
dcl var1 SomeType2;
task var1 := Some1;
```

**UML**

```
enum SomeType {
  Some1,
  Some2,
  Some3
}
datatype SomeType2 : SomeType
{
```

```
  public <<External="true">> SomeType2( SomeType);
  public static <<External="true">> SomeType2 '\='(
SomeType2, SomeType2);
  public static <<External="true">> Boolean '=='(
SomeType2, SomeType2);
  public static <<External="true">> Boolean '!='(
SomeType2, SomeType2);
}
SomeType var0 = SomeType::Some2;
SomeType2 var1;
var1 = SomeType2::Some1;
```

### Generator Ref and operators “&” and “*>”

Predefined generator Ref representing C++ pointer is imported to UML template datatype `CPtr<T>` from `TTDCppPredefined` profile.

Operator “`&`” is mapped to the call of template function `GetAddress` defined in CPtr datatype. Operator “`*>`” is mapped to the call of the `GetValue` function.

**Example 292: Operators “&” and “*>” mapping**

**SDL**

```
newtype Iptr
  Ref(Integer);
endnewtype;

dcl a Integer;
dcl p Iptr;
task p := &a;
task a := p*>;
```

**UML**

```
syntype Iptr = CPtr<Integer>;
Integer a;
Iptr p;
p = GetAddress<Integer>(a);
a = p.GetValue();
```

### Data type inheritance

Data type inheritance is mapped correspondingly. For all parents in inheritance chain conversion operators to parent types are generated to the inheriting datatype.

**Example 293: Datatype inheritance**

**SDL**

```
newtype SomeType
  literals Some1, Some2, Some3
    default Some2
endnewtype;

newtype SomeType2
  inherits SomeType
endnewtype;

newtype SomeType3
  inherits SomeType2
endnewtype;
```

**UML**

```
enum SomeType {
  Some1,
  Some2,
  Some3
}
datatype SomeType2 : SomeType
{
  public <<External="true">> SomeType2( SomeType);
  public static <<External="true">> SomeType2 '\='(
SomeType2, SomeType2);
  public static <<External="true">> Boolean '=='(
SomeType2, SomeType2);
  public static <<External="true">> Boolean '!='(
SomeType2, SomeType2);
}
datatype SomeType3 : SomeType2
{
  public <<External="true">> SomeType3( SomeType2);
  public <<External="true">> SomeType3( SomeType);
  public static <<External="true">> SomeType3 '\='(
SomeType3, SomeType3);
  public static <<External="true">> Boolean '=='(
SomeType3, SomeType3);
  public static <<External="true">> Boolean '!='(
SomeType3, SomeType3);
}
```

## Variables

SDL variable declarations are mapped to UML variable definitions without explicitly specified visibility.

**Example 294: Variables**

**SDL**

```
dcl v1 Integer;
dcl v2 Boolean;
```

**UML**

```
Integer v1;
Boolean v2;
```

## Viewed/revealed variables

Viewed and revealed variables are handled through ports. For each viewed variable (`name`) an interface called `<name>_var_I` is created. An implicit port `revealed_port_<name>` is generated to the class where revealed variable is defined. To each class where revealed variable is accessed an implicit port `viewed_port_<name>` is generated. Implicit connectors between class instances with revealed variable and class instances referencing this variable are generated:

**Example 295: Viewed/revealed variables**

**SDL**

```
block b;
  process p;
    dcl revealed j boolean;
  endprocess;

  process type pt2;
    viewed j boolean;
    dcl i boolean;
    task i := view (j);
  endprocess type;

  process p2(1,1) : pt2;
endblock;
```

**UML**

```
part active class b_T {
  interface j_var_I {
    Boolean j;
  }
  part active class p_T {
    public Boolean j;
    port revealed_port_j in with j_var_I;
```

```
  } p / 1;
  active public class pt2 {
    port viewed_port_j out with j_var_I;
    Boolean i;
    virtual statemachine pt2 {
      start {
        i = j;
        stop;
      }
    }
  }
  part pt2 [0..1] p2 / 1;
  connector p2_p_j_var from p2.viewed_port_j to
p.revealed_port_j with j_var_I;
} b / 1;
```

### Remote variables

Remote variables are handled through ports, similar to Remote procedure.

## General rules

### Process formal parameters

Process type formal parameters are imported as formal parameters of the state machine, which is added into the class definition that corresponds to the process type. Parameter names are prefixed with "`fpar_`".

For each process type formal parameter, an attribute is added to the class definition and these attributes are initialized in the first statements of the state machine:

**Example 296: Process formal parameters**

**SDL**

```
process type pt; fpar a, b Integer;
   …
  endprocess type;
```

**UML**

```
active public class pt {
  …
  virtual statemachine pt( Integer fpar_a,
                           Integer fpar_b) {
    start {
      pt::b = fpar_b;
```

```
        pt::a = fpar_a;
        …
      }
      …
    }
    Integer a = 0;
    Integer b = 0;
  }
```

Process formal parameters are added to the inline class that is created for the implicit process type.

### Qualifiers

Qualifiers are transformed to UML qualifiers, but the specification of the kind of an entity (for example a system or a process) is lost.

### Virtuality

Virtuality is imported.

### Comments

SDL comments are imported to UML model.

Textual SDL comments (comment '…';) are attached to the corresponding model element after import.

C-style comments from textual diagrams and symbols are imported, but their location is not always preserved and some of them are lost. Most of C-style comments from task symbols are lost. The result will be different depending on the source of the imported SDL (SDL Suite or ObjectGeode).

**Example 297: Importing C-style comments from SDL Suite**

SDL

```
signal sig1 /* 1 */;
signal sig2(Integer) /* 2 */;
signal sig3(Boolean /* 3 */) /* 4 */; /* 5 */
```

UML

```
signal sig1 comment " 1 ";
signal sig2( Integer);
```

```
signal sig3( Boolean) comment " 5 ";
```

**Example 298: Importing C-style comments from ObjectGeode**

For ObjectGeode import, the situation is that if a comment is written in the middle of the element definition, it is inserted before or after this definition in the imported UML.

SDL

```
/* 1 */
signal /* 2 */ s1;
signal s2 /* 3 */; /* 4 */
```

UML

```
/* 1 */
/* 2 */
signal s1;
/* 3 */
/* 4 */
signal s2;
```

### External definitions

SDL definitions marked `external 'C'` or `external 'C++'` are not imported to UML by SDL import, such definitions should be imported to UML with C/C++ Import.

In the case of an ignored definition there will be an information message, for example:

```
Information: TSI0211: External 'C' or 'C++' procedure
'DEK_c_AlgorithmPriority' has been ignored. Use C/C++
import to map C and C++ definitions to UML.
```

All other SDL definitions marked as external are imported to UML external definitions.

# Restrictions on SDL Import

## General SDL language restrictions

### Semantically correct SDL

An SDL specification must be a complete, semantically correct SDL system in order to be imported. Incomplete, or incorrect specifications cannot be imported.

### Case sensitivity

The SDL Z.100 recommendation exists in two flavours with respect to case sensitivity. Case sensitivity has been introduced in later versions of SDL and hence SDL systems designed with recent versions of SDL tools, such as SDL Suite 4.4 and later, can be designed using the SDL tool in case sensitive mode.

UML is a case sensitive language. SDL import operates in a case preserving mode and therefore it is required that the source SDL system is correct with respect to upper and lower case, otherwise the resulting UML may be semantically incorrect due to definitions and references that do not match, or keywords that are not be properly recognized.

This means that:

- SDL definitions and references to definitions must be written with case sensitivity through the SDL system.
- All SDL keywords must use upper or lower cases in a consistent fashion.

There may be a need to modify the system prior to import, in order to apply upper and lower cases in a consistent fashion. Refer to the SDL tool user documentation for support on how to convert an SDL system to case sensitive SDL.

# Not supported SDL language concepts

## Virtual process type definitions

Virtual process type definitions are imported, but use of virtuality for corresponding active classes is not supported in UML. This is a constraint on UML models.

Virtual process types are imported to virtual active classes, so all their contents is available, but semantic checker prints error messages about unsupported virtual types.

Guidelines:

Manually change the structure of imported model so that virtuality on types is not used. Individual solution for each model should be applied.

**Example 299: Virtual process type**

SDL

```
block type BType1;
  virtual process type pt1;
  endprocess type pt1;
  process p1(1,1) : pt1;
endblock type BType1;
block type BType2 inherits BType1;
  redefined process type pt1;
  endprocess type pt1;
endblock type BType2;
```

UML

```
active public class BType1 {
  active virtual public class pt1 {
  }
  part pt1 [0 .. 1] p1 / 1;
}
active public class BType2 : BType1 {
  active redefined public class pt1 {
  }
}
```

Messages

```
Class pt1: Error: TSC2023: The use of virtual, redefined
or finalized types is not supported.
```

### Service, service type, service type instance

Services are not imported. A possible workaround is to replace any services with other SDL constructs that can be imported, for example processes, and then manually change the structure of the imported model. One option can be to combine all services (imported, for example, as processes) into one UML state machine, although an individual solution for each model must be considered.

**Example 300: Service**

SDL:

```
PROCESS P1(1,1); SIGNALSET Sig1R, Sig2R, SigInternal;
  SERVICE S1; SIGNALSET Sig1R;
  ENDSERVICE;
  SERVICE S2; SIGNALSET Sig2R, SigInternal;
  ENDSERVICE;
ENDPROCESS;
```

Import messages:

```
Warning    : TSI0224: cdtserv01.sdl(25,11): Services are
not imported to UML. Service 'S1' has been ignored.
Warning    : TSI0224: cdtserv01.sdl(35,11): Services are
not imported to UML. Service 'S2' has been ignored.
```

SDL changed (services are replaced by processes):

```
/* PROCESS P1(1,1); SIGNALSET Sig1R, Sig2R, SigInternal;
*/
PROCESS S1; SIGNALSET Sig1R;
ENDPROCESS;
PROCESS S2; SIGNALSET Sig2R, SigInternal;
ENDPROCESS;
/* ENDPROCESS; */
```

UML (to be changed manually):

```
part active class S1_T {
  statemachine S1_T {
    …
  }
  port io_port in with Sig1R out with SigInternal, Sig1;
} S1 / 1;
part active class S2_T {
  statemachine S2_T {
    …
  }
  port io_port in with Sig2R, SigInternal out with Sig2,
ok;
```

```
} S2 / 1;
```

### Create this

The construction “create this” is not supported.

### Signal refinement

Signal refinement is not imported.

### Channel substructure

Channel substructure is not imported. All substructure elements are imported to the same scope as the channel is defined in.

### Delayed connectors

The “delay” property for connectors is not supported because in UML connectors do not delay signals. In UML the delaying property for connectors can be represented by a user-defined stereotype. The corresponding stereotype can be manually applied to imported channels, but it will not be processed by any of the existing code generators.

### Macros

Macros are not imported to UML.

### Select

The SDL construct “Select if (<Boolean expression>)” is the analogue to a Conditional Compilation. The Selected body appears in an SDL model only if the `boolean` expression is evaluated to `true`. This kind of preprocessing is done before any SDL Import, and any selected entities are imported. The Select construct itself is not imported to UML.

Example:

SDL:

```
system S;
select if ( false );
  signal ss1;
endselect;
```

```
select if ( true );
  signal ss2;
endselect;
  signal ok;
  …
endsystem;
```

UML:

```
package S {
  active class S {
    …
  }
  signal ss2;
  signal ok;
}
```

In UML, the select-if construct can for example be redesigned by using the Conditional Compilation feature.

### Name clashes

Some SDL constructs with the same name can result in name clashes in the generated UML. One example of such clashes are literals and other entities, for example:

**Example 301: Name clashes**

**SDL**

```
signal DR;
newtype IPDUType
  literals CR, CC, DR, DT, AK;
endnewtype IPDUType;
```

**UML**

```
signal DR;
enum IPDUType {
  CR, CC, DR, DT, AK
}
```

In SDL names of literals and other definitions can be correctly resolved, but in UML there will be errors because of name clashes.

Similar situations can appear when there are two entities with the same name in nested scopes.

### Transition option

SDL transition option can be imported to UML decision statement, but a transition option is not supported by the SDL Suite Analyzer.

### Include expression

SDL include expression is not supported.

### Axioms

Axioms cannot be imported to UML. It is not possible to represent SDL axiomatic information in UML.

### RPC transition

"`Input procedure my_x;`" – This code means that transition is made only when remote procedure is called. This is not supported in UML.

### Inline initialization of arrays

Inline initialization of array elements is not allowed in UML, but it is allowed in SDL. Initialization will be imported but the model will not be correct.

**Example 302: Inline initialization of arrays**

**SDL**

```
newtype St1 struct
  aSt1 Integer;
  bSt1 Charstring;
endnewtype;

newtype A Array (Integer, St1)
endnewtype;

block B1;

process P1 (1, 1);
signalset;

dcl var A := (. (. 10, 'hello' .) .);
start;
```

**UML**

```
class St1 {
  public Integer aSt1;
```

```
  public Charstring bSt1;
}

syntype A = Array<Integer, St1>;


part active class B1_T {

   part active class P1_T {


private A a = A (. St1 (.10, "hello".) .);
   statemachine P1_T {
      start {
...
```

The generated UML model will give the following error messages:

```
InstanceExpr 'St1 (.10, "hello".)': Error: TNR0047:
Failed to find definition of St1 (while looking for
InstanceOf).
InstanceExpr 'St1 (.10, "hello".)': Error: TNR0051:
Context resolution failed for InstanceOf in
'InstanceExpr <unnamed>'.
InstanceExpr 'St1 (.10, "hello".)': Error: TNR0034:
Failed to find InstanceOf of InstanceExpr (by ref:St1).
```

In UML an array can not be initialized in the same place where it is declared, so a workaround is to manually add initialization of array variables to the appropriate place. This could for example be done at the beginning of the state machine:

**Example 303: Initialization of array variables**

```
statemachine P1_T {
   start {
     a[0] = St1 (.10, "hello".);
```

## Procedure as qualifier

Procedures can not be used as qualifiers in UML. Such qualifiers are not imported to UML and a warning message is reported.

**Example 304: Procedure qualifiers**

**SDL**

```
process HelloWorld;
```

```
  dcl i Integer := 5;

  procedure op; fpar in a Integer; returns Integer;
    newtype XXX
      literals lit_a, lit_b, lit_c;
    endnewtype;
    dcl i Integer;
    dcl x XXX;
    start;
      task { procedure op i := a + 1; };
      task process HelloWorld i := a;
      task x := procedure op/type XXX lit_b;
      return i;
  endprocedure op;
  ...
endprocess HelloWorld;
```

**UML**

```
part active class HelloWorld_T {
  Integer i = 5;
  Integer op(in Integer a) statemachine {
    enum XXX {
      lit_a, lit_b, lit_c
    }
    Integer i;
    XXX x;
    start {
      i = a + 1;
      HelloWorld_T::i = a;
      x = XXX::lit_b;
      return i;
    }
  }
  ...
} HelloWorld / 1;
```

Messages

```
Information: TSI0200: Importing SDL started
Warning: TSI0204: Procedure can not be used as
qualifier, qualifier 'op' for identifier 'i' is not
imported
Warning: TSI0204: Procedure can not be used as
qualifier, qualifier 'op' for identifier 'XXX' is not
imported
Information: TSI0202: Importing SDL completed
```

### ERROR expression

The ERROR expression is not supported in UML, so it can not be correctly mapped. In the SDL import an ERROR identifier is generated for the SDL ERROR keyword and a warning message is issued. The imported ERROR identifier should be manually replaced by the appropriate code, because this identifier will not be resolved and will cause semantic errors in the UML model.

**Example 305: ERROR expression warning message**

Messages

```
Information: TSI0200: Importing SDL started
Warning: TSI0205: ERROR term is not supported in UML,
expect semantic errors on imported 'ERROR' ident
Information: TSI0202: Importing SDL completed
```

## Restrictions in import from Telelogic SDL Suite

### Import of datatypes with implementation

The implementation of a datatype are ignored when importing SDL with #ADT directives referencing datatypes in the following SDL-PR files, originating from SDL Suite:

```
access.pr
byte.pr
cm_pidlist.pr
file.pr
idnode.pr
list1.pr
list1_noname.pr
list2.pr
list2_noname.pr
random.pr
unsigned.pr
unsigned_long.pr
```

If datatypes with an #ADT implementation are imported from one of these files, the code resulting from these #ADT directives will not be imported. An information message will appear:

```
Information: TSI0215: D:\SDLImport\file.pr(123,5): #ADT
implementation for 'TextFile' datatype from predefined
'file.pr' has not been imported.
```

### #ADT directive used to implement operators

If the #ADT directive is used to implement operators from a newtype defined with generator transformations, it will not be correctly imported. Operators from this data type will be imported into the same scope as the datatype is defined in. The datatype itself is imported to a UML syntype. The corresponding ADT stereotype will be attached to the wrong node and then disregarded during code generation.

### Illegal re-declaration of connectors

In case several signal routes are connected to the same connector, then they must use the same graphical connection point in the SDL diagrams. Repeated "graphical declarations" of a connector in SDL will lead to semantic errors in UML, referring to illegal re-declarations of connectors.

A possible workaround is to use the same connection point.

### Multiple comment symbols result in syntax errors

Having multiple comment symbols connected to the same SDL symbol will cause SDL Suite to generate CIF files that are syntactically incorrect and that thus cannot be imported.

There are two workarounds.

- Concatenate contents into one comment symbol.
- Upgrade to SDL Suite 4.4.6 or later.

### Include graphical SDT References must not be checked

Turning the option **Include graphical SDT references** on will result in CIF comments not to be properly handled and in turn there will be no graphical elements (diagrams/symbols...) created.

### SDL analyzer is operated in case sensitive mode

The SDL analyzer (that checks the source SDL prior to transformation to UML) requires the source SDL to be case sensitive. Should errors be reported by the SDL analyzer, then the SDL must be modified.

**Hint**

*SDL Suite version 4.4 and later provides tool and script support to convert an SDL system to case sensitive SDL. Refer to the chapter "Migration Guidelines", section "Update to case-sensitive SDL".*

# Restrictions when importing from ObjectGeode

## Explicit use of implicit operators

Explicit use of implicit predefined operators for structured types, like modify, extract, make, is allowed only inside the operator definition body in SDL standard. Use of such operators in other places will not be correctly imported to UML, except when used as in Example 306 on page 702.

**Example 306: Explicit use of modify operator**

When the name on the left-hand side of an assignment is the same as the first argument in the construction`<name>modify();`. Then the `modify` operator will be imported to a field expression assignment on the form:

```
value.name = <expression>;
```

SDL

```
newtype N struct
  a Integer;
  b Boolean;
endnewtype;
dcl m N;
task m := amodify(m,1);
```

UML

```
class N {
  public Integer a;
  public Boolean b;
}
part N m;
m.a = 1;
```

# Example Section

## DemonGame (Imported from SDL Suite)

This example describes SDL-PR import, and diagram import, for the DemonGame system. Diagrams are imported from SDL Suite CIF comments that are inserted inside the textual SDL specification.

**Example 307: SDL-PR for DemonGame example**

```
system DemonGame;

  SIGNAL Newgame, Probe, Result, Endgame, Win, Lose,
Score(Integer), Bump;
  channel C1 from env to GameBlock with Newgame, Probe, Result,
Endgame;
  endchannel C1;
  channel C2 from GameBlock to env with Win, Lose, Score;
  endchannel C2;
  channel C3 from DemonBlock to GameBlock with Bump;
  endchannel C3;

  block GameBlock;
    SIGNAL GameOver;
    signalroute R2 from env to Game with Probe, Result;
    signalroute R1 from env to Main with Newgame, Endgame;
    signalroute R5 from Main to Game with GameOver;
    signalroute R3 from Game to env with Win, Lose, Score;
    signalroute R4 from env to Game with Bump;

    process Main;
      DCL GameP Pid;
      start;
        nextstate Game_Off;
      state Game_Off;
        input Newgame;
        create Game;
        task GameP := offspring;
        nextstate Game_On;
      endstate;
      state Game_On;
        input Endgame;
        output GameOver;
        task GameP := Null;
        nextstate Game_Off;
      endstate;
    endprocess Main;

    process Game;
      DCL Count Integer;
      start ;
        task Count:=0;
        nextstate Losing;
      state Losing;
        input Probe;
        output Lose;
        task Count:= Count-1;
        nextstate -;
        input Bump;
        nextstate Winning;
```

```
      endstate;
      state Winning;
        input Bump;
        nextstate Losing;
        input Probe;
        output Win;
        task Count:=Count+1;
        nextstate -;
      endstate;
      state *;
        input Result;
        output Score(Count);
        nextstate -;
        input GameOver;
        stop ;
      endstate;
    endprocess Game;

    connect C1 and R2, R1;
    connect C2 and R3;
    connect C3 and R4;

  endblock GameBlock;

  block DemonBlock;
    signalroute R1 from Demon to env with Bump;

    process Demon;
      timer T;
      start ;
        set(now+1,T);
        nextstate Generate;
      state Generate;
        input T;
        output Bump;
        set(now+1, T);
        nextstate -;
      endstate;
    endprocess Demon;

    connect C3 and R1;
  endblock DemonBlock;
endsystem DemonGame;
```

**Example 308: Resulting UML model for DemonGame example**

```
package DemonGame {
  active class DemonGame {
  part active class GameBlock_T {
    signal GameOver;

    part active class Main_T {
      Pid GameP;
      port R1 in with Newgame, Endgame;
      port R5 out with GameOver;
      statemachine Main_T {
        start {
          nextstate Game_Off;
        }
        state Game_Off;
        state Game_On;
        for state Game_Off;
          input Newgame() {
```

```
            Game = new Game_T();
            GameP = offspring;
            nextstate Game_On;
          }
        for state Game_On;
          input Endgame() {
            ^ GameOver();
            GameP = NULL;
            nextstate Game_Off;
          }
      }
    } Main / 1;

    part active class Game_T {
      private Integer Count;
      port R2 in with Probe, Result;
      port R5 in with GameOver;
      port R3 out with Win, Lose, Score;
      port R4 in with Bump;
      statemachine Game_T {
        start {
          Count = 0;
          nextstate Losing;
        }
        state Losing;
        state Winning;
        for state Losing;
          input Probe() {
            ^ Lose();
            Count = Count - 1;
            nextstate -;
          }
          input Bump() {
            nextstate Winning;
          }
        for state Winning;
          input Bump() {
            nextstate Losing;
          }
          input Probe() {
            ^ Win();
            Count = Count + 1;
            nextstate -;
          }
        for state *;
          input Result() {
            ^ Score(Count);
            nextstate -;
          }
          input GameOver() {
            stop;
          }
      }
    } Game / 1;

    connector R2 from C1 to Game.R2 with Probe, Result;
    port C1 in with Probe, Result, Newgame, Endgame;
    connector R1 from C1 to Main.R1 with Newgame, Endgame;
    connector R5 from Main.R5 to Game.R5 with GameOver;
    connector R3 from Game.R3 to C2 with Win, Lose, Score;
    port C2 out with Win, Lose, Score;
    connector R4 from C3 to Game.R4 with Bump;
    port C3 in with Bump;
  } GameBlock / 1;

  part active class DemonBlock_T {
```

```
    part active class Demon_T {
      timer T;
      port R1 out with Bump;
      statemachine Demon_T {
        start {
          set T() = now + 1;
          nextstate Generate;
        }
        state Generate;
        for state Generate;
          input T() {
            ^ Bump();
            set T() = now + 1;
            nextstate -;
          }
      }
    } Demon / 1;
    connector R1 from Demon.R1 to C3 with Bump;
    port C3 out with Bump;
  } DemonBlock / 1;
  connector C1 from Env to GameBlock.C1 with Newgame, Probe,
Result, Endgame;
  port Env in with Newgame, Probe, Result, Endgame out with Win,
Lose, Score;
  connector C2 from GameBlock.C2 to Env with Win, Lose, Score;
  connector C3 from DemonBlock.C3 to GameBlock.C3 with Bump;
}
  signal Newgame;
  signal Probe;
  signal Result;
  signal Endgame;
  signal Win;
  signal Lose;
  signal Score(Integer);
  signal Bump;
}
```

# Error Messages

### General

There are several sources for messages during SDL-PR and SDL CIF Import:

- Messages printed by the SDL Analyzer, which checks the correctness of the imported SDL file with respect to SDL syntax semantics. Message codes from SDL Analyzer are prefixed with TIL.
- Messages printed by the CIF Analyzer that checks the structure and contents of the CIF comments in the SDL specification.
- Messages printed during SDL import and CIF import (message codes that are prefixed with “OGC” refer to ObjectGeode import).

All the messages during SDL import are printed to the **Script** tab in the Output window.

### Messages from SDL and CIF import

Messages from SDL and CIF import:

| Code | Text | Comment |
|---|---|---|
| TSI19019 | Unable to get license | For OG and SDL Suite |
| TSI19001 | Invalid option: <string> | For OG and SDL Suite |
| TSI19002 | The option is set twice: <string> | For OG and SDL Suite |
| TSI19003 | Option requires additional argument: <string> | For OG and SDL Suite |
| TSI19011 | There should be at least one input file. | For OG and SDL Suite |
| TSI19012 | The errors occurred during profile(s) loading. | For OG and SDL Suite |
| TSI19013 | Only one of `-check`, `-quickcheck` and `-fullcheck` could be specified | For OG and SDL Suite |

| Code | Text | Comment |
|---|---|---|
| TSI19014 | `-semanPath` could not be used with `-quickcheck` or `-fullcheck` | For OG and SDL Suite |
| TSI19015 | `-semanPath` or `-semanConfig` could be used only together with check or transform options | For OG and SDL Suite |
| TSI19016 | Output file(s) format is invalid or undefined: '<name> | For OG and SDL Suite |
| TSI19017 | Input file(s) format is invalid or undefined: '<name> | For OG and SDL Suite |
| TSI19018 | Failed to load predefined package. | For OG and SDL Suite |
| TSI19004 | Cannot identify input file format: there is not extension. | For OG and SDL Suite |
| TSI19005 | Non-standard extension '<string>' in input file name (use `-inputFormat` to specify file type) | For OG and SDL Suite |
| TSI19061 | Cannot identify output file format: there is not extension. | For OG and SDL Suite |
| TSI19062 | Non-standard extension '<string>' in output file name (use `-outputFormat` to specify file type) | For OG and SDL Suite |
| TSI19007 | Error in Tcl script: <string> | For OG and SDL Suite |
| TSI19009 | Can not open output file. | For OG and SDL Suite |

| Code | Text | Comment |
|---|---|---|
| TSI0204 | Warning - Procedure can not be used as qualifier, qualifier '%s' for identifier '%s' is not imported | This message is printed when you use Procedure as qualifier. |
| TSI0205 | Warning - ERROR term is not supported in UML, expect semantic errors on imported 'ERROR' ident | This message is printed for each usage of the ERROR expression in imported SDL. |
| TSI0206 | Information - Procedure '%s' has been imported under the “exported as” name '%s' | Message is printed when Remote procedure with an “`exported-as`” name is imported. |
| TSI0209 | Information - SDL user-defined type conflicts with UML predefined type '%s' from package '%s'. SDL type is NOT imported. | Message is printed when there are Conflicts with UML Predefined types and SDL user-defined types. |
| TSI0210 | Warning - SDL user-defined type conflicts with UML predefined type '%s' from package '%s'. Consider revising imported type. | Message is printed when there are Conflicts with UML Predefined types and SDL user-defined types after import. |
| TSI0211 | Information - External 'C' or 'C++' `%s '%s'` has been ignored. Use C++ Importer to map C and C++ definitions to UML. | See External definitions. |

| Code | Text | Comment |
|---|---|---|
| OGC0517 | (Information, Warning, Error, `FatalError`) - OG native message: "`%s`", line `%d, %s: %s`. | ObjectGeode native message is printed. The first two parameters are processed file and line number. The last two parameters are native error code and native error message. The severity depends on severity of native error message. See ObjectGeode documentation (Appendix E Geodecheck, E.3.3). |
| OGC0518 | Information - Converting OG to SDT started. | Message is printed when converting from ObjectGeode to SDT is started |
| OGC0519 | Information - Converting OG to SDT completed. | Message is printed when converting from ObjectGeode to SDT is completed |

# 18

# *Rose Import*

This chapter describes how to import models created in Rational Rose into Tau. The purpose of this feature is to migrate Rose models to Tau.

**Note**

*Using the Rose import tool is the preferred way to import Rose models. Using XMI import for Rose models is discouraged since it will produce an inferior result.*

# Overview

The Rose Importer is a feature for migrating Rational Rose models into Tau.

The main features of the Rose Importer are the following:

- Full or partial import of Rose model files
  - Model elements and diagrams including layout information
- The importer works directly on Rose model files and does not rely on having Rational Rose installed
- Fully customizable Rose model to U2 file mapping
- The importer can be executed interactively (see "Getting started" on page 713) or from the command line (see "Command line user interface" on page 722)

# Getting started

To import a Rose model into Tau:

- Start Tau and create a new project, or use an existing project
- Select the `Model` node in the Model View
- Start the **Import Wizard** by selecting **File/Import...**
- Select `Import from Rational Rose` and click **OK**
- Click your way through the different pages of the Rose Import Wizard

To import a model from a command window without starting Tau, use the Command line user interface.

# Rose Import Wizard

This section describes the different steps of the Rose Import Wizard in detail.

## The First Step of Rose Import Wizard

The first step of the import process is to specify which model files to import and how to interpret them.

- Specifying model files
- Model file locale
- Loading of referenced model files
- Default location of created U2 files

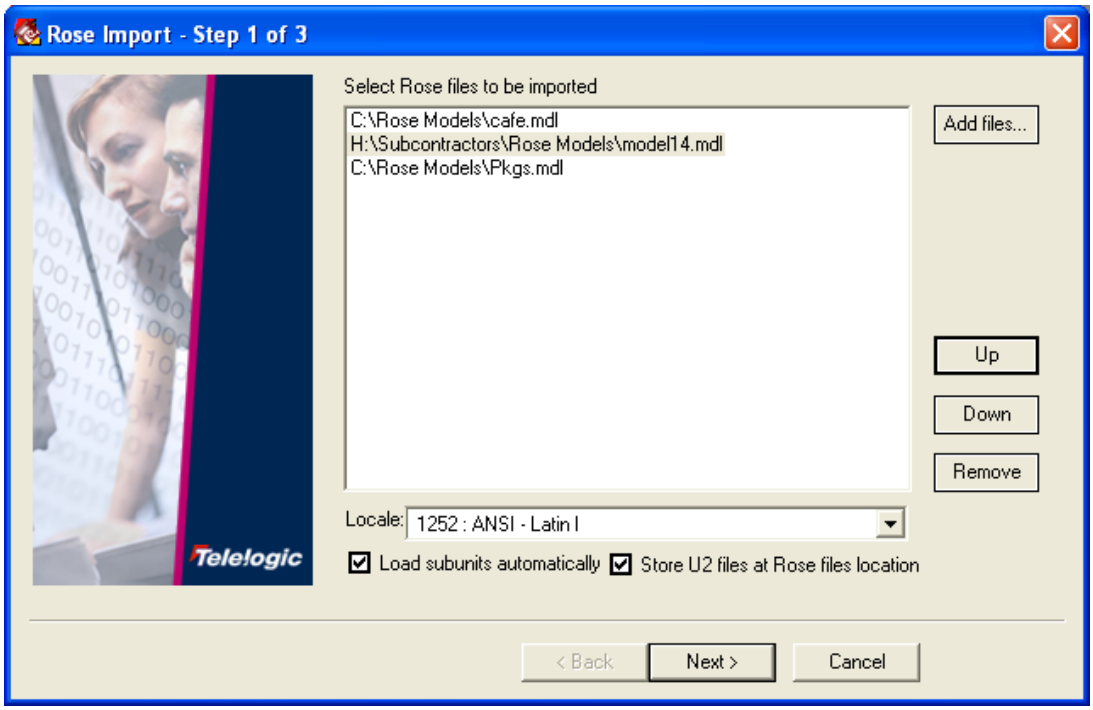



*Figure 172: The first step of Rose Import Wizard*

Specify all model file information and click the **Next** button. The import wizard then parses all the specified files.

**Note**

*Parsing of large model files can take some time. The parsing can be terminated by using the **Cancel** button.*

### Specifying model files

The main purpose of the first step is to determine the files to be imported. The set of files is listed in the centre of the dialog and can be changed by means of buttons **Add Files...**, **Up**, **Down** and **Remove**.

- Pressing **Add Files…** button invokes standard open file dialog with multiple selection allowed. The files selected in the dialog are added to the end of the file list.
- Pressing **Up** and **Down** buttons moves the files currently selected in the list on position up or down respectively.
- Pressing **Remove** button removes the files currently selected from the list.

### Model file locale

The encoding of the model files being imported is set using the **Locale** list box. By default the current system locale is used. When importing a file created using a different locale, the proper locale should be selected in this list box in order to correctly represent localized definitions in Tau.

### Loading of referenced model files

Checkbox **Load subunits automatically** is used to specify parser behavior when it finds a reference to an external file. If the checkbox is checked the parser will try to find the file by the path specified and parse it. Otherwise it will keep unit unloaded. See the next section for details.

### Default location of created U2 files

The default location of the U2 files created during import can be customized using the **U2 files at Rose files location** option.

By default, when the option is enabled, all created U2 files are stored at the same place as the corresponding Rose model files. When this option is disabled, all U2 files are created in the folder of the current Tau project file.

#### Example 309Default file location

Assume you are working with a project called `my_tau_proj` stored in

```
C:\Tau\my_tau_projects\my_tau_proj.ttp
```

and are importing a Rose model consisting of three files each stored in a different folder:

```
C:\my_rose_prj\a\first.mdl
C:\my_rose_prj\b\second.cat
C:\my_rose_prj\c\third.cat.
```

When importing these files with the default file location, three U2 files are created and stored the same directories as corresponding Rose model files:

```
C:\my_rose_prj\a\first.u2
C:\my_rose_prj\b\second.u2
C:\my_rose_prj\c\third.u2
```

If the options is disabled, the following files will be created instead:

```
C:\Tau\my_tau_projects\first.u2
C:\Tau\my_tau_projects\second.u2
C:\Tau\my_tau_projects\third.u2
```

---

**Note**

*This option only controls the default file locations. The file mapping can be fully customized in the next import step, see Specifying U2 file mapping.*

## The Second Step of Rose Import Wizard

The second step of the import process is to specify which parts of the models that should actually be imported and how to store the result.

- Specify which parts of the model to import
- Specifying U2 file mapping
- Locating missing files
- Setting up virtual paths

- Log file

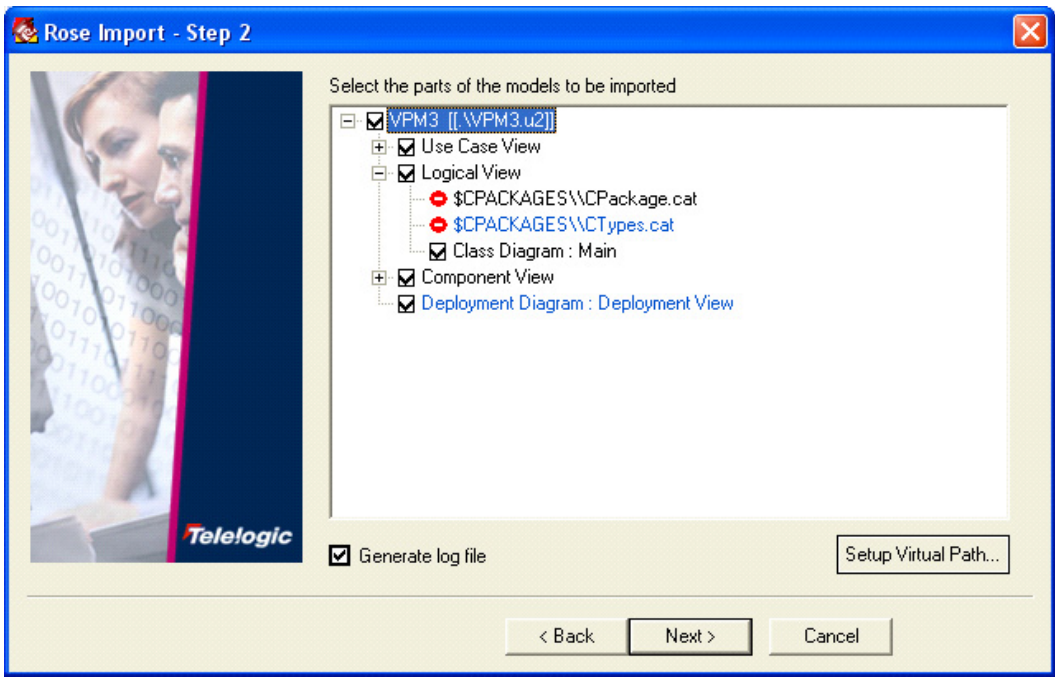



*Figure 173: The second step of Rose Import Wizard*

Specify what to import and the file mapping and then click **Next**. The model is imported and saved into U2 files as specified.

### Specify which parts of the model to import

The parsed models are presented in a tree-view control. The tree-view does not show the full structure of the models, only enough detail to specify a file mapping.

To the left of each tree item there is a checkbox describing its state in the import process.

- Empty checkbox means that neither the entity nor its children will be imported.
- Grey ticked checkbox means that the entity but not all (maybe none) of its children will be imported.
- Black ticked checkbox means that the entity as well as all of its children will be imported.

A red circle indicates an unresolved file reference. See Locating missing files for information on how to resolve any missing file references.

### Locating missing files

If the imported model file(s) contain references to external files which can not be located automatically, a red circle appears in front of the node in the tree-view. There are two ways of resolving missing file references:

- Locate the file manually

  Double click the node in the tree-view and locate the file using the standard open file dialog. Once located, the model file is parsed and the tree-view is updated to show the newly collected data.
- Set up the virtual paths. See Setting up virtual paths

### Specifying U2 file mapping

The tree-view is also used to specify the U2 file mapping for the imported models. Each black node in the tree-view can be associated with a U2 file.

To associate as node with a U2 file:

- Double click the node in the tree-view and browse to the desired location of the file
- Enter a file name and click **Save**

For every node that has a mapping to a U2 file, the file name is appended to the node name using the following format:

```
nodeName [[U2 file name]]
```

The default mapping is one U2 file for each Rose model file, with the same name and stored in the current project directory.

Diagrams cannot be placed in separate u2 files. Their tree nodes are highlighted with blue colour in the tree.

### Setting up virtual paths

Virtual paths are supported by Rational Rose in order to support collaboration by allowing a project to easily be reconfigured for new environment.

If a model containing virtual paths is to be imported the importer tries to resolve the path using Rational Rose registry entries. However if the resolution fails, e.g. if Rational Rose is not installed on the machine where the import is performed, then units with unresolved virtual paths appear as unloaded units in the model-tree view. Check Figure 173 on page 717 for an example.

To specify location for such units there are two ways available. The first is as described above, double-click on the unit tree item and pick the file in the dialog. The second option is to define virtual path variable values. This may be more useful in case when several unloaded units use the same virtual path variable.

To change the virtual path mappings:

- Click the **Setup Virtual Path...** button
- Add and/or edit the virtual path variables as described below
- Click **OK**

The **Virtual Path Setup Dialog** is depicted on Figure 174 on page 719. Initially it contains all the variables which can be extracted from Rational Rose registry data, however the importer stores them internally in the file

```
%APPLICATION DATA%/Telelogic/Shared/tau_virtpath.cfg
```

**Note**

*The changes made to Virtual Paths in the Rose Importer are not propagated to Rose if it is installed.*

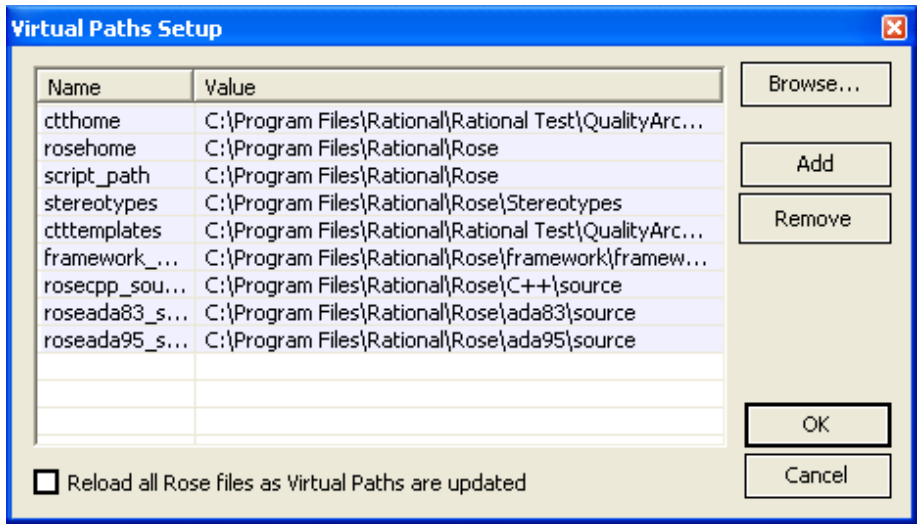



*Figure 174: Virtual Paths Setup dialog*

To add a virtual path mapping:

- Click **Add** and enter the variable name
- Select the newly added variable and click the **Browse...** button to locate the virtual path folder

To change the name of an existing virtual path variable:

- Double click in the name column of the variable and enter a new name.

To change the folder of an existing virtual path variable:

- Select the variable and click the **Browse...** button to locate the new virtual path folder

To remove an existing virtual path variable:

- Select the variable and click the **Remove** button

The option **Reload all Rose models as Virtual Paths are updated** controls whether the parsed model files shall be reparsed or not when applying changes to the virtual paths. If this options is enabled, all model files are reparsed using the new virtual paths.

**Note**

*When reparsing all model files, any customized U2 file mappings, are lost.*

### Log file

If **Generate log file** is checked, all of the error and warning messages will be written to a log file named the same as the first mdl file with the extension `.log` and it is stored in the current Tau project directory

## The Third Step of Rose Import Wizard

When the second step of the import wizard is completed, the import process is started. A progress bar indicates the current progress of the import, and warning and error messages are continously displayed.

**Note**

*Warning and error messages can optionally be written to a Log file.*

The import can be aborted at any time by pressing the **Cancel** button.

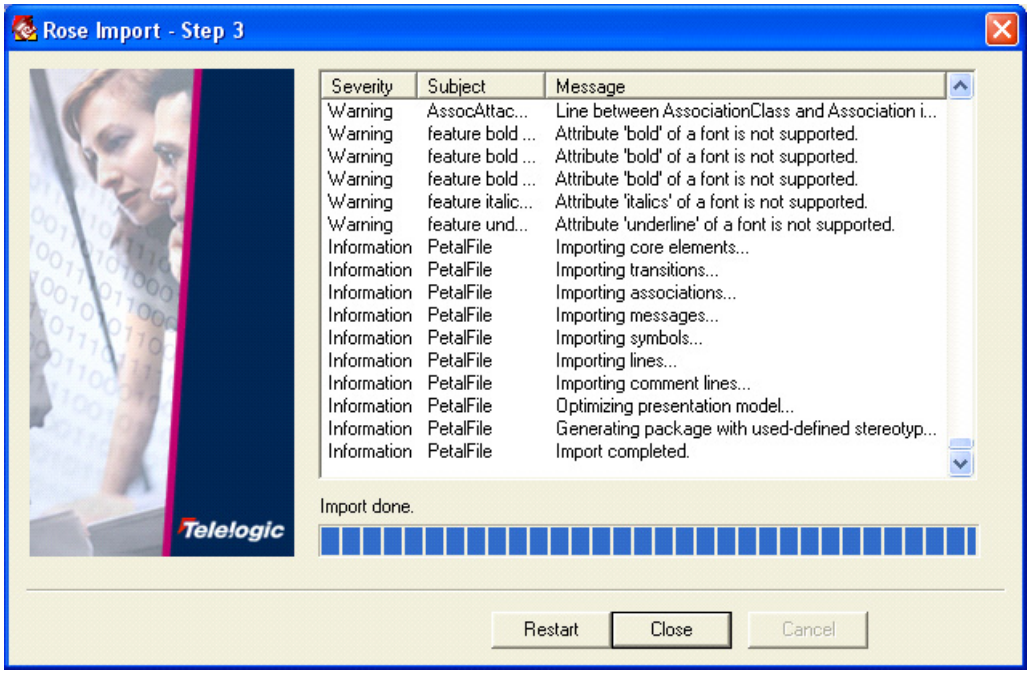



*Figure 175: Rose import is done*

When import is completed the buttons **Restart** and **Close** become active allowing a user to restart the importer or finish the wizard.

After a user has clicked the **Close** button, the files created during the import are added to the current Tau project and loaded in Tau.

# Command line user interface

The Rose Importer can be started from the command line using the following syntax:

```
MDLImpBatch.exe [-p <vpm file>] { <file>.mdl | <file>.ptl }+ <file>.u2
```

`<vpm file>`

A virtual path map file as described in Setting up virtual paths

`{ <file>.mdl | <file>.ptl }+`

A list of model and/or petal files

`<file>.u2`

The resulting top level .u2 file

When using the command line interface the generated file(s) have to be added to a project manually. This can be done by opening the project and using the **Project->Add to project...->Files** command. If the project is Discovery Based, command a refresh of the project by using Manual Rediscovery.

# Transformation rules

In most cases the elements from Rose are imported into Tau as the same kind of UML element, for example a class in Rose is imported as a class in Tau. However for certain constructs the import performs a transformation in order to be able to import as much as possible.

This section describes the transformation rules used during the import.

## Class diagram

An Actor symbol on a Class diagram is transformed to a Class symbol.

A UseCase symbol on a Class diagram is transformed to an Operation symbol.

A Class symbol on a UseCase diagram is transformed to an Actor symbol.

## Collaboration diagram

A Collaboration diagram is transformed to a Sequence diagram. All objects on a Collaboration diagram are transformed to LifeLine symbols.

## State diagram

State diagrams that contain nested states are imported according to the following algorithm: Diagrams are divided into several “levels”. Each level contains its own diagram cloned from the top-level diagram. Each of the nested states from the same level and from the same scope (i.e. substates of the same composition state) are placed on the same diagram. The position and size of state symbols and coordinates of transition lines are not changed. The transition line between states on the same level and scope are imported without transformation. Otherwise the importer creates additional transition lines.

## Activity diagram

Activity diagrams that contains nested activities are processed in the same way as State diagrams with nested states, see State diagram. A State symbol on an Activity diagram is transformed to an ActionActivitySymbol

## Tier diagram

A Tier diagram is transformed to a Class diagram.

## Common rules

A Text label is transformed to a Comment symbol.

# Known limitations

This section describes the known limitations and unsupported constructs.

**Note**

*Many of the limitations and transformations are caused by differences in the UML 1.x and UML 2.1 specifications.*

## Model file format

When importing old Rose models (Petal version less than or equal to 43) some data can be lost.

Rose models stored in a non-ASCII format cannot be fully imported. Elements containing non-ASCII characters cannot be imported.

## All diagrams

Compartments are not supported

### Fonts

The following attributes of class “Font” are not supported: script, bold, italics, underline, strike.

### Symbols

The importer will keep the layout of imported diagrams, including the size of the imported symbols. This may cause parts of the text in the symbols to be cut. To overcome this set the symbol to use autosize, see Autosize symbols.

**Example 310Cut symbol text**

The figure below illustrates symbol text that has been cut. The ellipsis, **...**, in the left diagram indicates that there is some text that is not visible. In the right diagram, autosize has been enabled on the use case symbol, and the entire text is displayed.

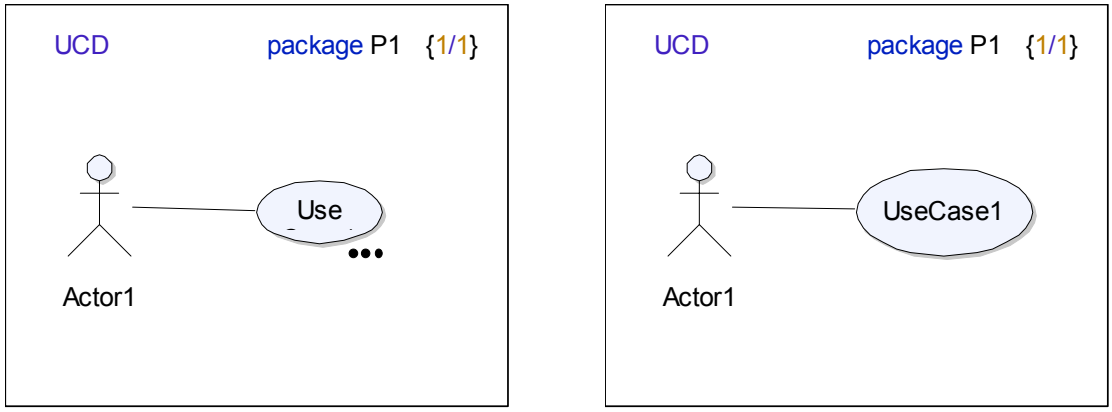



*Figure 176: Cut symbol text*

---

## Activity diagrams

History and HistoryAll are not supported.

## Class diagrams

Line between AssociationClass and Association is not supported.

Constraint line between associations is not supported.

'Name direction' is not supported.

## Sequence diagrams

Attribute “frequency” of a message is not supported.

Documentation on a message is not supported.

## State diagrams

Partition is not supported.

## Tier diagrams

Tiers are not supported

## UseCase diagrams

Name of association between UseCase and Actor is not supported.

Feature 'derived' of association between UseCase and Actor is not supported.

Generalizations between Actors are not supported in use case diagrams.

# 19

# *Together Import*

This chapter describes how to import models created in Borland Together into Tau. The purpose of this feature is to migrate Together models to Tau.

The importer supports *Together® Architect 2006 Service Pack 1 for Eclipse*. Models created using earlier versions have to be migrated to this version before they can be imported into Tau.

**Note**

*Using the Together import tool is the preferred way to import Together models. Using XMI import for Together models is discouraged since it will produce an inferior result.*

# Overview

The Together Importer is a feature for migrating Borland Together models into Tau.

The main features of the Together Importer are the following:

- Full or partial import of Together model files
  - Model elements and diagrams including layout information
- The importer works directly on existing model files and does not rely on having Together installed
- Fully customizable Together model to U2 file mapping
- The importer can be executed interactively (see "Getting started" on page 731) or from the command line (see "Command line user interface" on page 740)

# Getting started

To import a Together model into Tau:

- Start Tau and create a new project, or use an existing project
- Select the `Model` node in the Model View
- Start the **Import Wizard** by selecting **File/Import...**
- Select `Import from Borland Together Architect` and click **OK**
- Click your way through the different pages of the Together Import Wizard

To import a model from a command window without starting Tau, use the Command line user interface.

# Together Import Wizard

This section describes the different steps of the Together Import Wizard in detail.



## Step 1

The first step of the import process is to specify which projects to import and how to interpret them.



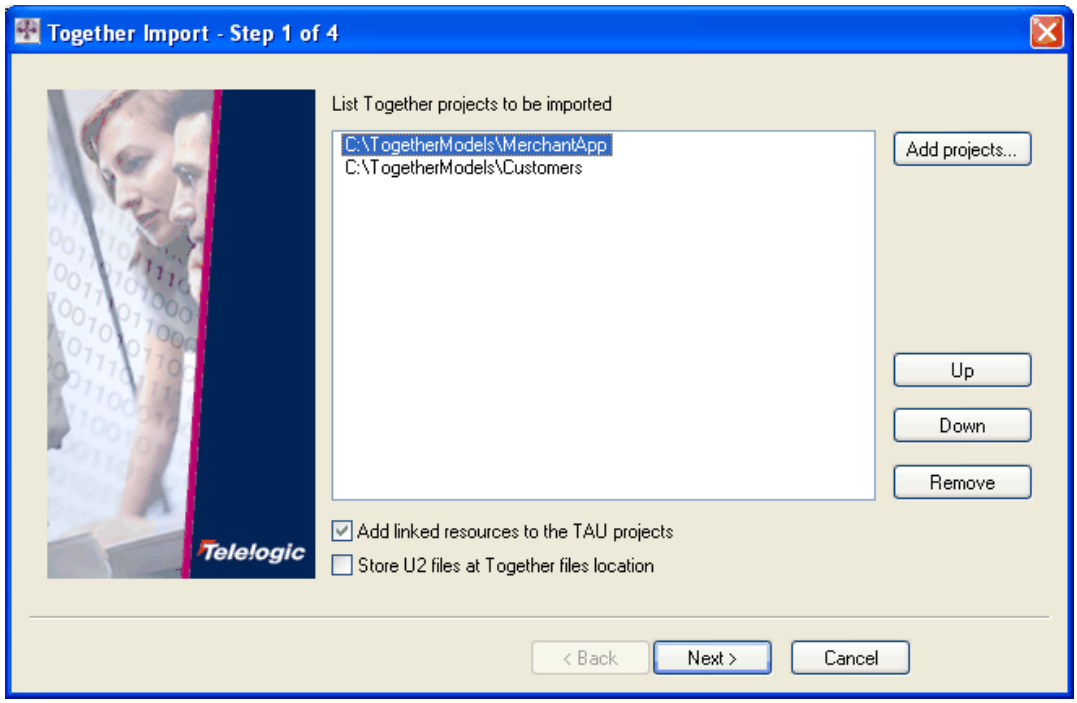



*Figure 177: The first step of the Together Import Wizard*

Specify all model file information and click the **Next** button. The import wizard then parses all the specified files.

**Note**

*Parsing of large project can take some time. The parsing can be terminated by using the **Cancel** button.*

### Specifying projects

To specify which projects to import, press the **Add Projects** button and select a folder. All projects within the folder (including any of its subfolders, recursively) are added to the project list.

This can be repeated any number of times to import many projects at once. A project will only be added to the list once, even if its folder is added more than once.

To edit or rearrange the list of projects, use the **Up**, **Down** or **Remove** buttons.

- Pressing **Up** and **Down** buttons moves the selected project(s) up or down in the list.
- Pressing **Remove** button removes the selected project(s) from the list.

### Linked resources

Linked resources (files and folders) are by default inserted into the Tau project. This can by changed by unchecking the **Add linked resource to the Tau projects** check-box.

### Default location of created U2 files

The default location of the U2 files created during import can be customized using the **Store U2 files at Together files location** option.

By default, when the option is enabled, all created U2 files are stored in the root folder of the corresponding Together project. When this option is disabled, all U2 files are created in the folder of the current Tau project file.

#### Example 311Default file location

Assume you are working with a project called `ImportedProjects` stored in

`C:\Tau\ImportedProjects\ImportedProjects.ttp`

and are importing three Together projects:

```
C:\Project1
```

```
C:\Project2
C:\Project3
```

When importing these files with the default file location, the U2 files are created and stored in the project folders:

```
C:\Together\Project1\Project1.u2
C:\Together\Project2\Project2.u2
C:\Together\Project3\Project3.u2
```

If the options is disabled, the following files will be created instead:

```
C:\Tau\ImportedProjects\Project1.u2
C:\Tau\ImportedProjects\Project2.u2
C:\Tau\ImportedProjects\Project3.u2
```

---

**Note**

*This option only controls the default file locations. The file mapping can be fully customized in the next import step, see Specifying U2 file mapping.*

## Step 2

During the second step the selected project files are parsed and the parse information is displayed.

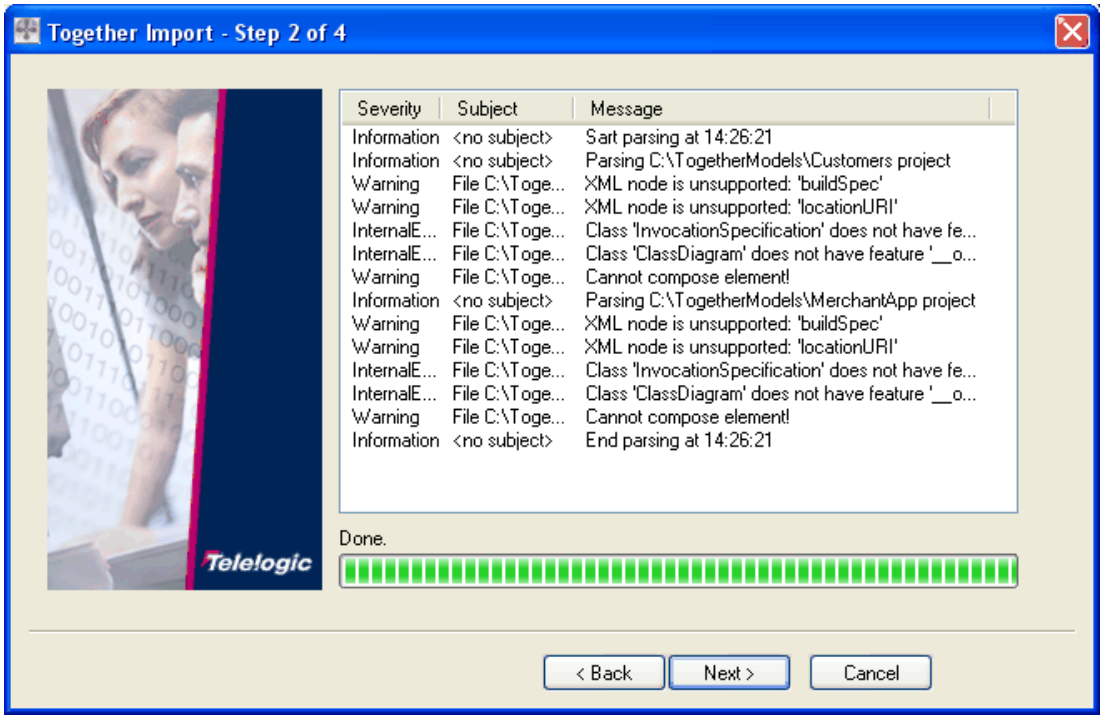



*Figure 178: The second step of the Together Import Wizard*

## Step 3

The second step of the import process is to specify which parts of the models that should actually be imported and how to store the result.

- Specify which parts of the model to import
- Specifying U2 file mapping
- Locating missing files
- Setting up path variables
- Log file

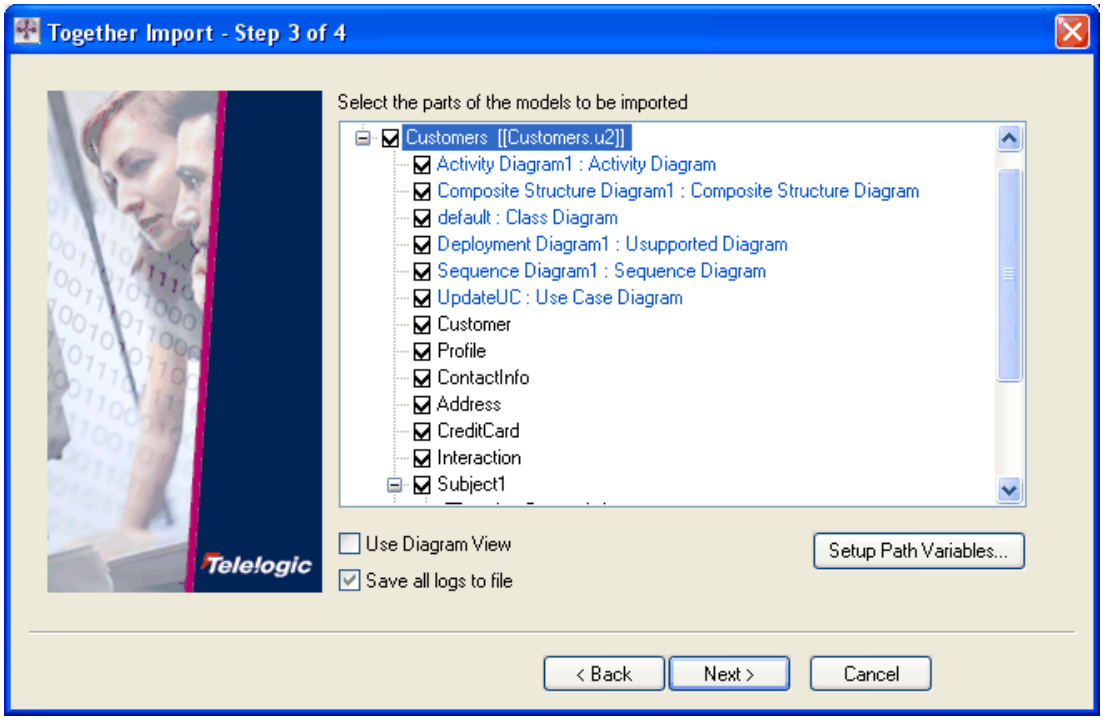



*Figure 179: The third step of the Together Import Wizard*

Specify what to import and the file mapping and then click **Next**. The model is imported and saved into U2 files as specified.

### Specify which parts of the model to import

The parsed models are presented in a tree-view control. The tree-view does not show the full structure of the models, only enough detail to specify a file mapping.

There are two tree-view modes: standard view mode and a diagram view mode. The **Use Diagram View** check box is used to toggle between the modes.

To the left of each tree item there is a checkbox describing its state in the import process.

- Empty checkbox means that neither the entity nor its children will be imported.
- Grey ticked checkbox means that the entity but not all (maybe none) of its children will be imported.
- Black ticked checkbox means that the entity as well as all of its children will be imported.

A red circle indicates an unresolved file reference. See Locating missing files for information on how to resolve any missing file references.

### Locating missing files

If the imported model file(s) contain references to external files which can not be located automatically, a red circle appears in front of the node in the tree-view. There are two ways of resolving missing file references:

- Locate the file manually

  Double click the node in the tree-view and locate the file using the standard open file dialog. Once located, the model file is parsed and the tree-view is updated to show the newly collected data.
- Set up the path variables. See Setting up path variables

### Specifying U2 file mapping

The tree-view is also used to specify the U2 file mapping for the imported models. Each black node in the tree-view can be associated with a U2 file.

To associate as node with a U2 file:

- Double click the node in the tree-view and browse to the desired location of the file
- Enter a file name and click **Save**

For every node that has a mapping to a U2 file, the file name is appended to the node name using the following format:

```
nodeName [[U2 file name]]
```

The default mapping is one U2 file for each Together model file, with the same name and stored in the current project directory.

Diagrams cannot be placed in separate u2 files. Their tree nodes are highlighted with blue color in the tree.

### Setting up path variables

Path variables are supported by Together in order to support collaboration by allowing a project to be easily reconfigured for a new environment.

When importing a project containing path variables the importer tries to resolve the path(s) using Together registry entries. If the resolution fails, e.g. if Together is not installed on the machine where the import is performed, then units with unresolved paths appear as unloaded units in the model-tree view. To specify the location for unresolved units there are two ways available. The first is as described above, double-click on the unit tree item and pick the file in the dialog. The second option is to define path variable values. This may be more useful in case when several unloaded units share the same path.

To change the path variables:

- Click the **Setup Path Variables...** button
- Add and/or edit the path variables as described below
- Click **OK**

The **Path Variables Setup Dialog** is depicted on Figure 180 on page 738. Initially it contains all the variables which can be extracted from Together registry data, however the importer stores them internally in the file

```
%APPLICATION DATA%/Telelogic/Shared/tau_virtpath.cfg
```

**Note**

*The changes made to path variables in the Together Importer are not propagated to Together if it is installed.*

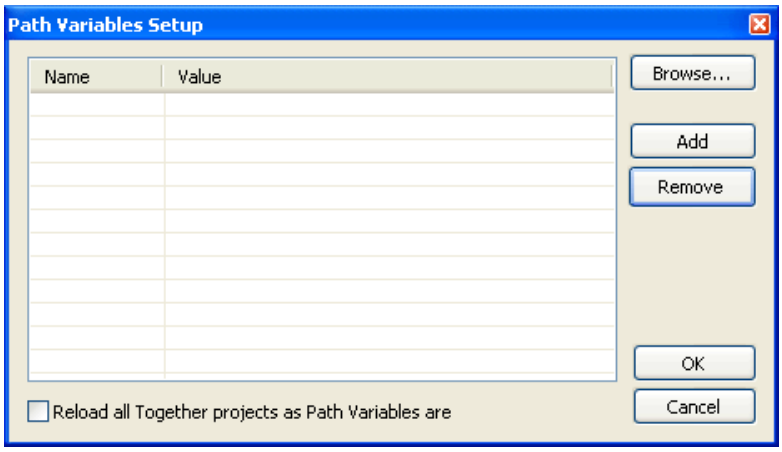



*Figure 180: Path Variables Setup dialog*

To add a path variable:

- Click **Add** and enter the variable name
- Select the newly added variable and click the **Browse...** button to locate the variable folder

To change the name of an existing path variable:

- Double click in the name column of the variable and enter a new name.

To change the folder of an existing path variable:

- Select the variable and click the **Browse...** button to locate the new variable folder

To remove an existing path variable:

- Select the variable and click the **Remove** button

The option **Reload all Together project as Path Variables are updated** controls whether the parsed model files shall be reparsed or not when applying changes to the path variables. If this options is enabled, all model files are reparsed using the new variables, if disabled only the ones resolved due to added variables are parsed.

**Note**

*When reparsing all model files, any customized U2 file mappings, are lost.*

### Log file

If **Save all logs to file** is checked, all of the error and warning messages will be written to a log file named the same as the first project with the extension `.log` and it is stored in the current Tau project directory

## Step 4

When the third step of the import wizard is completed, the import process is started. A progress bar indicates the current progress of the import, and warning and error messages are continuously displayed.

**Note**

*Warning and error messages can optionally be written to a Log file.*

The import can be aborted at any time by pressing the **Cancel** button.

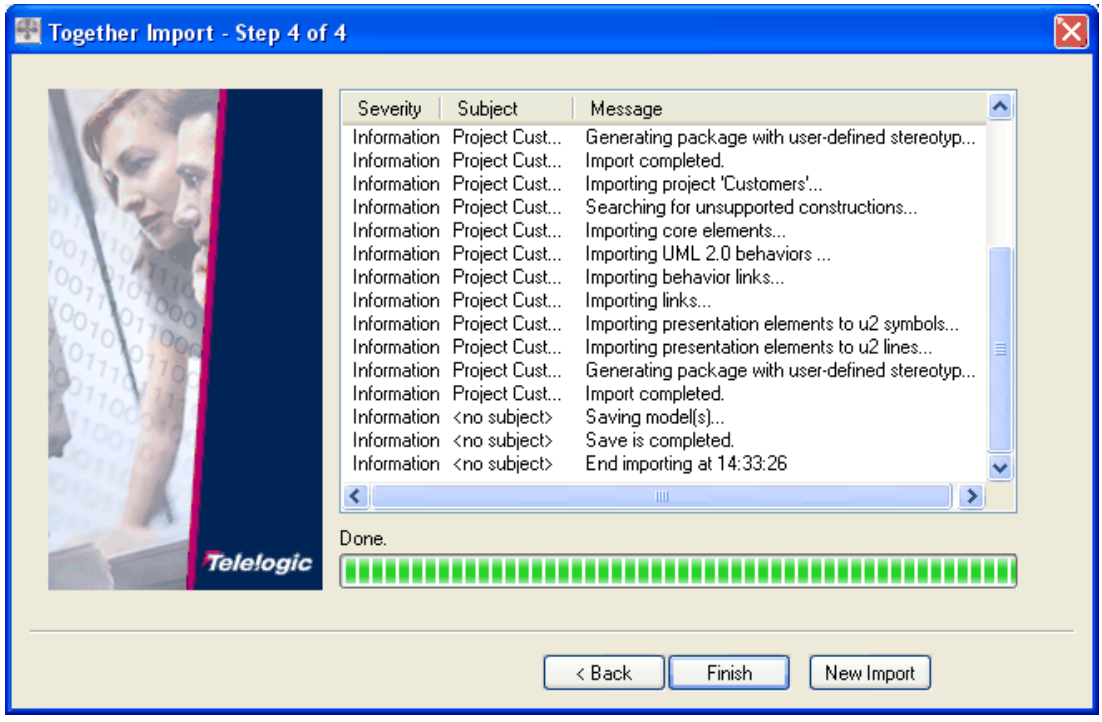



*Figure 181: The fourth step of the Together import wizard*

When the import is completed the **Finish** and **New Import** buttons are enabled to finish the wizard or to import more projects.

When the **Finish** button is clicked, the files created during import are added to the current Tau project and loaded in Tau.

# Command line user interface

The Together Importer can be started from the command line using the following syntax:

```
BTXImpBatch.exe [-noGui <inputDir>+ <outputFile>.u2]
```

If the importer is started without any parameters it is started with the graphical user interface.

`-noGui`

A flag that tells the importer to run without GUI

`<inputDir>+`

A list of folders containing Together models to import

`<outputFile>.u2`

The resulting top level .u2 file

When using the command line interface the generated file(s) have to be added to a project manually. This can be done by opening the project and using the **Project->Add to project...->Files** command. If the project is Discovery Based, command a refresh of the project by using Manual Rediscovery.

# Transformation rules

In most cases the elements from Together are imported into Tau as the same kind of UML element, for example a class in Together is imported as a class in Tau. However for certain constructs the import performs a transformation in order to be able to import as much as possible.

This section describes the transformation rules used during the import.

## Class diagram

### Actor symbol

An Actor symbol on a Class diagram is transformed to a Class symbol.

### UseCase symbol

A UseCase symbol on a Class diagram is transformed to an Operation symbol.

## UseCase diagram

### Class symbol

A Class symbol on a UseCase diagram is transformed to an Actor symbol.

## Communication diagram

A Communication diagram is transformed to a Sequence diagram. All objects on a Communication diagram are transformed to LifeLine symbols.

## State diagram

A State diagram that contains nested states is imported according to the following algorithm:

The diagram is divided into several “levels”. Each level contains its own diagram cloned from the top-level diagram. Each of the nested states from the same level and from the same scope (i.e. substates of the same composition state) are placed to the same diagram. The position and size of the state sym-

bols and the coordinates of transition lines are not changed. Transition lines between states on the same level and scope are imported without transformation. Otherwise additional transition lines are created.

## Activity diagram

The Activity diagram that contains nested activities are processed in the same way as a State diagram with the nested states.

### State symbol

A State symbol on an Activity diagram is transformed to an ActionActivitySymbol.

## Common rules

A Text label is transformed to a Comment symbol.

# 20

# *UML 1.x Import*

This chapter describes the import of UML 1.x models and diagrams created by other UML tools than Telelogic Tau.

# Operation Principles

### XMI

XMI - XML Metadata Interchange - is a UML metadata representation standard based on XML that allows to interchange UML models between different (separate) tools. XMI DTDs (XML Document Type Definitions) serve as syntax specifications for XMI documents, and allow generic XML tools to be used to compose and validate XMI documents.

A UML meta model class is represented in the XMI DTD by an XML element whose name is the class name. The element definition describes the attributes of the class; references to association ends relating to the class; and nested classes, either explicitly or through composition associations.

An attribute of a Metamodel class is represented in the DTD by an XML element whose name is the attribute name.

An association (both with and without containment) between metamodel classes is represented by two XML elements that represent the roles of the association ends.

## XMI import

During UML import a file that complies to the XMI standard is read, and after interpreting the contents of the XMI file a UML model is created. After the import has been done, presentation elements (diagrams and symbols) are created in order to visualize the imported contents. Furthermore, if the imported XMI file contains diagram and symbol information that, such information will be use to preserve the appearance of the resulting UML model.

XMI files without any diagram information will be imported, but only UML model elements will be created.

### XMI import add-in

The XMI import is provided among the Add-Ins and named **XMIImport**.

### XMI import architecture

The architecture of this feature is outlined in Figure 182 on page 745. The XMI Reader module reads a file with XMI specification. XMI Reader transforms information from each tag and passes it to the UML API.

All elements of the UML model are created in the UML API. The core of UML API is a set of C++ classes with the same class hierarchy as in the UML meta model. The UML API is the module builder, which (together with XMI Reader) creates a skeleton of UML model on the fly (tag by tag).

Some kinds of information can not be added to UML model in this phase. This is collected and passed to UML Resolver module.

The U2 Resolver performs a set of transformations to the skeleton of UML model.

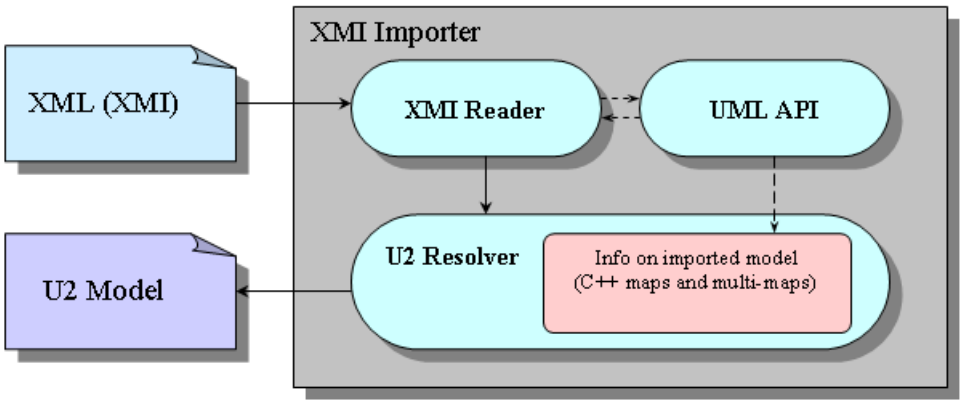



*Figure 182: XMI import architecture.*

**Example 312: UML resolver**

An example of information passed to the U2 resolver is import of an “enumeration” data type. For example Rational Rose will export “enumeration” as a class stereotyped by «enumeration», however in Tau “enumeration” is a `DataType`. Information about applied stereotypes is not available during Class import, thus this Class must be transformed later. Information about required transformation is passed to the U2 Resolver during stereotype import.

## Import an XMI file

The XMI import is called from Tau graphical user interface. In order to activate the XMI import, a workspace with a project must be open.

- Select a **Package** in the Model View.
- Open the Import Wizard (**File** menu, **Import...** command).
- Select **Import XMI** in the dialog window and press **OK**.
- Specify the XMI file to import in the dialog window that appears.

The following should be the result when the second dialog closes:

- A package **ImportedXMIDefinitions** is created in the model
- A stereotype **xmiImportSpecification** is applied to the package.
- The XMI file to import is stored as a value in the stereotype instance for the package.
- The import operation is performed, and the result is added to the created package.

### Importing XMI specification with the same settings once again

In the Model View select a package with the **xmiImportSpecification** stereotype applied.

Right-click on the package and in the pop up menu, select **Import XMI**.

The import operation is performed, and the result is placed into the package.

To change settings (select file to import), the properties in the stereotype instance for the package can be edited, before doing an **Import XMI** command.

**Note**

*When the Import wizard dialog is used, a new package is created. When Import XMI from the pop up menu is reused, the existing package is reused.*

# Supported XMI and UML

## Language and version support

The following languages and versions are supported by the XMI import:

- XMI 1.0/1.1
- UML 1.4

Listed below are the UML 1.4 entities that are supported by the XMI import. Relations and attributes to entities are also supported unless specified otherwise.

### Foundation / core

```
Association
AssociationEnd
Attribute
Class
Comment
Component
Constraint
DataType
Dependency
ElementResidence
Enumeration
EnumerationLiteral
Generalization
Interface
Method
Operation
Parameter
Permission
StructuralFeature
```

### Foundation / extension mechanisms

```
Stereotype
TaggedValue
TagDefinition
```

### Foundation / data types

```
Boolean
BooleanExpression
Expression
Integer
Multiplicity
MultiplicityRange
Name
ProcedureExpression
String
Uninterpreted
```

### Model management

```
Model
Package
```

```
Subsystem
```

### Behavioral elements / common behavior

```
ActionSequence
Argument
CallAction
CreateAction
DestroyAction
Exception
ReturnAction
SendAction
Signal
TerminateAction
UninterpretedAction
```

### Behavioral elements / collaborations

```
ClassifierRole
Collaboration
Interaction
Message
```

### Behavioral elements / use cases

```
Actor
Extend
Include
UseCase
```

### Behavioral elements / state machines

```
CompositeState
CallEvent
FinalState
Guard
Pseudostate
 Initial
 Choice
 Junction
 DeepHistory
 ShallowHistory
SignalEvent
State
SimpleState
StateMachine
```

## Supported diagram types

Provided that the XMI file contains the required diagram information, the XMI import supports the following UML diagram types:

- Class diagram

- Component diagram
- Deployment diagram
- Package diagram
- Activity diagram
- Sequence diagram
- Use case diagram
- State machine diagram

### Importing with preserved layout

Diagrams that belong to this category are diagrams in which the graphical layout is present in the XMI file.

- Class diagram
- Component diagram
- Deployment diagram
- Package diagram
- Activity diagram
- Use case diagram
- Sequence diagram
- State machine diagram

### Import of nested states

Although the layout is preserved some special considerations apply for nested states.

- For each state with nested states a set of diagrams will be created (one for each nested level).
- The positions of states on these diagrams will as far as possible be the same as on original.
- Start and Return symbols will be created on each new diagram when necessary. The positions of these symbols will as far as possible be the same as the position of corresponding symbols on the higher nested level.
- New entry and exit connection points will be created when necessary.
- Transition events and actions containing large amounts of text may overlap.

# Import from UML 1.x tools

In general terms, the XMI import tool supports XMI files from the following UML 1.x tools that comply to the supported XMI version(s).

- Rational Rose/Unisys (JCR.2 v.1.3.x)
- Tau UML Suite
- Borland Together
- IBM XMI Toolkit.

### Rhapsody

Rhapsody exports XMI, but without any diagram information The information in the XMI files originating from Rhapsody is used to create model elements. This will result in a UML structure in the workspace window (but no diagrams).

### Rational Rose

- Rational Rose with Unisys extensions exports XMI with diagram information. This information is used during the XMI import when creating the diagrams (provided that the diagrams are among the Supported diagram types).
- Diagram layouts are preserved for class diagrams, use case diagrams and sequence diagrams.
- Rational Rose names are supported.

**Note**

*It is strongly recommended to use the Rose Import feature to import models from Rational Rose instead of the XMI import. The Rose Import features works directly on the model files and is therefore able to preserve significantly more information.*

### Preserve DOORS links

It is possible to preserve DOORS links during import of XMI from Rational Rose.

- Export the UML model. Make sure that the “Generate UUIDs” check button is selected.
- Import the generated XMI into Tau.

- Export the new UML from Tau to DOORS, using the existing DOORS integration commands.
- Open the Tau surrogate module in DOORS, and select the menu choice Import Links from Rational Rose and follow the instructions.

When these actions will be completed, all links to or from the surrogate module in DOORS (created by the DOORS Rose Link integration) will then be copied for the Tau surrogate module.

**Tau UML Suite**

- Tau UML Suite with Unisys extensions exports XMI with diagram information. This information is used during the XMI import when creating the diagrams (provided that the diagrams are among the Supported diagram types).
- Diagram layouts are preserved for class diagrams, use case diagrams and sequence diagrams.

**See also**

"Language and version support" on page 746

# Restrictions

In addition to the level of XMI/UML support stated elsewhere in this chapter, the following sections describe other known restrictions.

## Type and variable definitions

- Local datatype definitions are not visible in state machine diagrams.
- Local variable definitions are not visible in state machine diagrams.

## Incomplete model

XMI specification must be a complete, semantically correct UML model in order to be imported. In general, incomplete, or incorrect, specifications cannot be imported to Tau, however in some cases such specifications can be imported as a complete specification or with losing some model information.

**Example 313: Import of incomplete model**

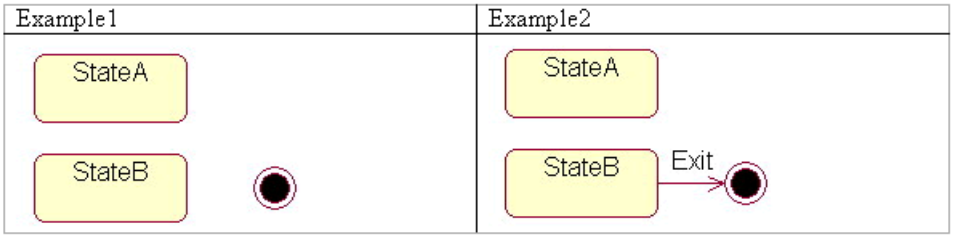



*Figure 183: Incomplete models.*

In Example1 FinalState will not be imported because this state will be transformed to a ReturnAction. This action should be owned by the incoming (to FinalState) transition, and as such a transition does not exist in the example FinalState will not be imported.

In Example2 all diagram elements will be imported, although this diagram is also incomplete (there is no InitialState in this diagram).

## Unsupported classes

Some UML constructs will not be processed during XMI import.

The error message TUI0004 (unsupported classes) will be printed for the following constructs:

**Foundation: Core**

- Artifact
- Association (between Use Cases)
- Binding
- Flow
- Generalization (between Actors)

**Behavioral Elements: Common Behavior**

- AttributeLink
- ComponentInstance
- DataValue

- Instance
- Link
- LinkEnd
- NodeInstance
- Object
- Reception
- Stimulus
- SubsystemInstance

**Behavioral Elements: ActivityGraphs**

- ClassifierInState
- ObjectFlowState
- Pseudostate (Shallow history and Deep history)

**Behavioral Elements: Collaborations**

- AssociationEndRole
- AssociationRole
- CollaborationInstanceSet
- InteractionInstanceSet

**Behavioral Elements: State Machines**

- ChangeEvent
- StubState
- TimeEvent

**Behavioral Elements: Use Cases**

- UseCaseInstance

## Unsupported attributes

An error message for unsupported attributes (TUI0006) will be printed for the following attributes:

**Foundation: Core**

- AssociationEnd
  - Specification
- Attribute
  - AssociationEnd
- BehavioralFeature
  - RaisedSignal
- Component
  - Deployment
- Constraint
  - ConstrainedStereotype
- Feature
  - Owner
- Method
  - Body
  - OwnerScope
- ModelElement
  - Presentation
  - Template
- Operation
  - Concurrency
  - Occurrence
  - Specification

**Foundation: Data Types**

- Expression
  - Language

**Foundation: Extension Mechanisms**

- Stereotype
  - Icon
  - StereotypeConstraint

**Behavioral Elements: Collaborations**

- Collaboration
  - RepresentedClassifier
  - RepresentedOperation
- Interaction
  - Context
- Message
  - Activator

**Behavioral Elements: State Machines**

- CompositeState
  - Concurrent property

**Behavioral Elements: Use Cases**

- Actor
  - Abstract property
- Use Case
  - ExtensionPoint

**Model Management**

- Subsystem
  - Instantiable property

## Unsupported composition

An error message for unsupported composition (TUI0008) will be printed for the following constructions:

**Foundation: Core**

- `AssociationEnd`
  - `Qualifier`
- `Component`
  - `Implementation`

**Behavioral Elements: ActivityGraphs**

- `State`
  - `InternalTransitions` (actions of an activity)
  - `State` (from `StateMachine`)
  - `Pseudostate`: History (Shallow history and Deep History)

**Behavioral Elements: Collaborations**

- `Collaboration`
  - `ConstrainingElement`

**Behavioral Elements: State Machines**

- `StateMachine`
  - `SynchState` (Synchronization bar)
- `State`
  - `InternalTransition` (actions of a `state`)
  - `Pseudostate`: Junction

**Collaboration** diagram is not supported at all.

## Export restrictions

In some cases Rational Rose provides incomplete export. This may result in that some information will be lost after import. The known problems (not exported features) of Rational Rose exporter (Unisys 1.3.6) are listed below.

**Class diagram**

- Class
  - Type (ParameterizedClass, ClassUtility, InstantiatedClass etc.)
  - Multiplicity
  - Space
  - Concurrency
  - Format (show visibility)
- Attribute
  - Containment

- Operation
  - Protocol
  - Qualification
  - Size
  - Time
- Binary Association
  - Constraints
  - Containment
  - Derived
  - Friend
  - LinkElement
  - Name Direction
- Inheritance
  - Documentation
  - Virtual inheritance
  - Friendship Required
- Realization
  - Documentation
- Dependency/Instantiates
  - Multiplicity from
  - Multiplicity to
  - Friendship Required

**State diagram**

- Transition
  - Stereotype
  - Documentation

**Sequence diagram**

- Message
  - Frequency (periodic, aperiodic)
- Destruction Marker

**Use Case diagram**

- Actor
  - Type
  - Multiplicity
- Use Case
  - Stereotype
  - Rank
- Binary Association
  - Derived
  - Link Element
  - Name Direction
  - Constraints
  - Friend
  - Containment
- Dependency

**Package diagram**

- Dependency
  - Documentation

**Component diagram**

- Package
  - Global
- Component
  - Declarations

**Deployment diagram**

- Processor
  - Scheduling
- Process
  - Priority
- Device
  - Stereotype

- Connection

**Activity diagram**

- Swimlane
  - Documentation
- Object
- Object Flow

# Error Messages

### General

Messages during XMI import are printed in the Output window.

### Messages from XMI import

| Code | Text | Comment |
|---|---|---|
| TUI0004 | Attribute '<name>' (<name>) of class '<name>' is unsupported | Error occurs when XMI specification contains attribute that is not specified in XMI standard or it cannot be applied to current class in Tau. For example, attribute 'isAbstract' of class 'Actor' cannot be applied in Tau. |
| TUI0006 | Composition '<name>' from class '<name>' to class '<name>' is unsupported | This error message will be printed in case when composition from one class to other class is unsupported. For example, the 'qualifier' composition is unsupported in Tau. |
| TUI0008 | Class '<name>' is unsupported | Error occurs when imported XMI specification contains unsupported class, for example 'Instance'. |
| TUI0009 | Graphical element '<name>' of class '<name>' was not drawn | This error message will be printed in case when a corresponding ModelElement cannot be found for a PresentationElement. For example, the corresponding ModelElement is unsupported. |
| TUI0010 | Diagram representation of'<name>', with value '<name>', is unsupported | Error occurs when presentation element is not supported by Tau. For example, a PresentationElement for stereotype. |

| Code | Text | Comment |
|---|---|---|
| TUI0016 | Failed to open file '<name>' | File passed to XMI Importer cannot be open |
| TUI0017 | Parse error occurred during parsing of XMI file | This error message will be printed in case when XML parser cannot read information from XMI specification. For example, XML parser cannot find end tag. |
| TUI0022 | Internal error | Internal error occurs during importing. |

# 21

# *UML 1.x Export*

This chapter describes how Tau supports export of model data in XMI format to Tools using UML 1.x.

# XMI Export

## Operation principles

The UML exporter generates a file format that complies with the XMI standard. During export, both model elements and presentation elements are written out to the file. Diagram and symbol layout information is included in order to preserve the appearance of the UML model according to Unisys XML plug-in.

### XMI export add-in

The XMI export is provided among the Add-Ins and is named **XMIExport**.

### Export to an XMI file

The XMI exporter is called from the Tau graphical user interface.

- Initiate the XMI Export (**Tools** menu, **Export Model to XMI...** command).
- Specify the XMI file to export to in the next dialog window that appears.

When more than one project exists in the workspace, the XMI export is done on the selected project. Otherwise, i.e. when zero or more than one projects are selected, the menu choice is dimmed.

## Supported XMI and tool versions

The XMI exporter supports the following:

- XMI 1.1

The XMI exporter is tested for the following target tool environment:

- Rational Rose Enterprise Edition 2003
- Rose XML Tools (UniSys XML plug-in) 1.3.6 for Rational Rose

## Supported UML entities

Following is a list of tables covering Tau UML entities that are supported by the XMI exporter and shows:

- **UML Entity**
  The UML entity - in Tau
- **Export**
  The resulting entity in Rose if exported from Tau and imported into Rose.
- **Roundtrip**
  The resulting entity in Tau if doing an XMI roundtrip.

All other entities not mentioned in this list are not exported.

| **UML Diagram** | **Export** | **Roundtrip** |
|---|---|---|
| Activity diagram | [Same] | [Same] |
| Class diagram | [Same] | [Same] |
| Component diagram | Class diagram | Class diagram |
| Deployment diagram | Class diagram | Class diagram |
| Package diagram | Class diagram | Class diagram |
| Sequence diagram | [Same] | [Same] |
| State machine diagram | [Same] | [Same] |
| Text diagram | Class diagram with a note | Class diagram with a note |
| Use case diagram | Class diagram | Class diagram |

| **General** | **Export** | **Roundtrip** |
|---|---|---|
| Comment symbol | Note | [Same] |
| Annotation line | Anchor | [Same] |
| Text symbol | Note | Comment symbol |
| <Any> | | |
| Comments | Documentation | Nothing |
| Stereotype | [Same] | [Same] |

| General | Export | Roundtrip |
|---|---|---|
| Links | Files | Nothing |
| Color | [Same] | [Same] |
| Font | [Same] | [Same] |

| Activity diagram | Export | Roundtrip |
|---|---|---|
| Activity symbol | [Same] | [Same] |
| • Name | [Same] | [Same] |
| Actions | ‘Entry’ action | Nothing |
| Activity | Class stereotyped as «activity» | Class stereotyped as «activity» |
| Initial Node | [Same] | [Same] |
| Activity Final | End State | Activity Final |
| Flow Final | End State | Activity Final |
| Activity line | Transition | [Same] |
| Text | Transition Label | [Same] |
| Fork/Join | Synchronization | [Same] |
| Decision | [Same] | [Same] |
| • Name | [Same] | [Same] |

| Activity diagram | Export | Roundtrip |
|---|---|---|
| SendSignalSymbol | Unnamed activity with a ‘Do/send’ action | Auto-renamed activity without action |
| AcceptEventSymbol | Unnamed activity with a ‘Do/receive’ action | Auto-renamed activity without action |
| Accept-TimeEventSymbol | Unnamed activity with a ‘Do/receive’ action | Auto-renamed activity without action |

| Class diagram | Export | Roundtrip |
|---|---|---|
| Class | [Same] | [Same] |
| • Name | [Same] | [Same] |
| • Abstract | [Same] | [Same] |
| • Template parameters | Formal arguments | [Same] |
| • Visibility | Export control | [Same] |
| Class Attribute | [Same] | [Same] |
| • Name | [Same] | [Same] |
| • Type | [Same] | [Same] |
| • Visibility | Export control | [Same] |
| • Default value | Initial value | [Same] |
| • Derived | [Same] | [Same] |
| Class Operation | [Same] | [Same] |
| • Name | [Same] | [Same] |
| • Return type | [Same] | [Same] |

| Class diagram | Export | Roundtrip |
|---|---|---|
| • Visibility | Export control | [Same] |
| • Raised exceptions | Exceptions | [Same] |
| Class Operation Parameter | [Same] | [Same] |
| • Name | [Same] | [Same] |
| • Type | [Same] | [Same] |
| • Default value | [Same] | [Same] |
| Required Interface | Interface | Class stereotyped as «interface» |
| Realized Interface | Interface | Class stereotyped as «interface» |
| Interface | [Same] | Class stereotyped as «interface» |
| Timer | Class stereotyped as «timer» | Class stereotyped as «timer» |
| Signal | Class stereotyped as «signal» | [Same] |
| Stereotype | Class stereotyped as «stereotype» | Class stereotyped as «stereotype» |
| Operation | Class stereotyped as «operation» | Class stereotyped as «operation» |
| State machine | Class stereotyped as «`statemachine`» | Class stereotyped as «`statemachine`» |
| Primitive/Enumeration | Class stereotyped as «primitive»/«enumeration» | Class stereotyped as «primitive»/DataType |
| Artifact | Class stereotyped as «artifact» | Class stereotyped as «artifact» |
| Collaboration | Class stereotyped as «collaboration» | Class stereotyped as «collaboration» |

| Class diagram | Export | Roundtrip |
|---|---|---|
| Choice | Class stereotyped as «choice» | Class stereotyped as «choice» |
| Association line | [Same] | [Same] |
| • Name | [Same] | [Same] |
| Association role | [Same] | [Same] |
| • Name | [Same] | [Same] |
| • Visibility | Export control | [Same] |
| Constraints | [Same] | [Same] |
| Multiplicity | [Same] | [Same] |
| Aggregation | Aggregate, Containment | [Same] |
| Owner scope | Static | Nothing |
| Generalization/Realization line | [Same] | [Same] |
| Dependency line | [Same] | [Same] |
| Extension line | Dependency stereotyped as «extend» | Dependency stereotyped as «extend» |

| Component diagram | Export | Roundtrip |
|---|---|---|
| Component symbol | Class stereotyped as «component» | [Same] |

| Deployment diagram | Export | Roundtrip |
|---|---|---|
| DeploymentSpecificationSymbol | Class stereotyped as «deploymentSpecification» | Class stereotyped as «deploymentSpecification» |
| ExecutionEnvironmentSymbol | Class stereotyped as «executionEnvironment» | Class stereotyped as «executionEnvironment» |
| NodeSymbol | Class stereotyped as «node» | Class stereotyped as «node» |

| Package diagram | Export | Roundtrip |
|---|---|---|
| Package | [Same] | [Same] |
| • Name | [Same] | [Same] |
| Dependency line | [Same] | [Same] |

| Sequence diagram | Export | Roundtrip |
|---|---|---|
| Lifeline | [Same] | [Same] |
| • Name | [Same] | [Same] |
| • Type | Class | [Same] |
| Message | Simple message | [Same] |
| • Name | [Same] | [Same] |
| Method call | Procedure Call message | Message |
| • Name | [Same] | [Same] |
| Method reply | Return message | [Same] |
| • Name | [Same] | [Same] |
| Timeout | Timeout message | Message |

| Sequence diagram | Export | Roundtrip |
|---|---|---|
| • Name | [Same] | [Same] |
| Create line | Message where the name is suffixed by ':{Create}' | Message where the name is suffixed by ':{Create}' |
| Interaction | Class stereotyped as «interaction» | Class stereotyped as «interaction» |

| State machine diagram | Export | Roundtrip |
|---|---|---|
| State | [Same] | [Same] |
| • Name | [Same] | [Same] |
| Multi-state (state with a state list or asterisk state) | State with state name set to the original state text | State with state name set to the original state text |
| Transition line | [Same] | [Same] |
| Label | [Same] | [Same] |
| Decision | [Same] | [Same] |
| Decision question | Name | [Same] |
| Decision answer symbol | Transition Guard Condition | Transition Guard Condition |
| Start | [Same] | [Same] |
| Stop | End State | Return |
| Return | End State | [Same] |
| Flow line | Transition | Transition |
| Signal Receipt | Transition Event | Transition Event |

| State machine diagram | Export | Roundtrip |
|---|---|---|
| Guard symbol | Transition Guard Condition | Transition Guard Condition |
| Action symbol | Transition Action | Nothing |
| Signal sending | Transition Send Event | Nothing |

| Use Case diagram | Export | Roundtrip |
|---|---|---|
| Actor | [Same] | [Same] |
| • Name | [Same] | [Same] |
| • Visibility | Export control | [Same] |
| Use Case | [Same] | [Same] |
| • Name | [Same] | [Same] |
| Performance line | Association stereotyped as «performance» | [Same] |
| Dependency line | [Same] | Nothing |
| • Name | [Same] | Nothing |
| Generalization line | [Same] | [Same] |

## Model hierarchy

The containment hierarchy in Rational Rose is structured as shown in Figure 184 on page 773.

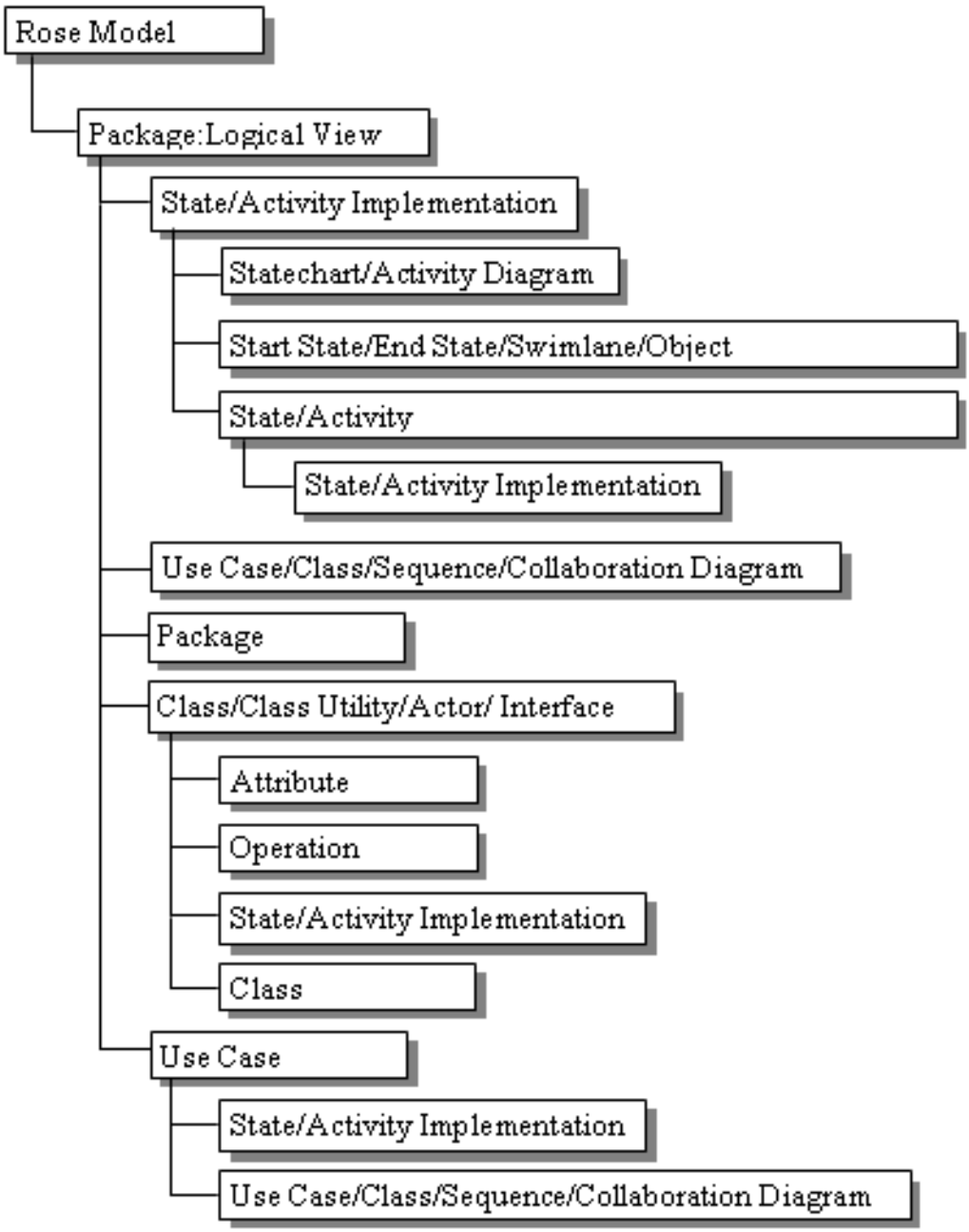



*Figure 184: The containment hierarchy*

Rational Rose views are defined as packages. Logical View is a hard coded predefined package, which is the default place where the Rational Rose XMI module imports model elements and diagrams.

State/Activity Implementations represent state machine specifications and are placed directly beneath the element for which they apply. The hierarchy can be infinitely deep since Packages and Classes (via Classes, Class Utilities, Actors and Interfaces) can be nested. All elements can have Files and URLs beneath them.

As a general rule, containments in an XMI file that are not supported will be lost on XMI import.

### Model transformations

Some transformations take place in order to preserve as much model information as possible.

The table below shows:

- Tau entity.
- A description of the reason to move entities in exported XMI.
- Entities that are moved up in the hierarchy if contained by the Tau entity.

| Tau | Description | Moved Entities |
|---|---|---|
| Internals | This has no counterpart. | `Class diagram, Package diagram, Text diagram, UseCase diagram, Activity, Actor, Artifact, Association, Attribute, Choice, Class, Collaboration, DataType, Interaction, Interface, Operation, Signal, StateMachine, Stereotype, Timer, UseCase` |
| State Machine Implementation | This level is very restricted regarding the types of entities allowed. | `Activity, Actor, Artifact, Association, Attribute, Choice, Class, Collaboration, DataType, Interface, Operation, Signal, Stereotype, Timer, UseCase, Class diagram, Package diagram, Text diagram, Use Case diagram` |
| Activity Implementation | This level is very restricted regarding the types of entities allowed. | `Actor, Artifact, Choice, Class, Collaboration, DataType, Interface, Signal, Stereotype, Timer, Use Case diagram` |
| Interaction Implementation | This has no counterpart. | `Activity, Actor, Artifact, Attribute, Choice, Class, Collaboration, DataType, Interface, Operation, Signal, StateMachine, Stereotype, Timer, UseCase, Sequence diagram, UseCase diagram` |
| Nested classes | State machine diagrams beneath nested classes are not imported. | `State machine diagram` |

| Tau | Description | Moved Entities |
|---|---|---|
| Class | Interface beneath classes are not imported. | `Interface` |
| Attribute | Do not contain anything. | `Artifact, Choice, Class, Collaboration, DataType, Interface, Stereotype` |
| Choice | Transformed into a class. | `UseCase` |

## Restrictions for XMI export to Rational Rose

There are a number of limitations in the Rational Rose XMI Import on Tau exported XMI data. The following is a list of known issues.

| General Features | Description |
|---|---|
| Visibility options | These settings cannot be transferred through XMI and therefore sometimes diagram elements overlap if they do not have the same visibility options set as in Tau. Examples of this are Class attributes, operations and operation signatures. If they are switched off in Tau, but switched on when importing the XMI data, this might cause symbol overlap because of the resulting difference in size of Class symbols. |
| Diagram types | Use Case diagrams are imported as Class diagrams in Logical View. |
| Lines | Lines lose their vertices on import. |
| | Line color is not imported. |
| Elements | Cannot import more than one instance of a symbol (e.g class) in the same diagram. |
| Notes | Size is not imported. |
| | A note gets duplicated once for each of its anchors. |

| Activity diagram | Description |
|---|---|
| Activity | When an activity has both a stereotype and a sub-activity beneath it, it does not import/display properly. |
| | Fill color, Font and Font size are not imported. |
| Decision | Fill color, Font and Font size are not imported. |
| Object | Not imported. |

| Class diagram | Description |
|---|---|
| Class | Multiplicity is not imported. |
| | A Class beneath an Interface is not imported. |
| | Attributes and Operations are not imported for nested classes. |
| Class Attribute | Static is not imported. |
| Interface | Attributes are not imported. |
| Package | Font, Font size and Fill color are not imported. |
| Association | Derived is not imported. |
| | Constraints is not imported. |

| Package diagram | Description |
|---|---|
| Package | Fill color, Font and Font size are not imported. |

| Sequence diagram | Description |
|---|---|
| Lifeline | The horizontal spacing may not show correctly in exported XMI, especially if the associated text is long. |
| | Fill color, Font and Font size are not imported. |

| Sequence diagram | Description |
|---|---|
| Message | Space is added vertically between messages on import. |
| | Messages on the same vertical coordinates get into different levels on the lifeline. |
| | Line color, Font and Font size are not imported. |
| Destruction Marker | Not imported. |
| Note | Does not connect Anchors to Messages on import. |

| State machine diagram | Description |
|---|---|
| State | If a state exists more than once in the same diagram, only one symbol is imported. |
| | Fill color, Font and Font size are not imported. |
| Decision | Fill color, Font and Font size are not imported. |
| Transition line | Line color, Font and Font size are not imported. |

| Use Case diagram | Description |
|---|---|
| Actor | Size is not imported. |
| | Multiplicity is not imported |
| Use Case | Stereotype is not imported. |
| | Size is not imported. |
| Dependency | Not imported if drawn between Use Cases. |

## Error and warning messages

Error and warning messages are given in the Output window in a tab called **XMIExport** and these messages are all navigable.

UML entities that cannot be represented in XMI generates an error message.

There is a warning message given when UML entities are transformed or moved in the containment hierarchy. This is due to incapabilities of Rational Rose to handle such constructs,.

# 22

# *CORBA IDL Exporter*

This document describes how to export CORBA IDL from UML models.

# The CORBA IDL Exporter

The CORBA IDL exporter is one of the delivered Add-Ins, and is comprised of several parts:

- a CORBA profile, which contains a number of stereotypes that are relevant when mapping UML to CORBA IDL; this package also includes stereotypes that are used to control code generation options, such as file options and naming conventions, as well as the agents that perform the actual code generation
- a CCM profile, which contains a number of stereotypes that allows you to take the CORBA Component Model (CCM) and CORBA Implementation Framework (CIF) into account
- a CORBA package, which contains a number of predefined IDL types and type templates to customize the code generation and to allow the use of native IDL types in UML models
- a TCL script that
  - controls the contextual menus that apply to the IDL exporter
  - simplifies the application of stereotypes to model elements through a CORBA specific menu and a CCM specific menu

## Activating the CORBA IDL add-in

The add-in is called `CORBAIDLGenerator`, and is activated by checking the corresponding entry under the add-in tab in the Tools | Customize dialog.

## Creating a CORBA IDL artifact

A CORBA IDL artifact is used to determine what should be exported from a UML model.

The easiest way to create such an artifact is to right-click on an appropriate model element in the model view, and select the “New CORBA IDL Artifact” menu item. The menu item is only applicable when the selected model element is a package, class, or interface. The command is also available from the CORBA menu that becomes available when the CORBA IDL add-in is activated.

An alternative way of creating an IDL artifact is to create a deployment diagram, in which an artifact is manually marked with the stereotype «IDLGenerator». In addition, it is necessary to create a manifest dependency to the model element(s) that are going to be exported into IDL.

**Note**

*The IDL artifact must be saved in a file before any IDL can be generated.*

### The IDLGenerator stereotype

The stereotype «IDLGenerator» extends Artifact. The purpose of the stereotype is to provide a number of options when exporting UML to IDL.

- **targetDir**: the directory in which the exported IDL file(s) should be stored; if a relative path is used, the target directory is relative to the directory in which the model (.u2) file of the artifact is stored.
- **fileName**: the file name of the generated file (if file mapping is ONE_FILE); otherwise, the name of the top-most file
- **fileSuffix**: a suffix that is appended to all generated file names
- **fileMapping**: governs how model elements are mapped into files; the available options are ONE_FILE, MODULE_STRUCTURED, and MODULE_FLAT.
- **isAsyncDefault**: indicates whether operations should be one-way or not by default; default is one-way (true). The default value can be overridden by applying the stereotype «CORBAOperation», and then setting the tagged value overrideIsDefaultAsync.

### File Mappings

When the file mapping is ONE_FILE, all model elements indicated by an artifact are generated into a single IDL file adhering to the file options.

In the case of MODULE_STRUCTURED, each package in the model that is marked as a «CORBAModule» is generated into a file of its own. Each package also corresponds directly to a directory in the file structure. In the MODULE_FLAT case, each package marked as a «CORBAModule» is also generated into a file of its own. In this case, however, all file names are mangled using the scoped name of the package as the file name, and placed in a single directory.

The file name is only applicable to the top-level file if more than one file is generated. In all other cases, the name of the module is used also as the file name.

## Exporting IDL

Once an IDL artifact has been created, it is possible to export IDL. This is done by right-clicking on the IDL artifact, and selecting "`GenerateCORBA IDL`". This command is also available from the CORBA menu.

Exported files are by default put in a directory of its own in the same directory as where the artifact's `.u2` file is stored. The target directory and file name can be modified using the options described above.

## Marking model elements

Currently, only model elements that are marked using a CORBA-specific stereotype (see "The CORBA Profile" on page 789) are considered when mapping IDL.

The easiest way to mark model elements is to select them, either in the model view or in a diagram, and then use the CORBA menu or the CCM menu to select the IDL concept that the model element represents. This will automatically apply the appropriate stereotype from the CORBA or CCM profiles, respectively. It may be necessary to tune the settings using the property editor, such as indicating whether an operation should be one-way or not.

It is also possible to mark a model element by right-clicking on it, and selecting the menu command "Stereotypes...". The resulting dialog box only shows the applicable stereotypes, and you check the appropriate one.

Alternatively, as a third approach, you can select "Stereotypes..." from within the properties editor when the appropriate model element is selected.

## Using CORBA IDL datatypes

Activating the CORBA profile gives access to a package CORBA (under Libraries in the model view) that contains IDL datatypes that can be used when modeling in UML. This also includes a number of template types, such as sequence and array. An exhaustive list of these types can be found in "Predefined IDL types" on page 785.

# Predefined IDL types

The CORBA IDL exporter comes with a number of predefined types that can be used when modeling in UML. These types are then reused when exporting IDL. This further means that they are independent of any target programming language to which the exported IDL may later be mapped.

## Simple types

The predefined types are:

```
short
long
long long
unsigned short
unsigned long
unsigned long long
float
double
long double
boolean
char
octet
wchar
string
wstring
Object
native
any
TypeCode
```

## Template types

The following template types are defined:

```
sequence
array
string
wstring
fixed
```

### sequence

The sequence template comes in two flavors: one that has an upper bound and one that does not. Their signatures are:

```
sequence<type T>
sequence<type T, const Natural index>
```

The following is an example of how these sequence templates might be used in UML:

```
// UML
<<CORBATypedef>> syntype Seq1 = sequence<long>;

<<CORBATypedef>> class Seq2 {
        sequence<long, 5> dummy;
}

<<CORBATypedef>> syntype Seq3 = sequence<Seq1>;
```

### array

The array template is different from how arrays work in IDL, since multidimensional arrays are not directly supported, and also because the square brackets are not used. The signature of the array template is:

```
array<type T, const Natural index>
```

**Note**

*To accomplish the effect of a multidimensional array, you have to use an array of an array, as is the case for* `MultiArray` *below.*

The following is an example of how this array template might be used in UML:

```
// UML
<<CORBATypedef>> syntype MyArr1 = array<4, long>;

<<CORBATypedef>> class MyArr2 {
        array<5, char> dummy;
}

<<CORBATypedef>> syntype MyArr3 = array<6, MyArr2>;

<<CORBATypedef>> syntype MultiArray = array<3, MyArr1>;
```

### string

The string type comes in two flavours, only one of which is a template type. Their signatures are:

```
string
string<const Natural size>
```

The following is an example of how the string and string template might be used in UML:

```
<<CORBATypedef>> syntype MyStr1 = string;
```

```
<<CORBATypedef>> class MyStr2 {
        string<5> dummy;
}

<<CORBAInterface>> interface I3 {
        string a;
        MyStr1 b;
        MyStr2 c;
};
```

### wstring

The wstring type comes in two flavours, only one of which is a template type. Their signatures are:

```
wstring
wstring<const Natural size>
```

The following is an example of how the wstring and wstring template might be used in UML:

```
<<CORBATypedef>> syntype MyStr1 = wstring;

<<CORBATypedef>> class MyStr2 {
        wstring<5> dummy;
}

<<CORBAInterface>> interface I3 {
        wstring a;
        MyStr1 b;
        MyStr2 c;
};
```

### fixed

The fixed template has the following signature:

```
fixed<const Natural digit, const Natural scale>
```

The following is an example of how the fixed template might be used in UML:

```
// UML
<<CORBAInterface>> interface I5 {
        fixed<7, 3> a;
};
```

## UML predefined types

In addition, it is possible to use predefined UML types, such as Integer, Natural, and Real. Their mappings to IDL are described in "Predefined type" on page 809.

# The CORBA Profile

The following stereotypes are used to control the mappings when exporting UML to IDL. The corresponding mappings are outlined in "Mapping rules" on page 798.

The stereotypes defined here comprise the ones that are available in the UML profile for CORBA specification from the OMG. However, they have been updated to take UML 2.1 into account, so there are some differences in which stereotypes are supported and the metaclasses that they extend.

### CORBA

The stereotype «CORBA» is abstract, and is used only as a superclass to other stereotypes.

### CORBAAttribute

The stereotype «CORBAAttribute» extends Attribute. See "Attribute" on page 799 for further information.

The tagged value `isConstant` is used to indicate that the attribute represent a constant value. Note that there is no need to use a utility class to represent module level constants, as it is allowed to put attributes directly in packages.

The fact that an attribute is readonly is represented through the metafeature `isReadOnly`, which is a property of the base metaclass `Attribute`.

The stereotypes on an attribute can be visualized in a diagram by dragging the attribute from the browser view into for example a class diagram.

### CORBABoxedValue

The stereotype «CORBABoxedValue» extends Syntype and Class.

### CORBAEnum

The stereotype «CORBAEnum» extends DataType. See "Enumeration" on page 802 for further information.

### CORBAException

The stereotype «CORBAException» extends Class. See "Exception" on page 803 for further information.

### CORBAInclude

The stereotype «CORBAInclude» extends Dependency. See "Include" on page 804 for further information.

### CORBAInterface

The stereotype «CORBAInterface» extends Interface. See "Interface" on page 804 for further information. For backward compatibility reasons the stereotype also extends Class, but this use is not recommended despite the fact that the CORBA specification uses only this approach. With UML 2.1, the metaclass Interface provides a much more natural mapping to the CORBA interface.

The tagged value `isLocal` is used to indicate whether the interface is local or not. By default, interfaces are not local.

### CORBAModule

The stereotype «CORBAModule» extends Package. See "Package" on page 806 for further information.

### CORBAOperation

The stereotype «CORBAOperation» extends Operation and Signal. See "Operation" on page 805 and "Signal" on page 810 for further information.

It has a tagged value `overrideIsDefaultAsync`, which is used to override the default mapping for operations (which is as one-way operations).

It has another tagged value `context`, which carries strings that are associated with the operation.

The fact that an attribute is readonly is represented through the metafeature `isReadOnly`, which is a property of the base metaclass `Attribute`.

The stereotypes on an operation can be visualized in a diagram by dragging the operation from the browser view into for example a class diagram.

### CORBASequence

The stereotype «CORBASequence» extends Class. See "Sequence" on page 810 for further information.

### CORBAStruct

The stereotype «CORBAStruct» extends Class. See "Struct" on page 810 for further information.

### CORBATruncatable

The stereotype «CORBATruncatable» extends Generalization.

### CORBATypedef

The stereotype «CORBATypedef» extends Syntype, Class, DataType, and Interface. See "Syntype" on page 811 for further information.

### CORBAUnion

The stereotype «CORBAUnion» extends Class. See "Union" on page 812 for further information.

It has a tagged value `isSimple`, which is used to declare simple unions with a discriminator automatically set to the type long, and where the case labels are set to 0, 1, 2, etc. No default case can be defined in this mode.

Related stereotypes are case and discriminator.

### CORBAValue

The stereotype «CORBAValue» extends Interface.

The tagged value `isCustom` is used to indicate whether the value uses custom marshalling or not. See "Value" on page 812 for further information.

### CORBAValueFactory

The stereotype «CORBAValueFactory» extends Operation.

### case

The stereotype «case» extends Attribute. A case can only be used on classes that are marked «CORBAUnion».

The stereotype has a tagged value label that is used to express the condition that is to be used to select ("switch") the case.

It also has a tagged value isDefault that is used to indicate that the case is the default one. If this value is true, the case label is ignored.

A union can have at most one case that have isDefault set to true.

### discriminator

The stereotype «discriminator» extends Attribute. A discriminator can only be used on classes that are marked «CORBAUnion», and each such union can have only one discriminator.

### IDLFile

The stereotype «IDLFile» extends Artifact. See "Include" on page 804 for further information.

It has a tagged value file, which is used to indicate the actual IDL file that is represented by the artifact.

### IDLGenerator

The stereotype «IDLGenerator» extends Artifact. See "The IDLGenerator stereotype" on page 783 for further information.

## Extraneous stereotypes

The following stereotypes are defined in the CORBA profile specification, but not supported by the IDL exporter. Either they are defined as abstract in the CORBA profile specification (in which case they have no practical purpose for the exporter), or they are replaced by other stereotypes or constructs as outlined previously.

- CORBAAnonymousArray (see Array template)
- CORBAArray (see Array template)
- CORBAAnonymousSequence (see Sequence template)

- CORBAConstant (see CORBAAttribute)
- CORBAConstants (see CORBAAttribute)
- CORBAConstructedType (abstract)
- CORBACustomValue (see CORBAValue)
- CORBAUserDefinedType (abstract)
- CORBAIndexedType (abstract)
- CORBAObjectType (abstract)
- CORBAStructType (abstract)
- CORBAStructuredType (abstract)
- CORBAUserDefinedType (abstract)
- CORBAValueSupports (implicit through generalizations between CORBAValues and CORBAInterfaces)
- CORBAWrapper (abstract)
- oneway (see CORBAOperation)
- readonly (see CORBAOperation and CORBAAttribute)
- readonlyEnd (redundant;)
- switch (see discriminator)
- switchEnd (redundant)

# The CCM Profile

The following stereotypes are used to control the mappings when exporting UML to IDL. The corresponding mappings are outlined in "Mapping rules" on page 798.

The stereotypes defined here comprise the ones that are available in the UML profile for CCM specification from the OMG. However, they have been updated to take UML 2.1 into account, so there are some differences in which stereotypes are supported and the metaclasses that they extend.

## Supported stereotypes

### CORBAArtifact

The stereotype `«CORBAArtifact»` extends Class. See "Artifact" on page 798 for further information.

### CORBAComponent

The stereotype «CORBAComponent» extends Class. See "Component" on page 800 for further information.

### CORBAComponentImpl

The stereotype «CORBAComponentImpl» extends Class. See "Component" on page 800 for further information.

The stereotype has a tagged value `category` which is used to indicate what kind of component is represented; the available literals of the enumeration are: `entity`, `process`, `service`, or `session`.

The stereotype has another tagged value `composite`, which is used to indicate the name of the enclosing composition.

### CORBAEvent

The stereotype «CORBAEvent» extends Interface. See "Event" on page 803 for further information.

### CORBAEventSink

The stereotype «CORBAEventSink» extends Port. See "Port" on page 807 for further information.

This stereotype replaces «CORBAConsumes» from the CCM specification, since the port concept provides a more natural mapping.

### CORBAEventSource

The stereotype «CORBAEventSource» extends Port. It has a tagged value `sourceKind`, which is an enumeration with the allowed literals `Emitter` or `Publisher`. See "Port" on page 807 for further information.

This stereotype replaces «CORBAPublishes» and «CORBAEmits» from the CCM specification, since the port concept provides a more natural mapping.

### CORBAFacet

The stereotype «CORBAFacet» extends Port. See "Port" on page 807 for further information.

This stereotype replaces «CORBAProvides» from the CCM specification, since the port concept provides a more natural mapping.

### CORBAFactory

The stereotype «CORBAFactory» extends Operation. See "Home" on page 803 for further information.

### CORBAFinder

The stereotype «CORBAFinder» extends Operation. See "Home" on page 803 for further information.

### CORBAHome

The stereotype «CORBAHome» is a special kind of «CORBAInterface». See "Home" on page 803 for further information.

### CORBAHomeImpl

The stereotype «CORBAHomeImpl» extends Class. See "Home" on page 803 for further information.

### CORBAImplements

The stereotype «CORBAImplements» extends Dependency. See "Implements" on page 804 for further information.

In the CCM specification, this stereotype extends Association, but the metaclass Dependency carries much less overhead.

### CORBAIsProvidedBy

The stereotype «CORBAIsProvidedBy» extends Dependency.

### CORBAManages

The stereotype «CORBAManages» extends Dependency. See "Manages" on page 805 for further information.

In the CCM specification, this stereotype extends Association, but the metaclass Dependency carries much less overhead.

### CORBAPrimaryKey

The stereotype «CORBAPrimaryKey» extends Attribute.

### CORBAReceptacle

The stereotype «CORBAReceptacle» extends Port. See "Port" on page 807 for further information.

The stereotype has a tagged value isMultiple, which is used to indiate whether multiple connections are allowed or not.

This stereotype replaces «CORBAUses» from the CCM specification, since the port concept provides a more natural mapping.

### CORBASegment

The stereotype «CORBASegment» extends Class. See "Segment" on page 809 for further information.

The stereotype has a tagged value features that is used to indicate the features that are supported by the segment.

It also has a tagged value isSerialized to indicate whether the segment is serialized or not.

### CORBASupports

The stereotype «CORBASupports» extends Generalization.

## Extraneous stereotypes

The following stereotypes defined in the CCM profile specification, but are not supported by the IDL exporter. Either they are defined as abstract in the CCM profile specification (in which case they have no practical purpose for the exporter), or they are replaced by other stereotypes as outlined previously.

- CORBAConsumes (see CORBAEventSink)
- CORBAEmits (see CORBAEventSource)
- CORBAEventPort (abstract)
- CORBAProvides (see CORBAFacet)
- CORBAPublishes (see CORBAEventSource

- CORBAUses (see CORBAReceptacle)

# Mapping rules

This section describes how the different UML constructs are mapped when exporting them to IDL.

In most cases, it is required that a UML construct is marked using a relevant CORBA stereotype for any IDL to be generated.

**Note**

*It is not a good idea to apply several CORBA stereotypes to the same model element, as only one of them will be considered (arbitrarily) when generating IDL.*

## Artifact

A class marked «CORBAArtifact» in UML is mapped to an artifact of a segment in IDL. No IDL is generated for classes that are not related to segments through a «CORBAIsProvided» dependency. Also see "Segment" on page 809. An example is shown in "Component" on page 800.

## Association

Associations are not directly mapped. Instead, each navigable association end is mapped as an IDL attribute, case, or field, depending on the owner. If the association end has a multiplicity, this is mapped as described in "Multiplicity" on page 805.

It is in most cases sufficient to mark the owning classifier for the association ends to exported. A notable exception is a union's case and discriminators, as described in "Union" on page 812.

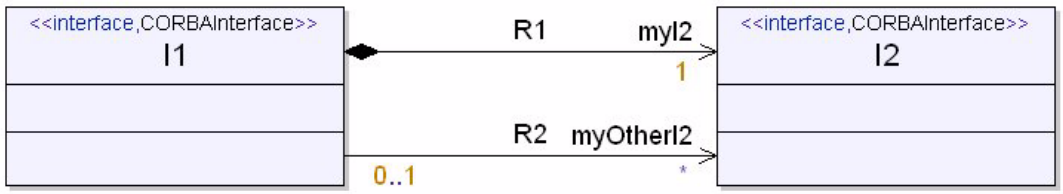



*Figure 185*

The UML model in Figure 185 on page 798 is mapped to the following IDL:

```
interface I2 {
};

interface I1 {
        attribute I2 myI2;
        attribute sequence<I2> myOtherI2;
};
```

## Attribute

An attribute marked «CORBAAttribute» in UML is mapped to an attribute in IDL, unless it is constant (see "Constant" on page 801). It it normally not necessary to use the stereotype, since all attributes of a marked class will be exported.

The following snippet of UML:

```
<<CORBAInterface>> class C1 {
        long a;
        char [0..1] b;
        wchar [*] c;
        boolean d;
}

<<CORBAStruct>> class S1 {
        string e;
        'unsigned long' [8] f;
        C1 [1] g;
}
```

is mapped to the following IDL:

```
interface C1 {
        attribute long a;
        attribute sequence<char, 1> b;
        attribute sequence<wchar> c;
        attribute boolean d;
};

struct S1 {
        string e;
        sequence<unsigned long, 8> f;
        C1 g;
};
```

### Restrictions

Unless they are constant, attributes may only be defined as part of classes or interfaces to be considered for IDL export. Attributes that are defined in packages are thus not mapped to IDL.

## Class

A class marked «CORBAInterface» in UML is mapped to an interface in IDL. An example of this is shown in "Attribute" on page 799.

A class marked «CORBAStruct» in UML is mapped to a struct in IDL. Also see "Struct" on page 810.

A class marked «CORBAUnion» in UML is mapped to a union in IDL. Also see "Union" on page 812.

A class marked «CORBAException» in UML is mapped to an exception in IDL. Also see "Exception" on page 803.

A class marked «CORBATypedef» in UML is mapped to an exception in IDL. Also see "Type definition" on page 811.

### Restrictions

Classes may only be defined as part of packages to be considered for IDL export. Classes that are declared inline are thus not taken into account.

## Comment

Comments that are attached to model elements that are exported are turned into IDL comments (preceded by '//'). Comments can either be written in comment symbols or in the property editor field intended for comments.

## Component

A class marked «CORBAComponent» in UML is mapped to a component in IDL. Similarly, a class marked «CORBAComponentImpl» in UML is mapped to a component composition in IDL, which contains a reference to its managing home implementation. A dependency marked «CORBAComponent» from the component implementation to a component in UML is mapped to a an implements relationship from the composite component to the corresponding component in IDL.

If the component implementation has parts, and those parts are typed by a class that is marked «CORBASegment» in UML, then the composite component will show those segments in IDL (together with their artifacts).

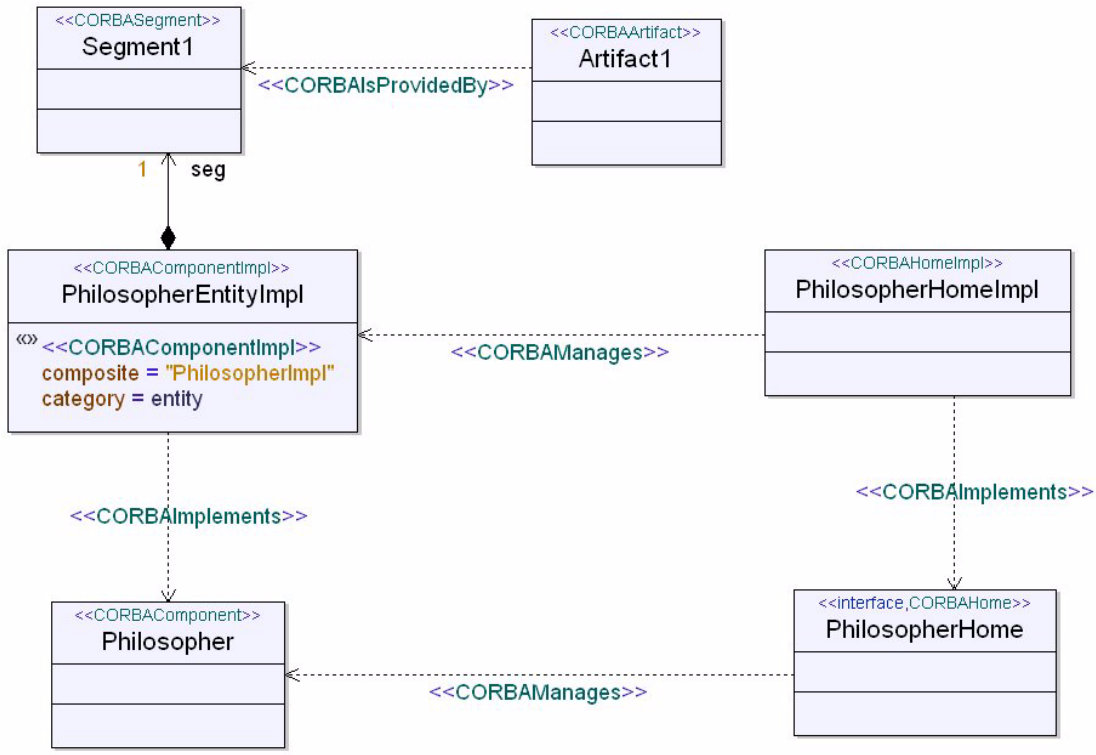



*Figure 186*

The example shown in Figure 186 on page 801 is mapped to the following IDL:

```
composite entity PhilosopherImpl {
        home executor PhilosopherHomeImpl {
                implements PhilosopherHome;
                manages PhilosopherEntityImpl {
                        segment Segment1 {
                                provides (Artifact1);
                        };
                };
        };
};
```

## Constant

An attribute marked «CORBAAttribute» in UML that is also constant is mapped to a constant in IDL.

**Note**

*Constants used as multiplicities are evaluated to their integer values when mapped to IDL.*

The following snippet of UML:

```
<<CORBAModule>> package M5 {
        <<CORBAAttribute(.isConstant=true.)>>
        const Integer nval = 5;

        <<CORBAAttribute(.isConstant=true.)>>
        const Integer mval = nval;

        <<CORBAInterface>> class C1 {
                <<CORBAAttribute(.isConstant = true.)>>
                const Real cval = 22.7;

                long [nval] x;
        }
}
```

is mapped to the following IDL:

```
module M5 {
        const Integer nval = 5;
        const Integer mval = nval;

        interface C1 {
                const double cval = 22.7;
                attribute sequence<long, 5> x;
        };
};
```

**Note**

*Constant expressions are not evaluated, which means that only integer literals and other constants are currently supported.*

# Enumeration

An enumeration marked «CORBAEnum» in UML is mapped to an enumeration in IDL.

The following snippet of UML:

```
<<CORBAEnum>> enum Color {
        red,
        green,
        blue
}
```

is mapped to the following IDL:

```
enum Color {
        red,
        green,
        blue
};
```

## Event

An interface marked as «CORBAEvent» in UML is mapped to an event type in IDL.

## Exception

A class marked «CORBAException» in UML is mapped to an exception in IDL.

The following snippet of UML:

```
<<CORBAException>> class Error {
        string e;
}

<<CORBAInterface>> class C1 {
        <<CORBAOperation(.overrideIsDefaultAsync = true.)>>
        void op() throw Error();
}
```

is mapped to the following IDL:

```
exception Error {
        e string;
};

interface C1 {
        void op() raises (Error);
};
```

## Home

An interface marked as «CORBAHome» in UML is mapped to a home declaration in IDL. The home interface can have a primary key, which in UML is represented through a dependency or association marked «CORBAPrimaryKey»; the supplier of the dependency must be a value type.

An example is shown in "Component" on page 800. An operation marked as «CORBAFactory» in UML is mapped to a factory operation in IDL. Similarly, an operation marked as «CORBAFinder» in UML is mapped to a finder operation in IDL.

The following snippet of UML:

```
<<CORBAHome>> interface Ifc {
        <<CORBAFactory>> void myFactory(in boolean b);
        <<CORBAFinder>> void myFinder(in long a);
}
```

is mapped to the following IDL:

```
home Ifc {
        factory myFactory(in boolean b);
        finder myFinder(in long a);
};
```

## Include

When generating IDL, the include statements that are required when multiple files are generated from an artifact are automatically created. However, sometimes it is necessary to refer to already existing IDL files, and rather than to add those manually in the generated files they can be represented in the model. For this purpose, the stereotype «CORBAInclude» is provided.

Such a dependency from an artifact marked «IDLGenerator» to an artifact marked «IDLFile» in UML is mapped to an include statement of the indicated file.

**Important!**

*The «CORBAInclude» stereotype is taken into account only when it is used between properly marked artifacts. In particular, it is not a replacement for the normal access or import dependencies between packages.*

## Implements

A dependency marked «CORBAImplements» in UML is mapped to an implementats statement in IDL. The dependency always goes from a component implementation to its component, or from a home implementation to its home interface.

An example is shown in "Component" on page 800.

## Interface

An interface marked «CORBAInterface» in UML is mapped to an interface in IDL.

### Restrictions

Interfaces may only be defined as part of packages to be considered for IDL export.

## Manages

A dependency marked «CORBAManages» in UML is mapped to a manages statement in IDL. The dependency always goes from a home implementation to a component implementation, or from a home interface to a component.

An example is shown in "Component" on page 800.

## Multiplicity

Multiplicities may consist of multiple ranges, but must only be made up of integer literals or constants. The multiplicities are evaluated into a single range that includes all legal values. All multiplicities are mapped into sequences, which are either bound or unbound, with the exception of those that have a multiplicity of exactly 1 (one).

In the case of multiple ranges, the upper bound equates to the maximum value present in the ranges.

| UML | IDL |
|---|---|
| `long [1]` | `long` |
| `long [0..1]` | `sequence<long, 1>` |
| `long [0..*]` | `sequence<long>` |
| `long [1..*]` | `sequence<long>` |
| `long [6..*]` | `sequence<long>` |
| `long [3..8]` | `sequence<long, 8>` |
| `long [7..7]` | `sequence<long, 7>` |
| `long [1, 7..9, 3, 5]` | `sequence<long, 9>` |

## Operation

An operation marked «CORBAOperation» in UML is mapped depending on its tagged values. By default, an operation is mapped to a one-way operation in IDL. By setting the stereotype `overrideIsDefaultAsync` to true, the IDL operation will be generated as an ordinary operation in IDL.

The following snippet of UML:

```
<<CORBAInterface>> interface I1 {
```

```
        signal op1;

        <<CORBAOperation(.overrideIsDefaultAsync = true.)>>
        long op2();

        void op3(in long a);

        <<CORBAOperation(.overrideIsDefaultAsync = true.)>>
        string<5> op4(inout char b, in long c);
}
```

is mapped to the following IDL:

```
interface I1 {
        oneway void op1();
        long op2();
        oneway void op3(in long a);
        string<5> op4(inout char b, in long c);
};
```

### Restrictions

If an operation is mapped to a one-way operation only in-parameters are allowed.

Operations may only be defined as part of classes or interfaces to be considered for IDL export.

## Package

A package marked «CORBAModule» in UML is mapped to a module in IDL.

The following snippet of UML:

```
<<CORBAModule>> package P1 {
        <<CORBAInterface>> interface I2 {
                long a;
        }
}
```

is mapped to the following IDL:

```
module P1 {
        interface I2 {
                attribute long a;
        };
};
```

## Parameter

Parameters need not be marked using a CORBA stereotype, because they are always associated with their owning operations or signals.

The direction kind of a parameter in UML correspond directly to the same direction kind in IDL.

| UML | IDL |
|---|---|
| in | in |
| inout | inout |
| out | out |
| return | return |

## Port

There are several different stereotypes that make it possible to map ports into IDL:

- A port marked «CORBAFacet» is mapped to a provides.
- A port marked «CORBAReceptacle» is mapped to a uses.
- A port marked «CORBAEventSink» is mapped to a consumes.
- A port marked «CORBAEventSource» is mapped to an emits or publishes.

To make the mappings more visually distinct, it is possible to apply icons to the ports, as is shown in Figure 187 on page 808. This is done through the icon stereotype.

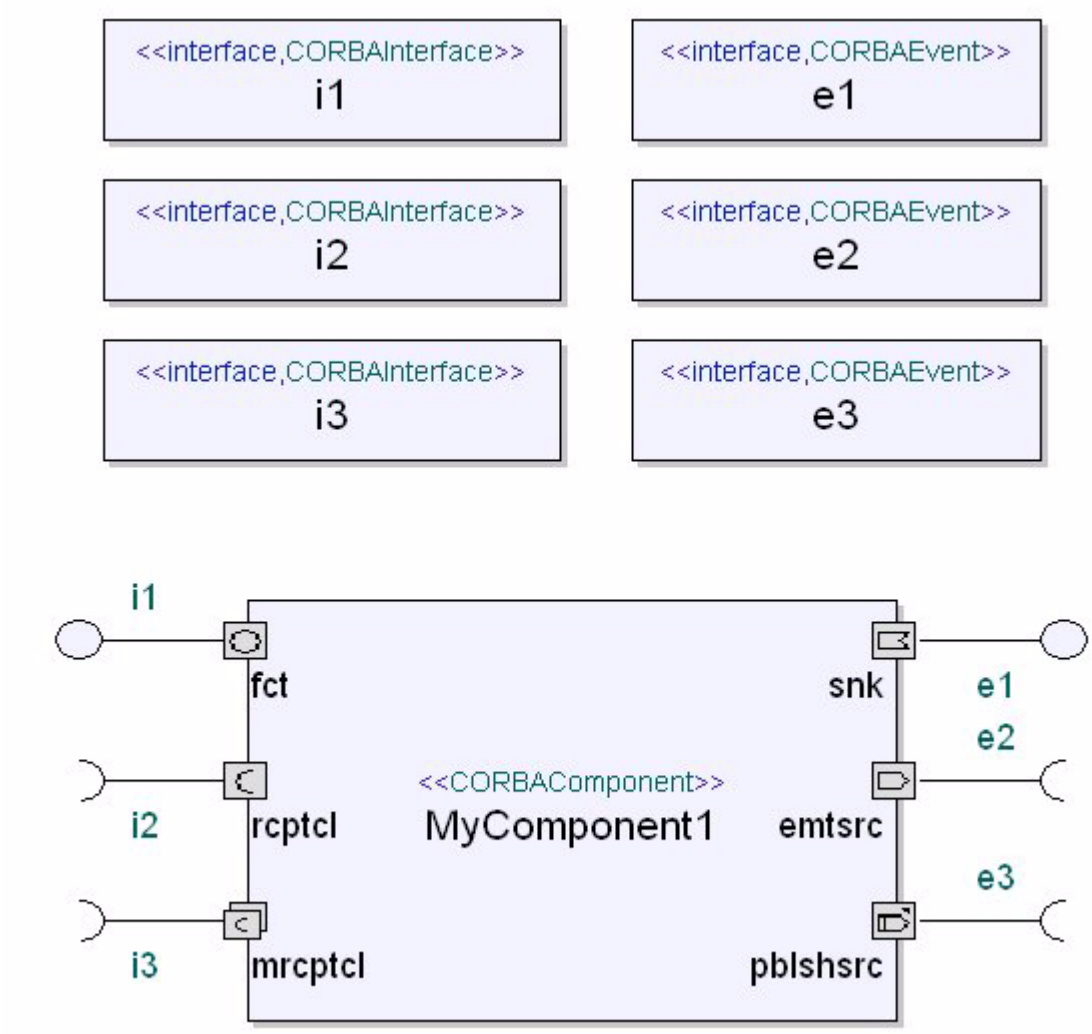



*Figure 187*

This example is mapped to the following IDL:

```
interface i1 { };
interface i2 { };
interface i3 { };
eventtype e1 { };
eventtype e2 { };
eventtype e3 { };

component MyComponent1 {
        provides i1 fct;
        uses i2 rcptcl;
        uses multiple i3 mrcptcl;
```

```
        consumes e1 snk;
        emits e2 emtsrc;
        publishes e3 pblshsrc;
};
```

## Predefined type

The simple predefined IDL types (see "Predefined IDL types" on page 785) are mapped as is.

The template types (see "Predefined IDL types" on page 785) are mapped to the corresponding IDL types.

You can also use the predefined UML data types, which are then mapped according to the following table:

| UML | IDL |
|---|---|
| Natural | unsigned long |
| Integer | long |
| Boolean | boolean |
| Character | char |
| Real | double |
| Charstring | string |

### Restrictions

The tagged value `isOneway` must be set to true when the stereotype `«CORBAOperation»` is applied to signals.

## Segment

A class marked `«CORBASegment»` in UML is mapped to a segment of a component implementation in IDL. No IDL is generated for classes that are not parts of component implementations. Also see "Artifact" on page 798. An example is shown in "Component" on page 800.

## Sequence

A class marked «CORBASequence» in UML is mapped to a sequence. The type of the sequence is specified using an attribute. The name of the attribute is irrelevant; only the type is taken into account. This is an alternative to using the sequence template type.

The following snippet of UML:

```
<<CORBASequence>> class MySeq {
        string dummy;
}

<<CORBASequence>> class MyOtherSeq {
        MySeq x;
}

<<CORBASequence>> class MyThirdSeq {
        MyOtherSeq y;
}
```

is mapped to the following IDL:

```
typedef sequence<string> MySeq;

typedef sequence<MySeq> MyOtherSeq;

typedef sequence<MyOtherSeq> MyThirdSeq;
```

## Signal

A signal marked «CORBAOperation» in UML is mapped to a one-way operation. See "Operation" on page 805 for an example.

**Note**

*This results in the same mapping as a UML operation that is marked* «CORBAOperation»*, where the tagged value* isOneway *is set to true and only in-parameters are used.*

### Restrictions

Signals may only be defined as part of interfaces to be considered for IDL export.

## Struct

A class marked «CORBAStruct» in UML is mapped to a struct in IDL. See "Attribute" on page 799 for an example.

## Syntype

A syntype marked «CORBATypedef» in UML is mapped to a typedef in IDL. This is an alternative to using a type definition.

The following snippet of UML:

```
<<CORBATypedef>> syntype MyOtherType = string;
<<CORBATypedef>> syntype SeqLong = sequence<long>;
```

is mapped to the following IDL:

```
typedef string MyOtherType;
typedef sequence<long> SeqLong;
```

## Type definition

A class marked «CORBATypedef» in UML is mapped to a typedef in IDL. This is an alternative to using a syntype.

The following snippet of UML:

```
<<CORBATypedef>> class MyType {
        long dummy;
}
```

is mapped to the following IDL:

```
typedef long MyType;
```

It is also possible to use a generalization to define a type definition. However, the generalization can only be between elements of the same kinds, such as two data types, two interfaces, or two classes.

The following snippet of UML:

```
<<CORBATypedef>> datatype MyBool : boolean {
}
```

is mapped to the following IDL:

```
typedef boolean MyBool;
```

**Important!**

*Rather than using template types such as sequences and arrays anonymously for attributes and parameters (in the form* `attr: sequence<long>`*), it is better to define type definitions of them, and then use those instead (e.g.* `attr: SeqLong`*).*

# Union

A class marked «CORBAUnion» in UML is mapped to a union in IDL. The type of an attribute marked «discriminator» is mapped to the union discriminator. The name of the attribute marked as discriminator is ignored. The tagged value label of attributes marked «case» is mapped to a case label for the union member.

The following snippet of UML:

```
<<CORBAUnion (.isSimple = true.)>> class U1 {
        long a;
        char b;
        string c;
}

<<CORBAUnion>> class U2 {
        <<discriminator>> Color d;
        <<'case'(.label = green.)>> long e;
        <<'case'(.label = blue.)>> boolean f;
        <<'case'(.isDefault = true.)>> string g;
}
```

is mapped to the following IDL:

```
union U1 switch (long) {
        case 0: long a;
        case 1: char b;
        case 2: string c;
};

union U2 switch (Color) {
        case green: e long;
        case blue: f boolean:
        default: g string;
};
```

### Restrictions

Only attributes of the class that are marked «discriminator» or «case» are taken into account when exporting IDL, unless the union is marked as simple, in which case discriminators and cases are ignored.

# Value

An interface marked as «CORBAValue» in UML is mapped to a value type in IDL.

# Known restrictions

### Anonymous type

Anonymous template types may not work with all IDL compilers.

# 23

# *Import of MSVS Solution files*

This chapter describes how to import Visual Studio Solution files (.sln files) into Tau to visualize component dependencies.

# Overview

The MSVS Solution Importer is used to import Visual Studio (MSVS) Solution files into Tau. Once imported, the components that are generated by the solution and the dependencies between them can be viewed and browsed graphically using various UML diagrams. A component is created in Tau corresponding to one .vcproj file (MSVS project file) in the solution.

The default setting of the importer is to create a one-to-one mapping between a component and a .vcproj file, so the generated graphs can also be used to see the dependencies between .vcproj files. It is also possible to generate artifacts that represent import libraries. This option makes it possible to see all generated components for the Solution. One .vcproj file then results into two artifacts - one that represents the main output of the generation and one that represents its corresponding "import" library.

# Getting started

To import a MSVS Solution file:

- Start Tau and either create a new project or use an existing project.
- Start the **Import Wizard** by selecting **File/Import...**
- Select **Import MSVS Solution** and click **OK**
- Click your way through the MSVS Solution Import Wizard.

**Note**

*Please note that the only version of Visual Studio that is supported by the MSVS Solution importer is Visual Studio 2005.*

# MSVS Solution Import Wizard

This section describes the MSVS Solution import wizard in detail.

## The first step of MSVS Solution import wizard

The first step of the import process is to specify what Solution file to import, what configuration and platform to use and specify additional import options.

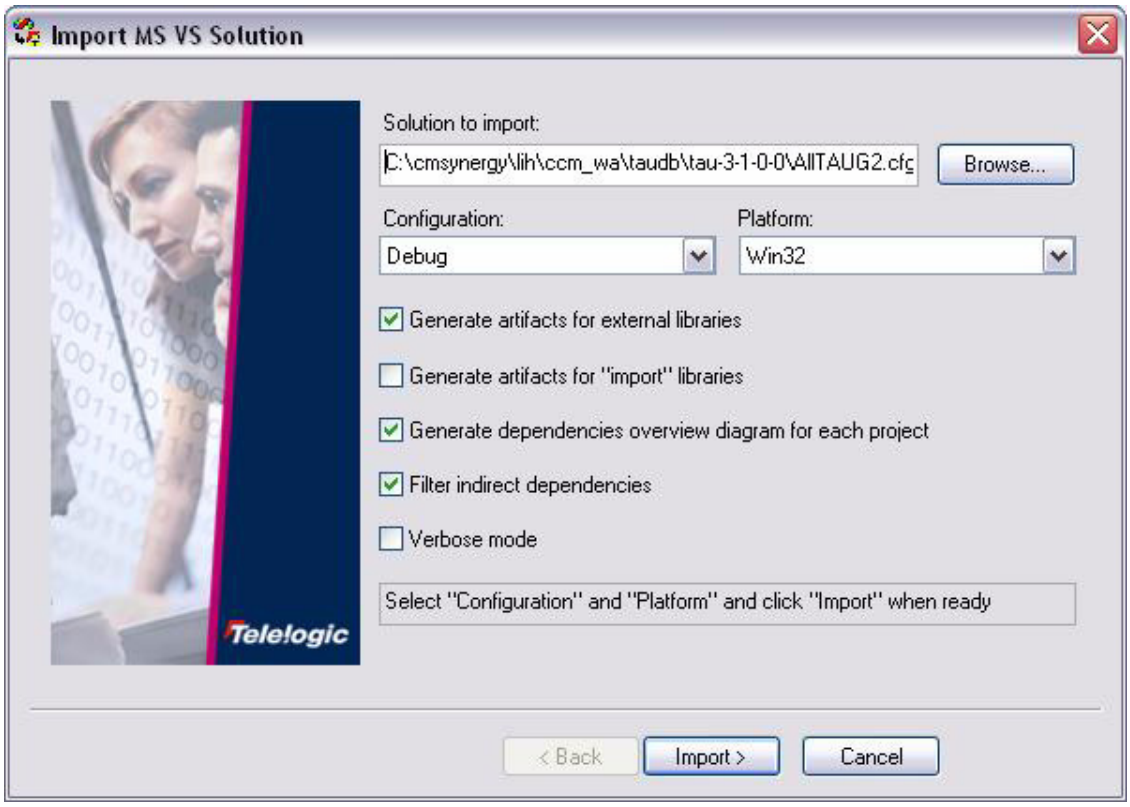



*Figure 188: The first step of MSVS Solution Import wizard*

- Initially the **Browse** button must be selected and a Solution file chosen. When that is done the importer wizard will read the Solution file and will after that show available configurations and platforms for that Solution.
- Select a Configuration and a Platform.
- As an optional step, select or deselect any of the options described below.
- Select **Import**.

If the **Generate artifacts for external libraries** option is unchecked, no artifacts and dependencies to these artifacts are generated for libraries referred to in the vcproj files but not generated by any of the vcproj files in the Solution. This could be Visual Studio libraries, Windows DLLs or other external components used by the vcproj files in the Solution.

If the **Generate artifacts for "import" libraries** is checked and a vcproj file has specified a library in the ***Import Library*** option in the ***Linker -> Advanced*** section of the vcproj file, an artifact representing that library and dependencies to that artifact are generated.

If the **Generate dependencies overview diagram for each project** option is unchecked, only one diagram is generated. That diagram will show all artifacts and their dependencies. If the option is set, the importer will generate one diagram for each vcproj file showing the components and its dependencies for that particular project.

In vcproj files it is required to specify all dependencies needed to build the specified component, so also indirect dependencies are necessary to have in the vcproj file. Importing a Solution file can generate a very complex overview diagram, showing all indirect dependencies. The **Filer indirect dependencies** option optimizes the dependency graph and will not generate the indirect dependencies. Unchecking this option will show all dependencies specified in the vcproj file.

The MSVS Solution importer requires that you have Visual Studio installed, because the importer uses Visual Studio to extract the information from the Solution and vcproj files. If the **Verbose mode** option is checked, a detailed description of what is done during import is in the importer log.

## The second step of MSVS Solution import wizard

The second step in the import wizard is actually only a progress page showing the current status of the import. As soon as you get to this step, the importer starts to communicate with Visual Studio to extract information from the Solution and vcproj files.

During import it is possible to abort the import by selecting **Cancel**. Selecting **New Import** will show the first step of the import wizard again and a new import can take place

Pressing **Finish** will close the import wizard.

## Result of import

The MSVS Solution importer will generate a package with a <<MSVSSolution>> stereotype set. This stereotype has the same tagged values as the options in the first step of the importer wizard.

All artifacts are placed inside the package. Neither the package nor the entities in the package are saved in a file automatically, that has to be done by the user. To do this, right-click on the top-level package and select "Save in New File...", enter a valid file name and press OK.

# Re-import of a Solution

Right clicking a package that is the result of a MSVS Solution import (having the <<MSVSSolution>> stereotype set) enables the context menu option **Update model from Visual Studio solution**. Selecting this option will redo the import and everything inside that package will be removed and replaced with the content of the new import.

# 24

# *File/Folder Importer*

This chapter describes how file system entities (files and folders) can be imported into Tau to obtain a model representation of them. The File/Folder importer supports exensibility modules which can be used for processing imported files/folders in order to extend their default model representation.

# Overview

The File/Folder importer is a tool for importing files and folders into a UML representation. In its basic form the resulting model consists of file artifacts (representing imported files) and packages (representing imported folders).

The importer also supports the processing of imported files and folders by using domain specific **extension modules**. An extension module can for example analyze imported files of certain kinds and create dependencies between the corresponding file artifacts in order to visualize some aspects of the file contents. It is possible to add custom extension modules to be used with the importer.

## Getting Started

You access the File/Folder importer by using the Import Wizard:

- Create a new project or use an existing project.
- Start the **Import Wizard** by selecting **File/Import...**
- Select **Import Files/Folders** and click **OK**
- Click your way through the File/Folder Import Wizard.

# File/Folder Import Wizard

This chapter describes the File/Folder import wizard in detail.

## The First Step of the File/Folder Import Wizard

The first step of the import process is to specify which files and/or folders to import from the file system.

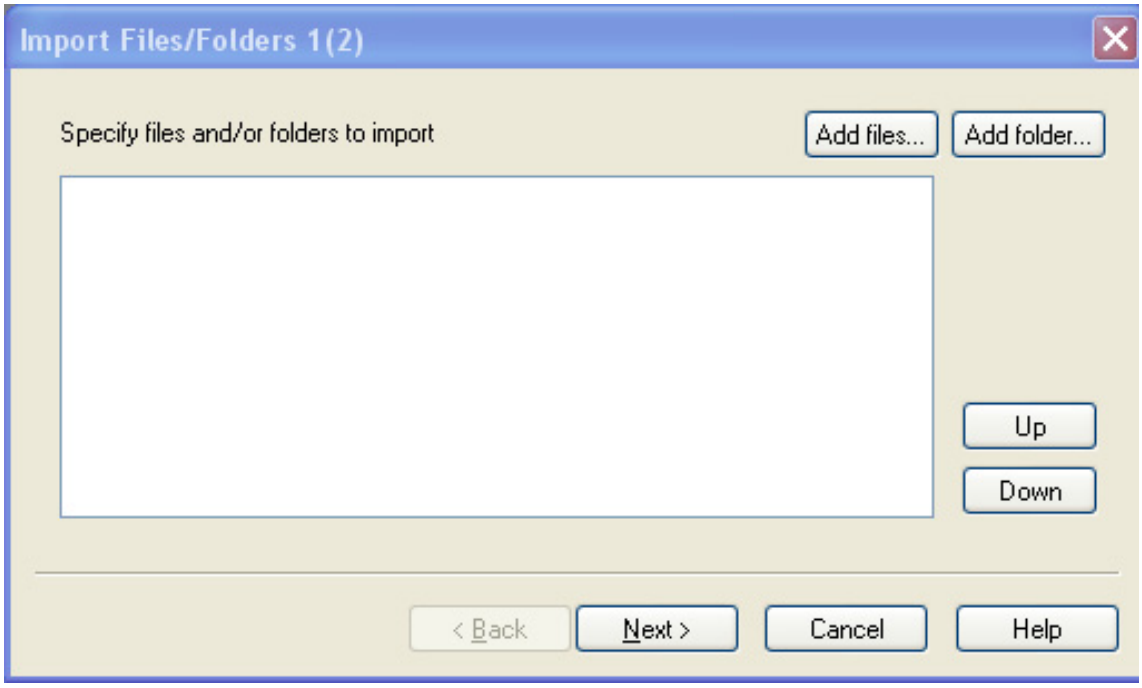



*Figure 189: The first step of the File/Folder Import wizard*

- Use the **Add files...** button to open a standard dialog for selecting files from the file system. The dialog allows multiple files to be selected.
- Use the **Add folder...** button to open a standard dialog for selecting a folder from the file system.
- Double-click in the white area to directly type the path of the file or folder to import. This can be useful if you want to use wildcards in the path (such as `C:\docs\*.txt`).
- Select an item in the list and press the **Delete** button in order to remove an entry from the list.
- Use the **Up** or **Down** buttons for moving items up or down in the list. The order in the list defines the order in which the paths will be imported.

Paths in the list may contain URNs. They may also be relative. A relative path is interpreted against the location of the project file.

When you have completed the list with files and folders to import, proceed to the next wizard page by pressing **Next**.

## The Second Step of the File/Folder Import Wizard

The second step in the import wizard presents a list of available extension modules. See Built-in Extension Modules for a description of those extension modules that are shipped with Tau.

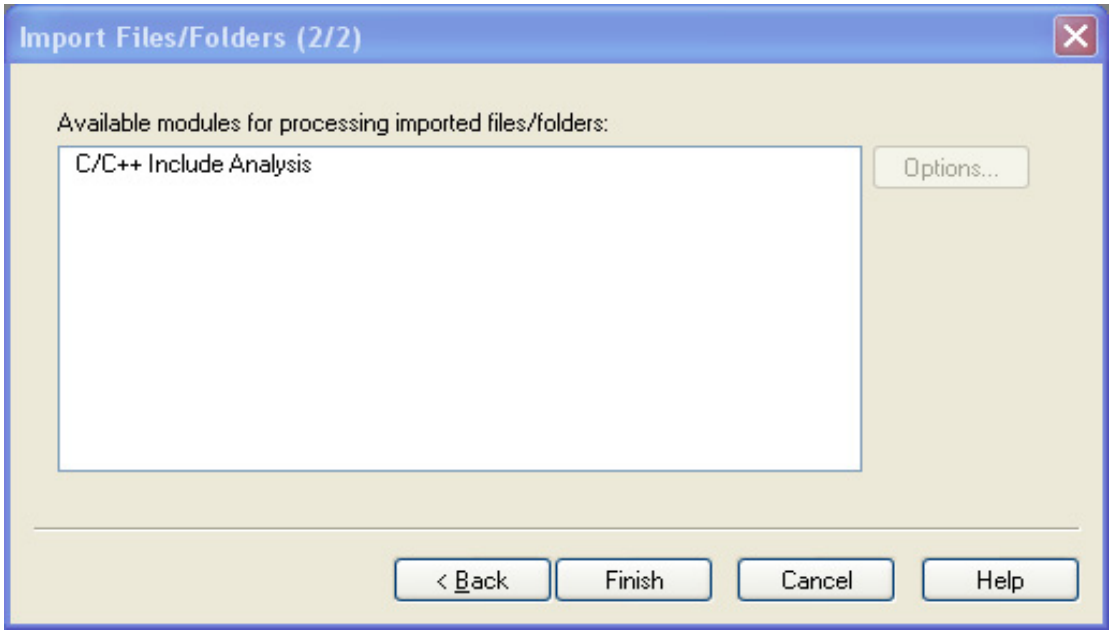



*Figure 190: The second step of the File/Folder Import wizard*

You may select one of these extension modules to process imported files and folders. If the selected extension module has some options these can be edited by pressing the **Options...** button.

Pressing **Finish** will close the import wizard and perform the import.

## Result of Import

The File/Folder importer places generated file artifacts and packages in a top-level package called `Imported Files/Folders<index>`, where `<index>` is an index to make the package name unique.

After the import you may use diagram generators (see Generate Diagram) in order to visualize imported elements including additional information built by the selected extension module.

**Hint**

*A file artifact representing a text file can be double-clicked in Tau in order to open the file in the text editor.*

## Reimport

It is possible to perform a reimport of the files/folders by following these steps:

- Select the package resulting from the import in the Model View.
- Right-click and select the command **Update Model for Files/Folders** in the context menu.

Options selected in the File/Folder import wizard are stored as tagged values on the package created by the importer. The Properties Editor may be used to edit these options prior to performing the reimport command, in case you want to change any of the options.

# Built-in Extension Modules

This chapter describes the extension modules for the Files/Folder importer that are shipped with Tau. See Adding Extension Modules for the File/Folder Importer for information about adding additional custom extension modules.

## C/C++ Include Analysis

This extension module performs an analysis of `#include` directives within imported C/C++ header files (files with suffix `.h`). Each `#include` directive is mapped to an `<<include>>` dependency between the file artifact that represents the including file, and the file artifact that represents the included file.

Use the `includePath` option (see Options) to define the paths where to look for files that are included using a relative path.

### Generating a Dependency Diagram

In order to visualize the include dependencies in a diagram a diagram generator may be used after the import:

- Select the package created by the File/Folder importer in the Model View.
- Right-click and select **Generate Diagram / Generate Dependency View for Contained Definitions**.

The resulting class diagram will show the file artifacts and their include dependencies.

### Options

The C/C++ Include Analysis extension module supports the following options:

**`includePath`**

A list of paths of where to look for included files. These paths are typically the same as is used with a C/C++ preprocessor when preprocessing the files.

# *UML to Applications*

The chapters under UML to Applications describe how to build applications based on UML models. The information in this section is common for all UML projects using the supplied code generators.

# 26

# *Building and Code Generation Overview and Examples*

In this chapter the following information is available:

- A user guide to building applications from UML models with the C, C++ and Java code generators.
- An overview of the build and code generation process in Tau, with emphasis on the C Code Generator and AgileC Code Generator.
- An introduction into how to use UML composite structure diagrams with a special focus on the support provided in the C Advanced and AgileC Code Generator. The part-whole relationships and how to use these concepts together with dynamic creation of instances.
- The CPtr type which is intended to provide features similar to a low-level pointer type in UML modeling. It is intended mainly for UML applications that interface with manually coded C or C++ components.
- The threaded integrations with Real-time operating systems.
- Examples of "C" applications generated with Tau, illustrating how to interface the generated code with the environment and how to deploy it to target.

**See also**

"C++ Support in Tau" on page 1515.

Chapter 42, Java Support.

# Building Applications with Tau

## General

The input to the build process from a UML model to an executing application consists of the following:

- The UML model itself.
- Build and code generation settings described in Using Build Artifacts.
  - Several build artifacts can be defined for each model.
- Configuration, allowing a group of build artifacts to be processed in one build.
- External (user provided) code, definitions and libraries that should be included in the build process to be compiled and linked with the generated code.

## Building in Interactive or Batch Mode

Building can be done either from the Tau graphical user interface, or in batch mode from the command line prompt.

**See also**

"Interactive Build Interface" on page 932

"Batch Build Interface" on page 945.

## Using Build Artifacts

A Build Artifact is an artifact in the UML model, with a build stereotype attached (i.e. «build» or any of its children), and holding properties dedicated to the build process. Build artifacts can be placed anywhere a class is allowed to be placed, and are not subject to the normal scope rules regarding the root object it refers to.

A build artifact contains the following information:

- Build Root. A build root defines an element in the UML model delimiting the scope of the build.

- Build Type. A build type defines which code generator to use for the build.
- Build Settings. Build settings define settings to use by the code generator. These settings can be specific for a particular code generator, or generic.
- Target Directory, specifying where to put the files that are produced by the tools invoked by the build.
- Error Limit. If the number of errors associated with a build exceeds this limit, the build is aborted.

### Adding a build artifact

Adding a build artifact to the model is easiest done in the following way:

1. Make sure the appropriate build type is enabled.
2. Right-click the model element to use as build root, select the build type, and then **New Artifact**
3. Use the Properties Editor to add additional stereotypes required to specify advanced options pertaining to the build type. The **Filter** drop-down menu contains the currently applied stereotypes.

### Accessing and specifying properties defined in a build artifact

Each of the build artifacts holds one instantiation of a build stereotype. The build stereotype defines the build settings relevant for the actual build type.

To access these attributes:

1. Right-click the Build Artifact in the workspace window.
2. Select **Build Settings**. This opens the Properties Editor.
3. Click **Stereotypes** if required, in order to add additional build stereotypes that should be instantiated on this artifact.
4. Select the build stereotype from the **Filter** drop-down menu.
5. The properties (build settings) are now displayed and can be changed by the user. Changes are committed as they are applied in the editor.

### Location of a build artifact

The location of a build artifact in the model view has no semantic meaning. The build artifact can be moved to any valid location of the user's preference, for example into the build root or any other location of the workspace area that feels convenient.

### Mandatory use of artifacts

At least one build artifact must be present for any build to take place.

– When attempting to build a model without a build artifact, a build artifact will be created by a Build Wizard where you will be prompted to specify the required information. The build artifact is also inserted into the active Configuration, so that a build of the configuration will include a build of that artifact

### Multiple build artifacts – configurations

For each UML model, multiple build artifacts can be used, so that different applications can be generated from one UML model. For instance, it may be convenient to have a build artifact that produces a Model Verifier for debug purpose on host computer, and a build artifact that builds an application for deployment on target.

Furthermore, multiple build artifacts can be grouped into a Configuration as a convenient way so specify the build of multiple artifacts through one user operation only.

Builds can be initiated with a selection. The selected objects are used to form a list of build artifact using the rules:

- If an object is a build artifact it is added to the list.
- If an object is not a build artifact, the set of build artifacts that "manifest" the object (or any parent objects) is formed
- If the set is empty the "Build Wizard" appears to aid the user
- If the set contains exactly one build artifact, that is added to the list
- If the set contains more than one build artifact, the "Build Wizard" appears to aid the user

For “Build” and “Verify” operations the list of build artifacts is then appended with the build artifacts to which any build artifact already in the list has a dependency. This means that if for example build artifact “A” depends on build artifact “B” and you build “A”, “B” will also be built.

The list of build artifacts is ordered with respect to dependencies before the build operation starts. If there are circular dependencies, the resulting order is not defined.

**See also**

“Using Configurations for Build” on page 851.

## Using Thread Artifacts

A Build Artifact is optionally composed of a number of **thread artifacts**. A thread artifact is an artifact with the stereotype «thread» attached. The Class diagram editor is used for modeling such classes.

A thread artifact can be used in several build artifacts.

Thread artifacts are used exclusively with the AgileC Code Generator and C Code Generator.

For the C++ Application Generator and the Java code generator threads are defined and manipulated programmatically using utilities in the TOR library.

### Examples of use

How to used thread artifacts is explained by studying the examples of use of thread artifacts provided at the end of this chapter

**See also**

“Application Examples” on page 924

## Using File Artifacts

In order to specify how model elements should be implemented on files, and also to add external components and specify “make” dependencies between sources and targets, you use file artifacts.

**Note**

*File artifacts cannot be used to specify implementation aspects or dependencies for applications generated by Model Verifier, C Code Generator and AgileC Code Generator. Instead, you must use the code generation settings dedicated to the associated code generators.*

### Using file artifacts to control C++ code generation

A typical application area for file artifacts is to specify the mapping scheme between classes and C++ code files generated by the C++ Application Generator, using the stereotypes C++ implementation file and C++ header file. By overriding the mapping scheme used by the C++ Application Generator (1:1 mapping is used by default, meaning that each class will be mapped to one C++ source file and one C++ header file), you can control in detail how the tool stores the generated code on files.

This fine granularity may simplify configuration management in a multi-user environment, and also will likely simplify the task of the C++ compiler. However it may be convenient to group classes that belong together, or classes that contain parts of the model that are stable and where few changes can be expected into packages and create a few C++ file artifacts that are associated with these classes or packages.

### Using file artifacts for C++ roundtrip

C++ file artifacts also 'connect' the classes with the generated code and this connection allows to synchronize the model and the code (by using the Update Configuration command) in order to propagate changes done by the user to the C++ files after they were generated by the code generator.

The desired level of granularity when manifesting C++ files depends on configuration management considerations, and on the level of flexibility that is needed. See "Using file artifacts to control C++ code generation" on page 837.

### Use file artifacts to specify C++ targets and make scheme

File artifacts can be used to specify that the model should be compiled and linked into a number of targets. Targets denote either libraries or executable applications. This allows to split up the resulting application into a number or libraries.

To specify a C++ target:

1. Add a file artifact specialized either as a library or as an executable.
   - The library file artifact can be further specialized to denote a statically or dynamically linked library.
2. Add dependencies from the library/executable to its 'sources'
   - library file artifacts should depend on all C++ implementation file artifacts that are part of the library.
   - executable file artifacts should depend on a C++ implementation file artifacts that manifest suitable UML elements – for instance an operation manifested as the `main()` function in the C++ code.
   - executable file artifacts should also depend on all the library file artifacts that are needed for the application to execute properly.

As a result of this, a make file that defines the sources, dependencies and how to link the libraries is created for you. The generated make file is adapted to support Windows and UNIX flavors of "make" and can be augmented with user-defined code, by using the attributes in the Make settings stereotype.

### Using file artifacts to specifying C++ objects

Tau in its present version does not support customizing the make dependencies between C++ implementation files and arbitrary object files. Make dependencies assume preserving file base names.

### Using file artifacts when generating Java

Just like for C++ the Java code generator uses file artifacts to define the mapping between model elements and files. However, contrary to C++ the Java language puts several constraints on the contents of Java source files. For example it is not allowed to have more than one top-level class in a Java file, and its name should match the name of the file.

Because of these constraints file artifacts are usually not manually managed when generating Java code from Tau. Instead file artifacts are added automatically as needed when performing code generation or roundtrip operations.

## Example of Use of File Artifacts

In this section is discussed how you could use file artifacts to customize the make dependencies and the manifestation of model elements on C++ files.

Consider a model that is specified to be deployed as an application in the following way:

- The model has a main() operation, from which is generated the C++ code for the `main()` function of the application. This code is to be put on separate `main.cpp` and `main.h` files
- The has a package, say “a”, containing global definitions. The package contains a class “a” that should be compiled as a library stored on `a.lib` The generated source code is to be manifested by the files `a.cpp` and `a.h`
- The model also includes a class, say “support”, which is defined and implemented externally, The implementation and definition of the class is available on the existing files `support.cpp` and `support.h` No code should be generated for the “support” class, but the files should be present in the compile and link scheme, and also must have dependencies to `a.h` where global definitions are found.
  - To specify that the class “support” is defined and implemented externally, its attribute **External** must be set to `true`
- The files above should be compiled and linked into the resulting application `my_application.exe`

### Creating the file artifacts

Start by creating the required file artifacts, for each of the files identified above (`main.cpp` etc.). For each file, proceed as follows:

1. Create a file artifact (**New -> Artifact**).
2. Name it in accordance with the name of the file you are processing.
3. Using the Properties Editor, specialize the file artifact as either a C++ implementation file, a C++ header file, an executable or a library.
   - Click **Stereotypes** if required in order to add the required stereotype.

The result after all files have been specified is depicted in Figure 191 on page 840.

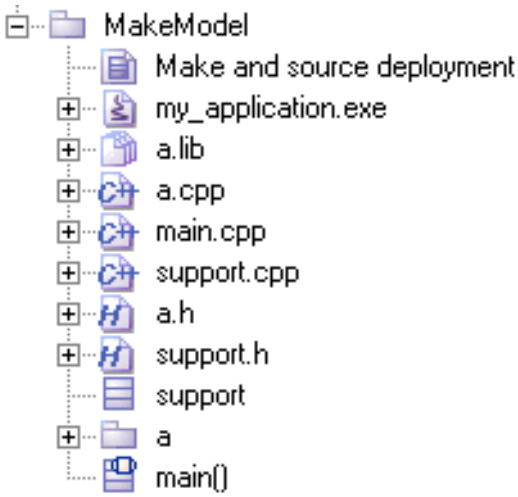



*Figure 191: Model View showing the required file artifacts*

### Specifying the make dependencies

You now need to specify the make dependencies, to ensure a correct and reliable compilation and linking scheme. To do this:

1. Create a class diagram, e.g. “Make and source deployment”.
2. Drag and drop the file artifacts that you created in the previous step to that class diagram.
3. Draw dependency lines that represent the make dependencies between sources/libraries/executable:
   - From `my_application.exe` to the library `a.lib`
   - From `my_application.exe` to the external code `support.cpp`
   - From `a.lib` to its implementation `a.cpp`
4. Draw dependency lines that represent the make dependencies between definitions and implementations. These dependencies should have the «include» stereotype added.
   - From `a.cpp` to `a.h`
   - From `support.cpp` to `support.h`
   - From `main.cpp` to `a.h` (since `a.h` contains global definitions)
   - From `support.h` to `a.h` (idem)

The resulting class diagram with the make dependencies specified should look like Figure 192 on page 841

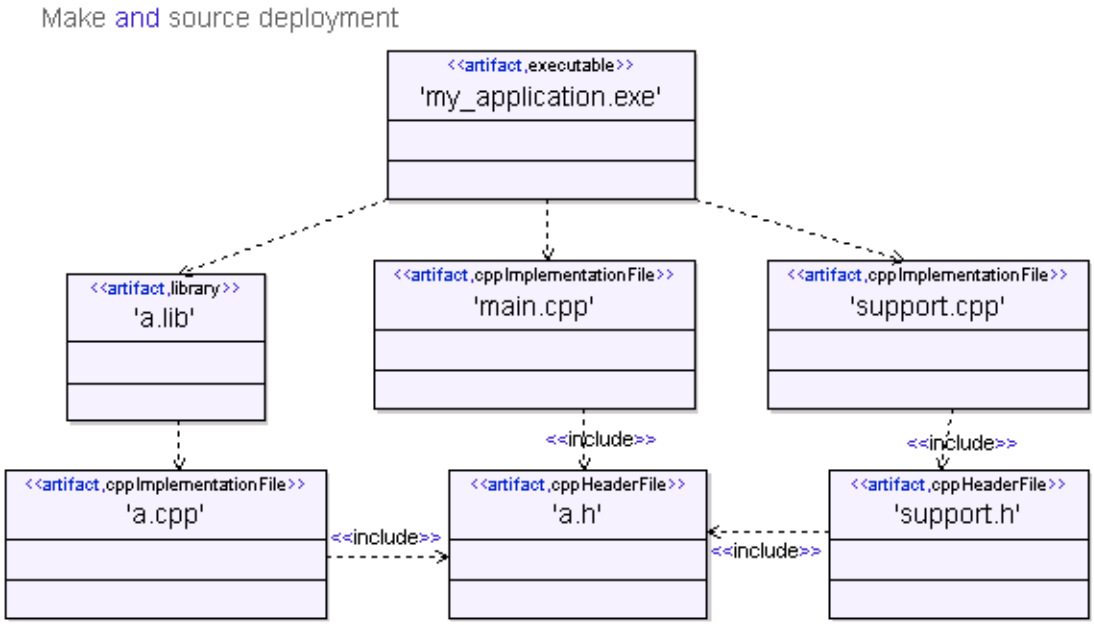



*Figure 192: The make dependencies for the files building up the application*

### Specifying the manifestations

The last task is to specify how the model elements “main()”, “a” and “support” should be connected to the files manifesting their implementations and definitions. This can be done in the class diagram used to specify the make dependencies, if you think it is handy to have all dependencies and manifestations depicted in one single diagram.

- The definition of class “a” is manifested by `a.h` (by drawing a «`manifest`» dependency, and its implementation is manifested by `a.cpp` (by drawing a «`manifest implementation`» dependency.
- In the same way, the definition and implementation of the class “support” is manifested by `support.h` and `support.cpp`
- The “main()” operation is manifested by its implementation `main.cpp` only (there is no `main.h`).

The result is depicted in Figure 193 on page 842.

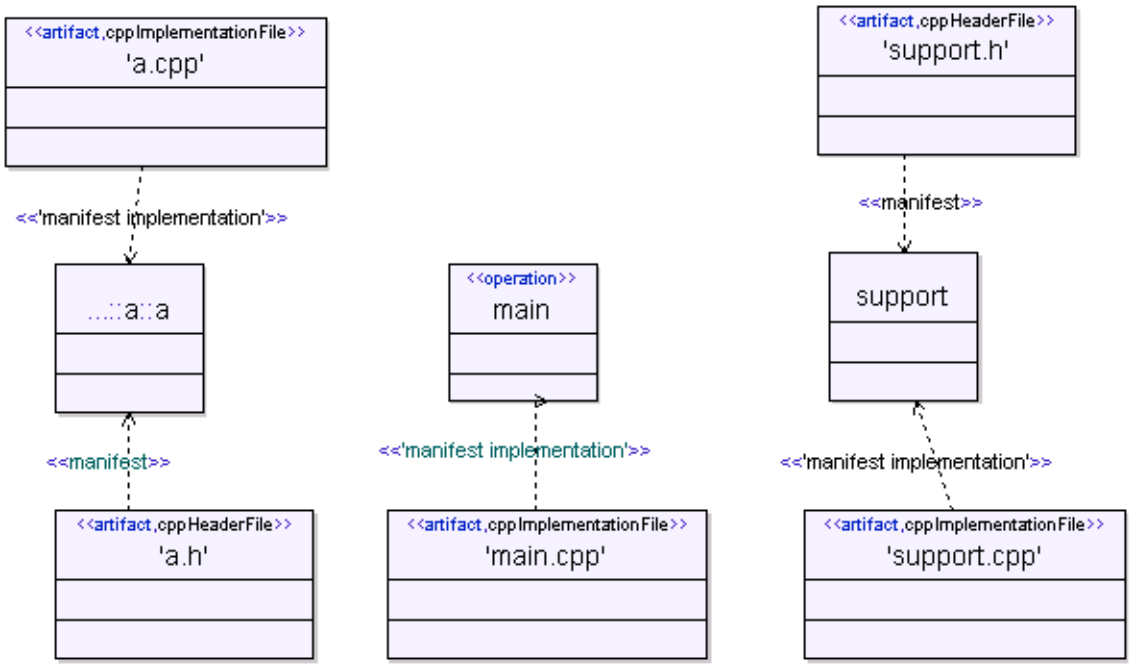



*Figure 193: Model elements manifested by C++ files.*

# Using Build Roots

For each Build Artifact, a Build Root has to be provided. A build root delimits the 'sub-model' that will be built – the model elements that are defined by the build root, or referred to and included according to the UML language scope rules.

– The build root of the artifact is identified by a «manifest» dependency from the build artifact to the element it uses as build root.

### Changing the build root

A build root is designated when creating the build artifact using the Build Wizard.

Should you want to change the build root, the most straightforward way to change it is the following:

1. Right-click the build artifact and from the shortcut menu select the command **Select Build Root**.
2. In the dialog that is issued, specify the model element of your choice (a package or a class) to use as root for the build.

3. Close the dialog using the OK button.

### Suitable build roots for C build types

The following elements may be suitable to use as build root for build artifacts that invoke the Model Verifier, C Code Generator and AgileC Code Generator:

- The topmost active class
- A package

**Note**

*Using a package as build root will build a library rather than an application. The attribute Target should be set to Library instead of Executable. Build of libraries is discussed in detail in Chapter 29, Guidelines for Large-Scale Application Development.*

### Suitable build roots for C++ build types

The following elements may be suitable to use as build root for build artifacts that invoke the C++ Application Generator:

- The topmost active class
- A package
- A class

### Suitable build roots for Java build types

The following elements may be suitable to use as build root for build artifacts that invoke the Java code generator:

- The topmost active class
- A package
- A class

## Using Build Types

To manage the large number of combinations that are the result of all possible settings for the build process and the components that are involved, the notion of **build type** is introduced: a build type defines in essence which code generator a Build Artifact uses.

Tau supports build types that use the **Model Verifier**, **AgileC Code Generator**, **C Code Generator**, **C++ Application Generator** and **Java Code Generator**. In addition to these 'true code generators' Tau also allows you to use the build types **Makefile Generator** and **Make**, which generate a "make file" and invoke the "make" utility.

**Note**

*The Model Verifier is, technically speaking, a specific application of the C Code Generator, in which the generated C code is instrumented to provide what is required to support for debugging, tracing and simulation at UML level (such as symbolic information, command-line interpreter and a graphical user interface). A Model Verifier build follows a similar build flow as when building a C application, but the resulting applications differ.*

Below follows an overview of the settings that are available to the user, for each build type:

### Enabling (loading) a build type

In order for a build type to become available, its corresponding **Add-in** must first be activated. This is done by the project wizard when creating a project.

Should you want to enable another build type than the one that is preset when creating the project, it can be done by loading the corresponding Add-in. From the **Tools** menu select Customize. The Add-Ins tab displays the currently available add-in modules. Each of these modules can be made available individually.

1. Check the toggle corresponding to the build type of your choice:
   - Select **Make** to enable the use of the "make" utility from within Tau,
   - **Makefilegen** to enable the Makefile Generator,
   - **AgileCApplication** to enable the AgileC Code Generator,
   - **CApplication** to enable the C Code Generator,
   - **ModelVerifier** to enable the Model Verifier
   - **CppGen** to enable the C++ Application Generator.
   - **JavaApplication** to enable the Java Code Generator.

2. Clicking **Close** loads the add-in modules that are selected in the dialog, and adds their profiles to the model. Information about which profiles are loaded is displayed in Script tab in the Output window.

   As a result of loading a profile, the associated build type/code generator and the settings it supports now becomes available for selection in the **Filter** drop-down menu of the Properties Editor.

### Specifying the build type

The build type can be specified in either of the following ways

- When attempting to build a model not having any build artifact, the Build Wizard is activated and prompts the user for a build type.
- With the shortcut menu activated on a model element that is possible to use as Build Root, select the desired build type and then select the command **New Artifact**. This creates a build artifact with the desired build type.

### Considerations related to projects with multiple build types

For projects where several Add-Ins have been loaded (see Enabling (loading) a build type, above), you may gain access to multiple build types. By adding additional artifacts to the Configuration this allows you to build a variety of applications from the model, using one single build command that builds multiple build artifacts.

You can for instance have a build artifact that generates a Model Verifier and one build artifact that generates an AgileC Code Generator application, and build both applications with one build command only.

**Note**

*You should not have multiple build types added to one build artifact. Having multiple build type stereotypes attached to one build artifact means that the build type that is actually used is undefined. Therefore you should Remove not used build types.*

### Remove not used build types

To remove a build type from a build artifact,

- Select the artifact and open the Properties Editor
- Use the **Stereotypes** button to open a dialog that displays the stereotypes currently added to the build artifact
- Clear the check box for any build type that should be removed and close the dialog.

### Changing the build type

It is not recommended to change the build type for a build artifact since the stereotypes are added with default values for their attributes, while the values for attributes that have been removed are lost and no longer possible to restore. It is instead recommended to create a new build artifact with the desired build type. It can however be done by using the Properties Editor, clicking **Stereotypes** and checking the stereotypes.

## Performing Separate Builds

By defining a Build Root different from the topmost active class (the class without owner), you can break down a build scheme into separate builds. For instance you may want to be able to generate code and compile a package only, instead of having to generate code for the whole model, enforcing a build of all packages even though no global changes have taken place since the last build.

For a successful separate build to take place is depending on if the 'separate build' feature is supported by the code generator which is defined in the Build Artifact.

- To break down your build into separate builds, create as many build artifacts as required and associate each of them with a suitable build root.
- Models that are built using C++ Application Generator can also be modularized by defining the C++ targets that should be managed by the Makefile Generator (executables or libraries).

**See also**

"Use file artifacts to specify C++ targets and make scheme" on page 837

"Building a Selective Model Element" on page 850.

## Using Build Settings

Build settings can either be generic for any build type (regardless of which code generator it uses), or be specific for a given Build Type.

### Build stereotypes

There are different types of build stereotypes, each of them corresponding to a different code generator. The values that are defined by each build stereotype represent the build settings supported by the corresponding code generator.

All of the build stereotypes inherit from the stereotype named «build»

- It may be convenient to check the **UML Basic Editing** option **Show stereotype instances** in order to visualize in the workspace window which stereotypes are attached to a build artifact.

### Accessing and changing build settings

Once a stereotype has been added, its properties – the corresponding build settings – can be accessed, and changed if required.

- To display and change these settings, the Properties Editor is used.

### Removing build settings

By removing a stereotype from a build artifact, all its properties are removed and the corresponding build settings are reverted to their default values.

**See also**

Section "Stereotypes and attributes" on page 1984 for a reference to the build settings available to the user.

# Specifying C Targets

### C target name

For the build types AgileC Code Generator, C Code Generator and Model Verifier, the base name of the target, that is to say the base name of executable file, is automatically derived from the name of the Build Root that the Build Artifact is referring to.

### Specifying and "make" of C targets

The make file used to compile and link such applications is at present time not fully embodied by the Tau user interface. The makefile contents is however adaptable through the use of "Make template files"

A stereotype with C code generation settings is available to specify which Make template file to use, but how to customize the make template file must be done outside the tool. To learn more about to customize make template file, see the chapter about the C Code Generator runtime libraries, section "Library files" on page 1065 in Chapter 33, *C and AgileC Runtime Libraries*.

# Specifying C++ Targets

### C++ target name

Unless you specify the name of the C++ target, the name of the resulting application is

- `application.exe` on Windows
- `application` on UNIX.

### Specifying and "make" of C++ targets

The default scheme (where all classes are mapped 1:1 to C++ files and then compiler and linked to one monolith) can be overridden by tailoring an arbitrary number of C++ targets. This is achieved by using file artifacts.

**See also**

"Use file artifacts to specify C++ targets and make scheme" on page 837

## Target Directory

The target directory is the location on the file system where all files are written for a Build Artifact.

Unless specified in the build artifact, the name of the target directory is the same as the build artifact.

For each build artifact, the attribute **Target Directory** can be used to override the naming convention described above.

**Hint**

*The name of the implicit target directory is subject to be changed in future releases. It is recommended to use an explicit target directory. If possible write code and make rules so they do not rely on the names of generated directories.*

### Target directory and make template files

A target directory can be specified either as an absolute path or a path relative to the project directory. When specifying relative path for a Make template file file, the location of this file should be given relative to the target directory.

The make template file typically contains references to source code files. The contents of the make template file is copied into the generated makefile. These references to source code files should in the case of a Bare application be made relative to the target directory.

When a threaded system is built, a sub-directory to the target directory is created, which name is derived from the Build Root. The generated code, including the make file, is placed in that sub-directory where it will be compiled and linked.

## Error Limit

If the number of error messages associated with the build artifact exceeds this number, the build is aborted. When a build is aborted the total number of errors may exceed this number as all pending error messages are processed. If the error limit is set to zero, builds are not aborted regardless of the number of error messages generated.

## Building Using Build Artifact

An artifact build is started by right-clicking the Build Artifact of the user's choice, selecting **Build (< build type >)** on the shortcut menu, and the appropriate item on the sub-menu:

- Check (performs semantic check)
- Generate (as Check, and then generates code)
- Build (as Generate, and then compiles and links)
- Launch (as Build, and then starts the application – this command is supported for Model Verifier only)
- Update (synchronizes with external C++ code, used for roundtrip engineering)
- Clean (executes `"make clean"`)

The tool reads the generic settings and also the settings that are tagged for the build type defined in the Build Artifact (e.g. code generator specific settings), and submits the Build Root defined by the build artifact to the code generator.

## Building a Selective Model Element

A build of a selective element is initiated by right-clicking the package or class to be built. In the shortcut menu, one item is present for each active build type.

- For each of these build types, there is a list of available build artifacts that manifest the selected element.
  - For each of the build artifacts, the available build commands is listed in a sub-menu. See "Building Using Build Artifact" on page 850.
- There is also a menu choice **New Artifact**. This menu choice creates a new build artifact with the selected element as Build Root.

# Using Configurations for Build

### Building a configuration

Ordering a build of a Configuration is performed in the following way:

1. Make sure the configuration that should be built is the active configuration in the project tool bar.
2. On the Build menu, select the appropriate menu choice (**Check**, **Generate**, **Build**...)

### Adding or removing a build artifact to/from a configuration

Build artifacts are added (or removed) to a configuration by using the project settings page.

1. Select the configuration that you want to add (remove) artifacts to in the project tool bar.
2. On the **Project** menu, select **Settings**. A dialog is displayed.
   - The tree that is displayed left-most in the dialog lists the files contained in the project. This information is however not used when building UML models and should be disregarded from.
   - The lists that are displayed beneath the **Build** tab show all build artifacts that are currently contained in the active configuration.
3. Select the artifacts of your choice and use the left and right arrow buttons to add or remove them from the configuration. These lists support selection of multiple items, so you can move several items in one operation.
4. Once the artifacts that you want to be part of the configuration are present in the left list, click **OK** to confirm.

**Note**

*Clicking **Cancel** has the same meaning as clicking OK. Hence, should you want to cancel the operation, the contents of the configuration should be reverted using **Undo** after the dialog is closed.*

# Errors and Warnings from Build

The build process from UML model to executing application includes several phases. Each of these phases will cause the tool to print messages in the Build tab in the Output window. The Error Limit can be used to abort a build when a certain number of errors has been reached.

- An initial analysis phase checks the model for semantic correctness in the context of the build type that has been chosen. This phase may return warnings, and sometimes errors, which should in most cases correspond to errors and warnings from the **Check Selection** or **Check All** commands.
- After the analysis phase is completed and the tool has determined that the model and build settings are suitable to build, the tool launches the code generator. Since there may be restrictions in the support of advanced UML constructs when generating code that the semantic checker does not detect, additional errors may be displayed in this phase.
- The make file generator defined by the build type creates a make file, and then executes “make”. In case default settings are used, meaning that no user defined make template file or make settings have been defined, this phase should never report any errors. Diagnostics reported by the “make” utility are printed in the Output window as well.
- As a result of executing “make”, the code generated from the model and user-provided code are compiled. In virtually all cases, code that is generated by the tool should always compile successfully. Compiler messages are echoed verbatim to the output window.
- Lastly, the object files are linked with the run-time library and with user-provided libraries. This phase should not report any errors provided that entities that are declared “external” in the UML model are also available in the libraries with correct type and name. Linker messages are displayed in the output window.

The severity of each printed message is one the following: Information / Warning / Error.

**Note**

*Sometimes the semantic checks that are performed prior to code generation do not report the same result as Check Selection or Check All would do. It is important to understand the difference between how these semantic checks are performed.*
*Check Selection and Check All operate on the in-memory model of Tau. Check Selection checks a selected model entity, while Check All checks the entire model. Note also that Check All starts by unbinding the entire model, in order to also detect binding inconsistencies and problems.*
*The semantic checks performed by the code generators are basically the same, but the checks are performed on the copy of the model that was loaded by the code generators. This is typically a subset of the original model. Also, a set of code generator specific checks will also be performed in this case.*
*To avoid confusion it is always best to use Check All prior to code generation to ensure that the model is correct. If Check All reports errors, fix these before attempting code generation.*

**See also**

"Error and Warning Messages" on page 421

"Restrictions in UML Support when Building C Applications" on page 951

# Makefile Generator

The Makefile generator is in principle a code generator that operates like other code generators with the difference that it is able to read a single model (.u2) file rather than an entire project (it can also read a project).

## Usage

The Makefile generator is very configurable using a large set of "generator parameters". These parameters govern how internal transformations are made, names are mangled, and how rules and commands are written to the Makefile.

The Makefile generator can be used in two ways;

**explicit**

The user creates a make-model in project and creates a Build Artifact stereotyped by "Makefile generator". The build artifact has a dependency to the make-model that is stereotyped "manifest". This is how all code generators work.

**implicit**

This is the "normal" modus operandi. The user does not invoke the Makefile generator as such. Instead, it is invoked as a result from another code generator (i.e. the C++ code generator) which as a by product generates a Makefile model file. Whenever the Application Builder receives a build result file of the type "make-model" after a code generator has completed, it launches the Makefile generator using the make model u2 file as input. Also, if the build artifact that was used to invoke the "main" code generator is stereotyped by "Makefile generator Settings" the settings of that stereotype is given to the Makefile code generator. I.e. the "Makefile Generator Settings" stereotype is used to change the default behavior of the Makefile code generator (it does not used as a prerequisite for the code generator to be run).

## Code Generator Stereotypes

As a code generator the Makefile code generator is defined in the profile package "MakefileGen". This defines the two stereotypes "Makefile generator" and "Makefile Generator Settings" (see diagram below). The stereotype "Makefile Generator Settings" contains all settings specific to the

Makefile code generator while the stereotype "Makefile generator" contains only what it inherits from "build" and "Makefile Generator Settings", it does not define any attributes by itself.

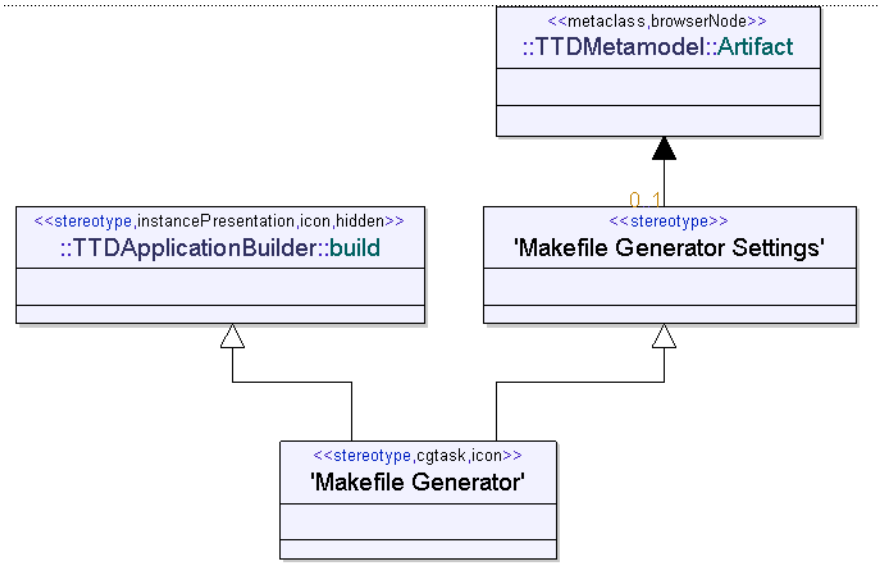



*Figure 194: "Makefile Generator" and "Makefile Generator Settings"*

When the Makefile code generator is used explicitly the build stereotype "Makefile Generator" is used. When the Makefile code generator is used implicitly, the stereotype "Makefile Generator Settings" may be added to the Build Artifact to change the default behavior. The default Generator Parameters used when a Makefile is generated depend on the host system (Win32, Solaris or Linux).

The settings defined in "Makefile Generator Settings" are:

**Dialect**

This determines the dialect of the generated Makefile; "gmake" for a GNU C/C++ (and classical) compatible makefile or "nmake" for a Win32 nmake compatible Makefile. The default depends on the Target Kind.

**Target**

This determines if the target is to build an executable or library (static or dynamic). The default is to generate a Makefile suitable to build an executable.

**Target Kind**

This controls the overall configuration of the Makefile generation. The default depends on the host system, for example "Win32 - cl" on Win32 systems, "Linux - g++"on Linux systems or "Solaris - CC"on Solaris.

### User Code

This field is used to override the settings implied by Target Kind. Any additional make variables can be defined here.

**Example 314: Build artifact for C++ code generation**

The Build Artifact "`HelloWorldArt`" manifests the active class "Hello". The artifact is used for C++ code generation. The artifact is also stereotyped with the "Makefile Generator Settings" stereotype that is used to override the defaults used when a Makefile is generated. The stereotype settings look like:

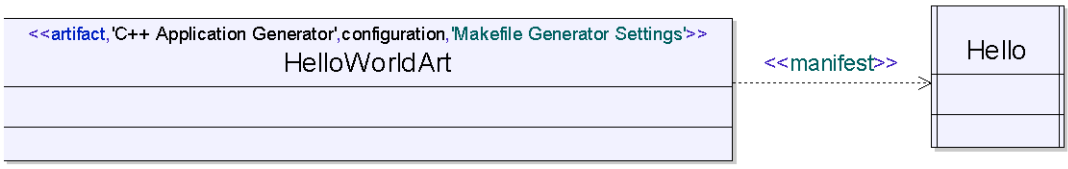



*Figure 195: The build artifact "`HelloWorldArt`" manifests the active class "Hello"*

This will produce a Makefile suited for Solaris and a GNU C/C++ tool chain.

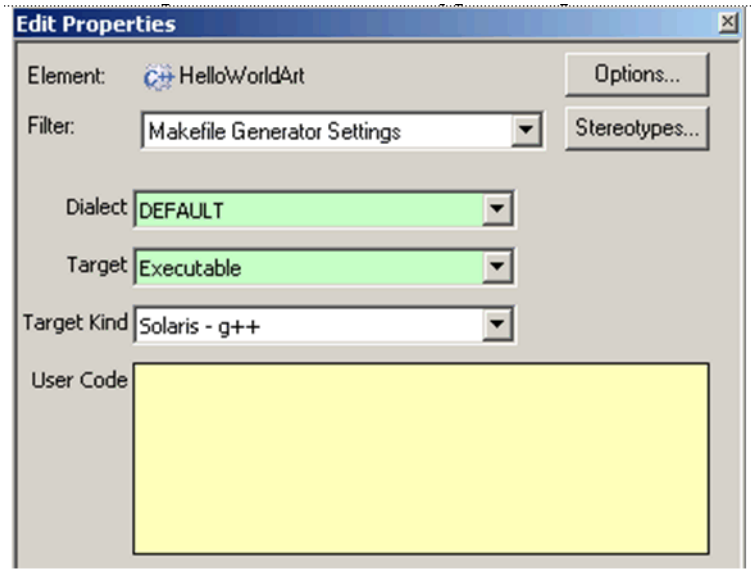



*Figure 196: Makefile Generator stereotype settings*

## File Stereotypes

The Makefile generator recognizes artifacts stereotyped by the file artifacts:

| Name | Defined in | Comment |
|---|---|---|
| PHONY | MakefileGen | This is used for “phony” targets such as “all” and “clean”. Note that this is not a stereotype that inherits the “file” stereotype, instead it just extends artifact. |
| objectFile | MakefileGen | This is used to represent object files. |
| executable | TTDFileModel | This is used to represent executable files. |
| library | TTDFileModel | This is used to represent libraries in general. Artifacts with this stereotype are treated as if stereotyped with “staticLibrary”. |
| staticLibrary | TTDFileModel | This is used to represent static libraries (archives under non Win32 systems and lib files under Win32). |
| dynamicLibrary | TTDFileModel | This is used to represent dynamic libraries (shared object files on ELF systems, DLLs under Win32). |
| cppImplementationFile | TTDFileModel | This is used to represent C++ implementation files. |
| cppHeaderFile | TTDFileModel | This is used to represent C++ header files. |

## Make Model

The input to the Makefile code generator is a model (or a part of a model) that contains file artifacts stereotyped with File Stereotypes. Dependencies between the file artifacts are interpreted as make dependencies. Any stereotypes on the dependencies are ignored. Transformations

A number of model transformations are performed before the final Makefile is written. In this section the notion "cpp file" is used for an artifact stereotyped by "`cppImplementationFile`", "`obj file`" is used for an artifact stereotyped by "`objectFile`" etc. The term lib file is used for an artifact stereotyped by "`library`", "`staticLibrary`" or "`dynamicLibrary`". The notion A "depends on" B is used for a dependency where A is the client and B is the supplier.

**Collect include models**

**Automatic object files**

For each cpp file that does not have an `obj` file that depends on it, a new `obj` file is created that depends on the cpp file.

**Routing to object file**

If an executable file, a library file or PHONY depends on a cpp file, that dependency is changed to depend on the `obj` file that depends on the cpp file. The transformation "Automatic object files" ensures that there is such an `obj` file.

**Default target**

If there are `obj` files without dependencies from executable or library files a new default target (`exe` or `lib`, depending on the "Target" attribute) is created and dependencies are set up to the `obj` files. If the new default target is an executable, dependencies are created from it to each library.

**Default "all" target**

If there is no PHONY target named "all", one is created that depends on all executable and library files.

**Default "clean" target**

If there is no PHONY target named "clean", one is created that depends on all executable, library and object files.

### Name transformation

Executable, library and object file names are transformed if the corresponding generator parameter is set. The names are transformed according to the generator parameters. The flag parameters are; “`transformExeName`”, “`transformObjName`”, “`transformStaticLibName`” and “`transformDynamicLibName`”. The names are mangled using the parameters; “`executableName`”, “`objectFileName`”, “`staticLibName`” and “`dynamicLibName`”. Before mangling, the artifact name is stripped of any path and known suffix, then the new name is mangled after witch the path is restored. Suffixes:

```
".exe", ".obj", ".lib", ".dll", ".u2", ".ttp", ".ttw",
".o", ".a", ".so", ".asm", ".m", ".mak", ".cxx", ".hh",
".hpp", ".C", ".rc", ".res", ".cc", ".hh".
```

### Path transformation

Paths are transformed according to the pathDialect parameter. The path separator is modified ('/' versus '\'), any drive specified is removed under UNIX and the path is quoted if it contains irregular characters. The transformation is applied to `.obj`, `.exe`, `.lib`, `.cpp` and `hpp` files and to the (internal) list of include paths.

**Example 315: Make model with targets and dependencies**

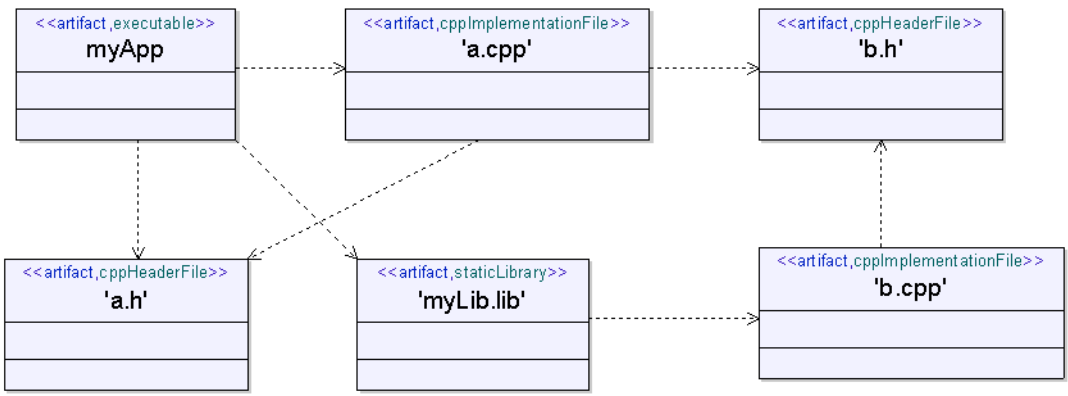



*Figure 197: Make model targets and dependencies*

```
all : myApp.exe myLib.lib
myApp.exe : a.obj myLib.lib
myLib.lib : b.obj
a.obj : a.cpp
b.obj : b.cpp
clean :
```

Note the intermediate object files.

---

## Generator Parameters

The generator parameters control how the Makefile is generated. The parameters are organized in sets; the default set, one set for each target kind and the user set. When the generator fetches a parameter it searches the user set, the set corresponding to the target kind and finally the default set.

Data written to the "User Code" attribute of the stereotypes "Makefile Generator Settings" or "Makefile generator" are put into the "user" set.

### Built-in parameters

In the following the term "default" indicates that the particular value of the parameter is in the "default" parameter set for each Target Kind.

Some of the values of the built-in parameters describe how a string should be modified. This is used using a set of codes, all composed of a '%' followed by a single character. The codes are:

| Code | Description |
|---|---|
| `%%` | The code is replaced by a single `%` |
| `%x` | The code is replaced by the name |
| `%q` | The code is replaced by the name but quoted if it contains "unusual" characters. |
| `%b` | The code is replaced by the base name |
| `%t` | The code is replaced by the target name |
| `%d` | The code is replaced by the first dependency |
| `%D` | The code is replaced by a space separated list of all dependencies. |

The last three can only be used in command parameters.

E.g. if "`dynamicLibName`" is "`lib%x.so`" and the name of the dynamic lib artifact is "`MyLib`", it will be changed to "`libMyLib.so`".

Also in command parameters the sequence '\n' (a back slash followed by a lower-case 'n') is expanded to a new-line followed by a horizontal tab. This allows commands to be multi line.

The base name replacement code `%b` is primarily intended for replacing library file names with the "base name" (as in "`-l%b`"). The replacements are: "`libX.a`" to "`X`", "`libX.so`" to "`X`" and "`X.lib`" to "`X`".

**applicationBaseName**

This is the base name of the Default target (executable). Default is "application".

**compileCppCommand**

This defines the command(s) used to compile a cpp file. The default is:

```
"$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) -o %t %d"
```

Codes: `"%%" "%t" "%d" "%D"`, expands '\n'.

**cppDefineFormat**

This is the format of command line defines to the compiler. The default is `"-D%x"`.

Codes: `"%%" "%x" "%q" "%b"`

**cppIncludeFormat**

This is the format of command line include paths to the compiler. The default is `"-I%x"`.

Codes: `"%%" "%x" "%q" "%b"`

**dynamicLibBaseName**

This is the base name of the Default target (dynamic library). Default is `"rary"` (because the base name is later mangled using `"lib%x.so"`).

Codes: `"%%" "%x" "%q" "%b"`

**dynamicLibName**

This defines how dynamic library names are mangled. The default is `"lib%x.so"`.

Codes: `"%%" "%x" "%q" "%b"`

**executableName**

This defines how executable names are mangled. The default is `"%x"`.

Codes: `"%%" "%x" "%q" "%b"`

**linkDynamicLibCommand**

This defines the command(s) that is used to produce a dynamic library. The default is:

```
"$(CXX) $(LDSOFLAGS) -o %t %D $(LDLIBS)".
```

Codes: `"%%" "%t" "%d" "%D"`, expands '`\n`'.

**linkExeCommand**

This defines the command(s) that produce an executable. Default is:

```
"$(CXX) $(LDFLAGS) -o %t %D $(LDLIBS)"
```

Codes: `"%%" "%t" "%d" "%D"`, expands '`\n`'.

**linkFormat**

This is used when an executable or dynamic library has a dependency to a library. The name of the library is added to the list of dependencies using this format. Default is: "`-l%b`"

Codes: `"%%" "%x" "%q" "%b"`

**linkStaticLibCommand**

This defines the command(s) that is used to produce a static library. The default is:

```
"$(AR) $(ARFLAGS) %t %D".
```

Codes:`"%%" "%t" "%d" "%D"`, expands '`\n`'.

**makeCommentFormat**

This defines the format of comments in the Makefile.Default is `"# %x"`.

Codes: `"%%" "%x" "%q" "%b"`

**makeDependencyFormat**

This defines the format of make dependencies. Default is `"%x"`.

Codes: `"%%" "%x" "%q" "%b"`

**makeDereferenceFormat**

This defines the format of make variable de-references used in the Makefile. The default is `"$(%x)"`.

Codes: `"%%" "%x" "%q" "%b"`

**makeDialect**

This is the dialect of the Makefile. There is no default.

**makeTargetFormat**

This defines the format for make targets. Default is `"%x"`.

Codes: `"%%" "%x" "%q" "%b"`

**makeTargetType**

This determines the type of Default target. Valid values are “Executable” (this is the default), “Static Library” and “Dynamic Library”. This parameter is set using the “Target” attribute of the stereotypes “Makefile Generator Settings” and “Makefile generator”.

**makefileName**

This is the name of the generated Makefile. The default is “`Makefile`”.

**objectFileName**

This defines how object file names are mangled. The default is `"%x.o"`.

Codes: `"%%" "%x" "%q" "%b"`

**staticLibBaseName**

This is the base name of the Default target (static library). Default is `"rary"` (because the base name is later mangled using `"lib%x.a"`).

**staticLibName**

This defines how static library names are mangled. The default is `"lib%x.a"`.

Codes: `"%%" "%x" "%q" "%b"`

**transformDynamicLibName**

This is a Boolean flag that is used to enable transformation of dynamic library file names. The default is "`true`".

**transformExeName**

This is a Boolean flag that is used to enable transformation of executable file names. The default is "`true`".

**transformObjName**

This is a Boolean flag that is used to enable transformation of object file names. The default is "`true`".

**transformStaticLibName**

This is a Boolean flag that is used to enable transformation of static library file names. The default is "`true`".

**pathDialect**

This defines the Path transformation. Possible values are:

- "`dos`"
  - Any '/' is changed to a '\'.
  - Any preceding drive specification is not removed.
  - If the path contains any other token than "`0-9A-Za-z/._-:\`" it is embraced in double quotes.
- "`unix`"
  - Any '\' is changed to a '/'.
  - Any preceding drive specification is removed.
  - If the path contains any other token than "`0-9A-Za-z/._-`" it is embraced in double quotes.
- "`cygwin`"
  - Any '\' is changed to a '/'.
  - Any preceding drive specification is not removed.
  - If the path contains any other token than "`0-9A-Za-z/._-:`" it is embraced in double quotes.

Any other value disables the transformations.

### Make parameters

All parameters that are not built in parameters are transferred to the Makefile as they are. The Makefile code generator does not in any way interpret or place any meaning into the value, presence or absence of any of these parameters.

The set of make parameters is not fixed but depend on the target kind (i.e. some parameters are only present for a specific target kind). Note also that the variables used are those normally used on the corresponding platform and tool chain. This means that the semantics can differ (i.e. “CPP” is used for the C pre-processor on all systems except Win32 where it is used for the compiler).

### Target Kind

The target kind is used to define a set of parameters. It combines the notion of target operating system and tool set (compiler linker etc.) to use. The following target kinds are used.

| Target kind | Description |
|---|---|
| `Win32 - cl` | Microsoft 32 bit Windows operating system using the tool chain supplied with the Microsoft Visual Development Environment. |
| `Cygwin - g++` | This targets the Cygwin environment running on top of Microsoft 32 bit Windows using the gnu tool chain supplied with Cygwin. |
| `Linux - g++` | This uses the gnu tool chain under Linux. |
| `Solaris - CC` | This uses the SUNWsPro tool chain under Solaris. |
| `Solaris - g++` | This uses the gnu tool chain under Solaris. |

In the following tables the names of the built-in parameters all start with a '$'. If a user needs to override the value of a built-in parameter it should be added to the “User Code” attribute with the initial '$'. Any parameter with a name that does not begin with a '$' is treated as a make parameter and totally ignored by the Makefile generator (except from being echoed to the Makefile without interpretation).

### Default values for makefile generator parameters

| Name | Value | Description |
|---|---|---|
| RM | rm -f | This is the delete file command used by the "clean" target. |
| DEFINES | $(TAUDEFINES -D_REENTRANT | |
| INCLUDES | $(TAUINCLUDES) | |
| CPPFLAGS | $(INCLUDES) $(DEFINES) | |
| CXXFLAGS | -O2 -fpic | |
| TAUDEFINES | | The value of this is set by the Makefile code generator. It should not be assigned in any other way. |
| TAUINCLUDES | | The value of this is set by the Makefile code generator. It should not be assigned in any other way. |

### Win32 - cl values for makefile generator parameters

| Name | Value | Description |
|---|---|---|
| $makeDialect | nmake | nmake is used on Win32 |
| $staticLibBaseName | library | |
| $dynamicLibBaseName | library | |
| $makefileName | makefile.mak | |
| $executableName | %x.exe | |
| $staticLibName | %x.lib | |
| $dynamicLibName | %x.dll | |
| $objectFileName | %x.obj | |

| Name | Value | Description |
|---|---|---|
| $cppIncludeFormat | /I %q | |
| $cppDefineFormat | /D %q | |
| makeTargetFormat | %q | |
| makeDependencyFormat | %d | |
| $linkExeCommand | link $(LINKEXEFLAGS) /out:%t %D | |
| $linkStaticLibCommand | lib $(LINKLIBFLAGS) /out:%t %D | |
| $linkDynamicLibCommand | link $(LINKDLLFLAGS) /out:%t %D | |
| $compileCppCommand | $(CPP) $(CPPFLAGS) /Fo%t /c %d | |
| $pathDialect | dos | |
| RM | del /f | |
| CPP | cl | |
| DEFINES | $(TAUDEFINES) /D \"WIN32\" /D \"NDEBUG\" $(SUBSYSDEF) | |
| CPPFLAGS | /nologo /MT /W3 /GR /GX /O2 $(INCLUDES) $(DEFINES) | |
| CXXFLAGS | $(CPPFLAGS) | |
| CFLAGS | $(CPPFLAGS) | |
| SUBSYSFLAG | /subsystem:console | |
| SUBSYSDEF | /D "_CONSOLE" | |
| LINKLIBS | kernel32.lib user32.lib gdi32.lib winspool.lib comdlg32.lib advapi32.lib shell32.lib ole32.lib oleaut32.lib uuid.lib odbc32.lib odbccp32.lib ws2_32.lib | |

| Name | Value | Description |
|---|---|---|
| LINKEXEFLAGS | $(LINKLIBS) /nologo $(SUBSYSFLAG) | |
| LINKLIBFLAGS | /nologo $(SUBSYSFLAG) | |
| LINKDLLFLAGS | $(LINKLIBS) /nologo $(SUBSYSFLAG) /DLL | |

### Cygwin - g++ values for makefile generator parameters

| Name | Value | Description |
|---|---|---|
| $makeDialect | gmake | |
| pathDialect | cygwin | |
| AR | ar | |
| ARFLAGS | crv | |
| CXX | g++ | |
| CXXFLAGS | -O2 | |
| LDFLAGS | | The value is empty |
| LDSOFLAGS | -shared $(LDFLAGS) | |
| LDLIBS | -lrt -lpthread -lm | |

### Linux - g++ values for makefile generator parameters

| Name | Value | Description |
|---|---|---|
| $makeDialect | gmake | |
| AR | ar | |
| ARFLAGS | crv | |
| CXX | g++ | |

| Name | Value | Description |
|---|---|---|
| LDFLAGS | | The value is empty |
| LDSOFLAGS | -shared $(LDFLAGS) | |
| LDLIBS | -lrt -lpthread -lm | |

### Solaris - CC values for makefile generator parameters

| Name | Value | Description |
|---|---|---|
| $makeDialect | gmake | |
| AR | ar | |
| ARFLAGS | -xar -o | |
| CXX | CC | |
| CXXFLAGS | -O2 -pic -instances=static | |
| LDFLAGS | | The value is empty |
| LDSOFLAGS | -G $(LDFLAGS) | |
| LDLIBS | -lsocket -lnsl -lrt -lpthread | |

### Solaris - g++ values for makefile generator parameters

| Name | Value | Description |
|---|---|---|
| $makeDialect | gmake | |
| AR | ar | |
| ARFLAGS | crv | |
| CXX | g++ | |

| Name | Value | Description |
|---|---|---|
| LDFLAGS | | The value is empty |
| LDSOFLAGS | -shared $(LDFLAGS) | |
| LDLIBS | -lsocket -lnsl -lrt -lpthread | |

# Makefile

The name of the generated Makefile is given by the "$makefileName" parameter. The generated Makefile is composed of the following sections (in order):

**Preamble**

This is a set of comments that describes the origin of the Makefile.

**Dialect specification**

This is a comment that is used later by the Makefile execution to figure out what dialect the Makefile uses. It is a comment with the string "MakeDialect=" immediately followed by "nmake" or "gmake".

**Make parameters**

All make parameters are written to the Makefile. No particular order is used.

**The all target**

The dependencies to this target are emitted in the order; executables, dynamic library targets, static library targets, library targets and other targets. This target does not have a command.

**Executable targets**

The target uses the parameter "$linkExeCommand".

**Dynamic library targets**

The target uses the parameter "$linkDynamicLibCommand".

**Library targets**

The target uses the parameter "$linkStaticLibCommand".

**Static library targets**

The target uses the parameter "`$linkStaticLibCommand`".

**Object file targets**

The target uses the parameter "`$compileCppCommand`".

**The clean target**

**Postamble**

This is a comment that says "`END`".

# Code Generation in Tau

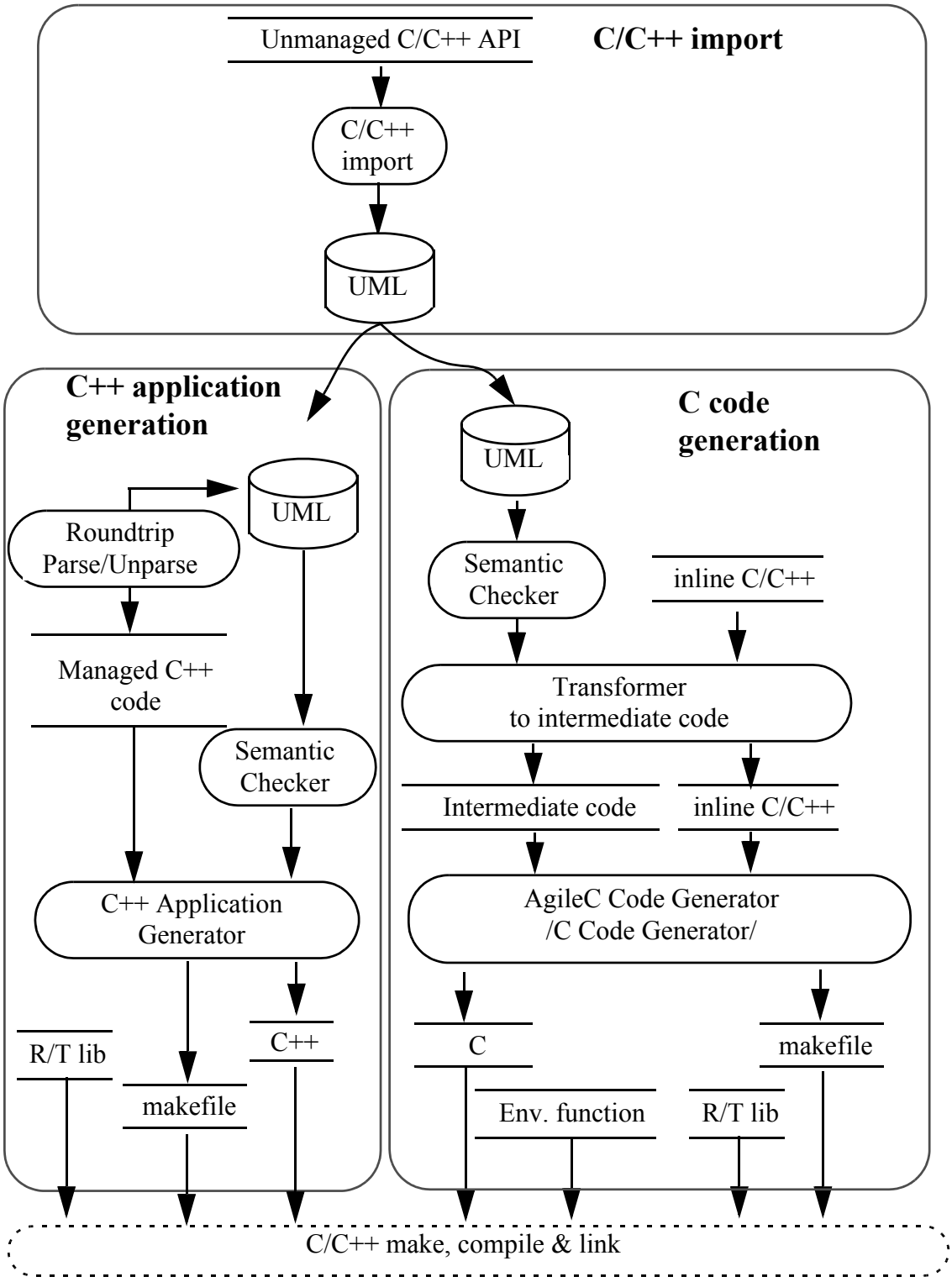



*Figure 198: Code generation in Tau.*

# C/C++ import

The C/C++ import is mainly intended to provides access to external C/C++ APIs from UML applications developed with Tau, hence allow to include legacy code in a project where UML is used for the development of new applications. Importing C/C++ code is achieved by parsing a set of C/C++ header files and then translating the C/C++ definitions to their corresponding UML definitions, according to a fixed set of C/C++ to UML translation rules.

## Target

The UML elements that are created during import are added to UML packages designated by the user. Such packages could already contain UML models under development, or be packages that act as a containers dedicated to hold the result from the import.

After an import, next step would be to refer to the imported UML elements from within the application modeled in UML. Imported UML could also be copied and pasted into the packages that contain the model that defines the actual application under development.

## Preprocessing

The C/C++ import is designed to take advantage of the preprocessor and the paradigm of Conditional Compilation that is frequently used in the C/C++ world. This allows a flexible importing that is aligned with the ideas that govern the design of the imported code and how it should be compiled.

## C and C++ support

When importing, you can specify whether the source language is plain ISO C, or if C++ can be expected in the input. The required subset of the C and C++ syntax and semantic rules is checked to ensure that the sources hold enough information to be able to correctly translate them to UML. Various popular dialects of the C and C++ languages are supported to broaden the application range. (See "C/C++ dialect" on page 983 in Chapter 28, *Stereotypes for Code Generation*.)

### Translation rules

The translation to UML obeys a set of translation rules that have been designed to capture a rich part of the C/C++ languages and also result into UML models that you can easily refer to from within the UML model under development.

### Trace back to source

Each of the UML elements that are generated during import contains attributes with information about what source header file, and where in the file the source definition is found.

**See also**

"C/C++ Import" on page 545.

## C code generation

From your UML models, the C Code Generator Reference and the AgileC Code Generator Reference create complete real-time application code. The generated code is designed to be easily augmented with external C/C++ code and is compiled and linked with the run-time library of your choice, which will give the application the desired properties. Conditional compilation through the means of the preprocessor is used extensively to achieve a high degree of flexibility and scalability.

### UML models

UML models are managed as 'native' source. Full support for UML is provided, including extensive semantic checks and complete generation of application code in C.

The elements in the model that will become part of the generated C code, include the following:

- **Active classes** become executable code. The behavior of the application is specified by **state machine** diagrams, which serve as detailed design or implementation specifications.
- Other classes (that is **non-active classes**) become data types, including operators on these data types. When used or instantiated, such classes may therefore, in addition to data, also result in code.

- **Composite structure diagrams** specify the interface to the environment, and also the internal interfaces between the state machines. From the composite structure diagrams the code that implements the connection and signalling between state machines is generated, and also the interface to the environment.
- **Build artifacts** are used to control build properties. A Build Artifact can exist in the model view and be presented in a class diagram.
- The project containing the model can also contain any number of **Configurations** that each may include any number of build artifacts.
- **Thread artifacts** are used to control the deployment of the application into threads. A Thread Artifact represents a set of instances, usually based on an active class. A build artifact can be explicitly modeled into one or more thread artifacts, which will be used to determine how the instances in a model are to be built. A thread artifact has a relation to one or more build artifacts.

### Settings

The settings for the code generation are defined in build artifacts that are incorporated in the model and that hold the information that should be separated from the abstract model.

Build artifacts specify for instance the code generator settings, runtime libraries and external code. Furthermore, a build artifact also holds the information required to successfully compile and link the target application.

## Inline C/C++

Tau supports “inline” C or C++ code. Such code can be added virtually anywhere in the UML model where it makes sense. Inline code is forwarded almost verbatim to the C Code Generator and the AgileC Code Generator and will hence become embedded in the final generated application code. There are some rules for the inline C code.

- Inline C code is written within double brackets `[[ ]]`
- Comments in inline C (using `/* */`) are not supported
- The ‘#’ sign is used to escape characters in UML code, for example a preprocessor operator is written double hash signs (`##`), UML entities can be accessed using “`#(<UML name>)`”

**Example 316: Inline C, implementing a UML operation**

The UML operation below (`SetClock`) sends an integer value to a C function (`SetTime`) used for initializing a user defined clock function.

```
void SetClock( Integer TimeNow) {
[[
##ifdef USER_CLOCK_FUNC
SetTime((xint32)#(TimeNow));
##endif
]]
}
```

The operation will be empty if the compiler switch `USER_CLOCK_FUNC` is not set at compile time. The UML Integer argument is type casted to a 32-bit integer type (`xint32`). (This type is defined by the C Code Generator.) Below is the corresponding C code for the function prototype of `SetTime`.

```
void SetTime (xint32 newTime);
```

## Escaping # in inline C code

In inline C code, '#' (hash) signs are used with a special meaning. Therefore '#' must be escaped when writing C code to be used inline in Tau. The '#' sign is also used to escape characters in UML code.

```
#(<UML name>)
##0, ##1, ..., ##9
####0, ####1, ..., ####9
```

The above constructs are replaced by the C name for some appropriate unit defined in the source. In all other cases a '#' will represent nothing but a '#'.

However there are some situations when the translation rules above make it impossible to include the intended C code. In format strings it is, for example, possible to have a # followed by a digit. To do this the '#' has to be escaped.

A sequence of three '#' in the intermediate code, that is ###, will be copied as one '#'. To achieve this it is necessary to escape each '#' in the UML code, thus a sequence of six '#' signs will result in one '#' in the generated code.

**Example 317: Escaping of ######**

```
######1
```

becomes:

#1

# Checking model before build

## UML semantic checking

Upon a build command, an exhaustive semantic check is done to verify that the input UML model is correct and complete. When a check is done in the context of a build of a C application, the semantic checker also performs additional checks that catch restrictions imposed by the C Code Generator / AgileC Code Generator and the run-time systems used by the applications they generate.

Given that the model is verified to be semantically correct, code generation is initiated.

## Intermediate code

As a prerequisite to application code generation to C, the UML tool set starts by transforming the UML model to an intermediate format. This intermediate code can be regarded as a refinement of the source UML model, enriched with action semantics and run-time dynamics that are needed for the C Code Generator and AgileC Code Generator to be able to generate application code with real-time properties.

The transformation is transparent to the user, and takes advantage of some powerful features. This transformation is controlled by one of the Add-Ins named **IMGen**.

- When creating this representation, the transformation tool will identify any inline C code that is added by the user, and forward it to the C Code Generator and the AgileC Code Generator verbatim.
- In the source UML model, you have the option to tag parts of the model that you do not want to generate code for in a build.
- The transformation tool also identifies any UML elements that the user has tagged as “external” in the model, meaning that the definition of the code is available elsewhere (likely in C/C++ files that should eventually be added to the compile and link scheme to build a complete application).

IMGen uses for performance reasons GUID binding and not name binding. As a result of this, some name resolution errors are not caught until late in the build process and certain TIL errors might occur. If that happens, run a "Check All" and correct your model.

### C code generation

The C Code Generator and the AgileC Code Generator first check that the intermediate representation code is correct and complete. After the intermediate code has passed these checks, the C Code Generator and the AgileC Code Generator generate complete application C code from it.

## UML to C

### UML to C translation, run-time semantics and optimization

When translating UML to C, the C Code Generator and the AgileC Code Generator use a number of translation rules that ensure that the code obeys run-time semantics, scheduling and signalling issues in a safe way, even in exceptional situations. The code is generated in such a way that it is given good execution performance, while keeping the code size reasonably low. Dynamic memory is managed in a wise way in order to avoid memory fragmentation.

### Navigation from model view to source code

The Code Generators enables model to code navigation. If there exist header or implementation files as a result of code generation, navigating from a UML element to the corresponding lines in code is done by right-clicking on it in the model view and choosing **Go to source**. This command is also available for navigation from the generated code and back to the model.

File references to the generated code will appear in a package in the Model view named "`Result of C/C++ Generation`".

This feature is for the C code generator stereotypes (Model Verifier, C Code Generator) associated with the option "**Generate reference package**". This option is activated by default.

### C and C++ support

The C code is generated as plain ISO C, or as C with support for C++ compilers, depending on the user's preferences. This option is provided in order to cope with issues related to the C/C++ compiler that is used, and also if any external C++ code should become part of the final application.

### Runtime libraries

When building C applications, you have the option to specify what target system the code should be compiled for. The workflow one would follow in most cases would to start by compiling the code so that it can be executed and debugged on your host computer. After the application has been verified to behave as expected, then you would probably proceed by compiling it for your target system, run further tests and finalize the integration with the environment.

The combinations are numerous and a number of pre-defined C and AgileC Runtime Libraries are provided, to be used for the frequent application areas.

### Environment support

To become a self-contained application, the application code needs to be interfaced with the environment it is designed to interact with. Such interaction is performed through the means of signals sent to and from the environment. By providing a few functions only, to ensure that the environment is properly initialized and closed, and to ensure that the interaction between the UML model and the "real world" is handled correctly, you have developed a self-contained application.

When the option "Generate environment Template Functions" is set, the C Code Generator and the AgileC Code Generator produce the following:

- Skeletons for the Environment Functions for C Applications, to help you write the code that handles the receiving and sending of signals to/from the environment.
- A System interface header file, in which the interface between the system and its environment is defined, and that also contains the definition of some data types that map the representation of UML concepts to C. This header file is required when including the functions that integrate the application code with its environment and is commonly referred to as system interface header file (the `.ifc` file).

- A make template file that specifies how to compile and link the application. This make template file can be used by the user to include external C or C++ code to the compile and link scheme.

**make**

As a final step in the build process, the Application Builder will invoke "make" for you, using the generated makefile and possibly the make template file that is specified to use in the C Code Generator and the AgileC Code Generator settings.

The generated C code, the run-time library of your choice and external code that is defined in the make template file are now compiled with the desired properties which were defined when the makefile was created.

The actual "make" program that is used depends on the operating system and compiler environment that you are using and have specified.

**See also**

Chapter 27, Building Applications Reference

Chapter 33, C and AgileC Runtime Libraries

Chapter 39, AgileC Code Generator Reference

## Execution modes

An application generated by the C Code Generator or the AgileC Code Generator can be executed in two modes, bare and threaded.

**Bare**

The application is executed as one unit, in which all the parts are scheduled by the internal scheduler in a quasi-parallel way. This means that a transition is always executed to its completion and can not be interrupted by another transition. When the transition is finished the event is selected and the corresponding transition is executed. This execution mode is called bare.

The bare execution mode does not require much from the execution platform. If the application uses timers or needs the current time for some other reason, the platform must include a way to read a clock. If the application requires dynamic memory, then this has to be implemented, either by using the

memory package included in the AgileC Code Generator that manages memory inside a static array, or by providing two functions similar to the C standard functions `malloc` and `free`.

### Threaded

In the other mode, called threaded, the execution relies on an underlying operating system that supports some kind of parallel execution (threads, tasks, processes or whatever the parallel executing entity is called). In this mode the parts are divided into a number of groups, where each group executes in one thread. Inside a thread the scheduling is the same as in the case above, quasi-parallel execution, but between threads the underlying operating system gives possibilities for context switches between the threads. In this way higher priority operations can be preempt a lower priority operation and be executed "at once" (given that it is supported by the underlying operating system).

Details about how the integration with the operating system is implemented for the threaded mode can be found in "Integration with Compiler and Operating System" on page 1285.

### Considerations when selecting execution mode

The decision if the mode without an underlying operating system is adequate or not, should be made by comparing the longest transition, which will be the maximum latency in the application, with the maximum time allowed from the most critical situation is detected, until the code handling this must start to execute. If the anticipated execution time for the longest transition is shorter than the required task switch latency, then you will have the best overall performance running without an extra operating system (Bare), or at least with all relevant active classes mapped to the same thread. If the latency is not sufficiently small, then you have to rely on preemptive context switching properties (between threads) within an external operating system (Threaded).

# Conditional Compilation

Sometimes one single UML model has to describe several different versions of an application. The different versions can for example be intended for different platforms or may be one debug/test version and one release version.

In Tau different versions of an application are normally created by having different artifacts describing the different versions. Each artifact should contain the necessary information to be able to define the details of the application to be produced by running a code generator based on the artifact.

This is also the case for the conditional compilation support. The artifacts are extended to also describe the conditions that define the conditional compilation.

## UML Level Support

On a UML level the conditional compilation is described by a predefined stereotype «`conditional`» that can be applied to the following entities:

- definitions (e.g. class, signal, operation and interface)
- composite statements (i.e. blocks surrounded by { } in UML textual syntax)
- transitions

In addition a possibility to define conditional decision statements to allow conditional compilation in graphical state machine diagrams is also introduced.

The «`conditional`» stereotype has one attribute: `expr:Boolean`. This attribute defines a condition that determines if the entity is part of an application generation or not. Prior to code generation these expressions are evaluated. Entities whose conditional expression evaluates to false will be removed from the model and hence are not subject to code generation.

**Note**

*Some code generators support to keep also conditional entities whose condition is false, and instead generate these entities within preprocessor blocks (thus postponing removal until preprocessing the generated code). See Impact on Code Generation for more information.*

The expressions used in «conditional» elements (and in conditional decision answers) may only reference Boolean constants, global or package level attributes stereotyped by «conditionalConstant», tests on the predefined attribute activeCG and may only include the literals 'true' and 'false' and the following operations:

- not
- and
- or

The tests on activeCG are used to check what code generator is used in the current build. The activeCG attribute is a Charstring attribute and its value is initialized automatically. Conditional expressions can use this attribute to test if it is equal to “CGEN” or “CPPAPPGEN” to include or exclude parts of the model in C and C++ code generation. The operators “==” and “!=” are supported for the tests. An example:

```
«conditional(.expr = activeCG=="CGEN".)>> {
    i = 10;
}
«conditional(.expr = activeCG!="CGEN".)>> {
    i = 20;
}
```

**Note**

*The* <<informal>> *stereotype can also be used for conditional compilation as it is equivalent to a conditional expression which evaluates to false.*

### Conditional Definitions

From a syntactic point of view the stereotype is shown using graphical syntax for definitions. The value of the attribute is edited using the Properties Editor like in Figure 199 on page 884.

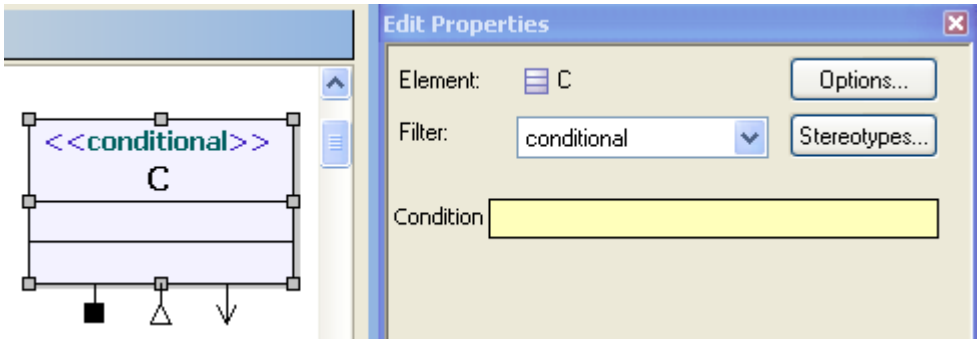



*Figure 199: Edit a conditional expression*

It is also possible to use UML textual syntax to apply the stereotype:

```
<<conditional(.expr = "A".)>> class myclass {

}
```

*Figure 200: The «conditional» stereotype applied in UML textual syntax*

### Conditional action statements

For composite statements the syntax would be only textual syntax as shown in the following statement sequence:

```
i = 10;
«conditional(. expr = A .)>> {
  j = 11;
}
```

### Conditional transitions

The «conditional» stereotype is not shown in graphics for transitions.

### Conditional Decisions

Decision symbols can be marked as conditional. The mechanism is to use to mark a decision as conditional using a predefined identifier **CONDITIONAL**. Give the conditional expression in one of the decision answers and include the conditional actions in this branch, see Figure 201 on page 885.

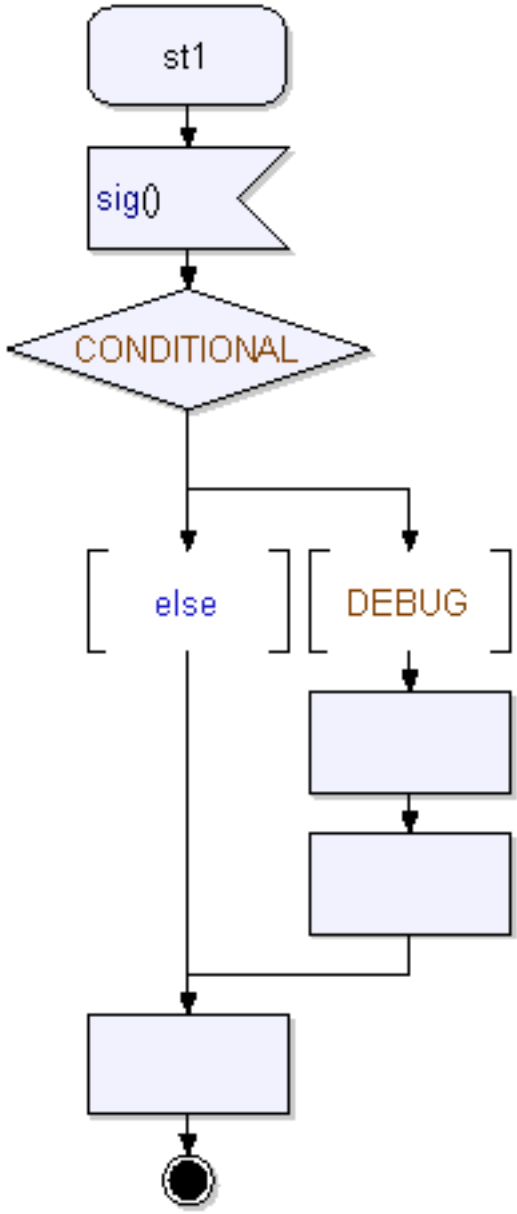



*Figure 201: Decision symbols marked as conditional*

### Restrictions on conditional decisions

There are a number of static constraints on conditional decisions:

- The decision expression must be a `Boolean` expression following the same rules as other conditional expressions
- There must only be two branches in the decision

- The branch not containing the conditional actions must be marked as 'else'
- The conditional branch must join the flow of the 'else' branch before the first symbol on the 'else' branch following the decision answer symbol.

### Artifacts and Conditional Compilation

When building an application in Tau the build settings are described by a Build Artifact.

This kind of artifacts are also used to set up the values controlling the conditional compilation.

Conditional expressions may only reference constants or «conditionalConstant» attributes defined in a global scope. The artifacts provide a means to define build specific values for the global scope attributes. This is provided by means of a special operation in the artifact that must be called conditionalInit(). The body of this operation may only contain a set of assignment statements that give values to «conditionalConstant» attributes.

**Note**

*This is the only way to provide values to the* «conditionalConstant» *attributes. It is for example not possible to give them default values so all* «conditionalConstant» *attributes that are used in the model must be given values in the* conditionalInit *operations.*

**Example 318: Assign values to global attributes**

Assume three global attributes:

```
<<conditionalConstant>> Boolean Debug;
<<conditionalConstant>> Boolean Test;
<<conditionalConstant>> Boolean Instrument;
```

Initialize these in the conditionalInit() operation:

```
void conditionalInit(){
  Debug = true;
  Test = false;
  Instrument = Debug or Test;
}
```

### Impact on Code Generation

The details of the support for the «`conditional`» stereotype / conditional decisions in different code generators is determined by the specific code generator but two different kinds of support are provided:'

- A pre-processing strategy
- A target language mapping strategy

If the pre-processing strategy is used the «`conditional`» stereotypes / conditional decisions are handled during a specific pass of the code generator. Essentially the expressions are interpreted and the model element for which the conditional expression is false is removed from the model. The code that is generated will thus not contain any representation of these model elements.

If a target language mapping strategy is used then the «`conditional`» stereotypes / conditional decisions are mapped to conditional statements in the target language, like e.g. `#ifdef` statements in C/C++.

The pre-processing strategy is mainly useful in a forward generation scenario, but can be applied independent of target language. For example it can be applied to Java code generation even though Java does not support a conditional compilation concept.

The target language mapping scenario works both for a roundtrip scenario and a forward generation scenario. But it is limited to code generation for languages that support a conditional compilation concept.

The C code generators always use the pre-processing strategy whereas the C++ code generator uses the target language mapping strategy if roundtrip is enabled and the pre-processing strategy if roundtrip is not enabled. The Java code generator uses the pre-processing strategy.

## Restrictions

Conditional compilation is handled at code generation time only. It is not possible to define e.g. two different classes with the same name in the same scope using different conditional expressions.

Conditional compilation is not supported by other code generators than the C, C++ and Java code generators.

# Composite Structures

The purpose of this section is to give an introduction into how to use UML composite structure diagrams with a special focus on the support provided in the C Advanced and Agile C code generators. In particular it will show how composite structure diagrams relate to part-whole relationships and how to use these concepts together with dynamic creation of instances and asynchronous communication.

## Parts vs. whole relations

In software engineering there is a common need to express that one entity is a part of another entity. Typically this is used to show how different components are composed of other components in a hierarchical fashion. This relation is in UML expressed as a composite association. The graphical syntax is as shown in Figure 202 on page 888.

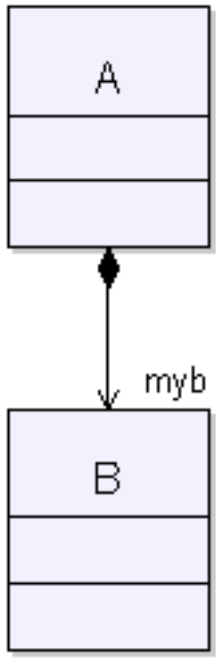



*Figure 202: Class A with B part called 'myb'.*

The same example can also be shown using one class symbol where the part is shown in the attributes compartment (Figure 203 on page 889).

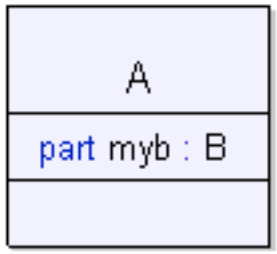



*Figure 203: Part shown in the attributes compartment'.*

Yet another way to show the same example is to use textual syntax in e.g. a text symbol in a diagram inside the A class (Figure 204 on page 889).

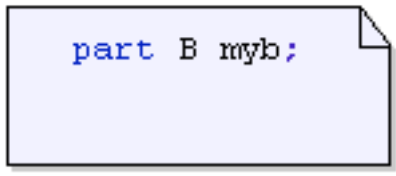



*Figure 204: Textual syntax for part.*

In the example so far the part-whole relation has been a one-to-one relation. However, the relationship can of course be more complex. The most common cases are when a container contains either a fixed number of parts or when the parts are created dynamically with or without an upper bound on the number of allowed instances. In any case this is shown by the Multiplicity of the part end of the composite association (Figure 205 on page 890).

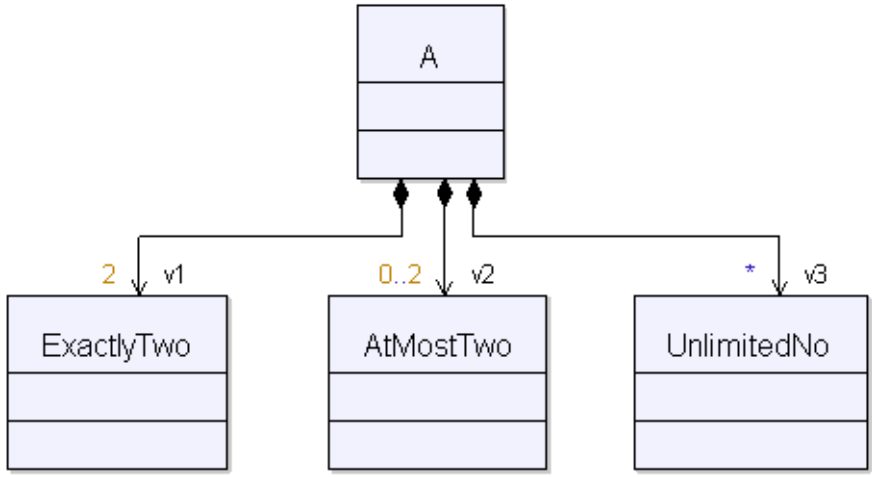



*Figure 205: Container with parts of various multiplicity.*

The same example using attribute definitions in a class symbol is shown in Figure 206 on page 890.

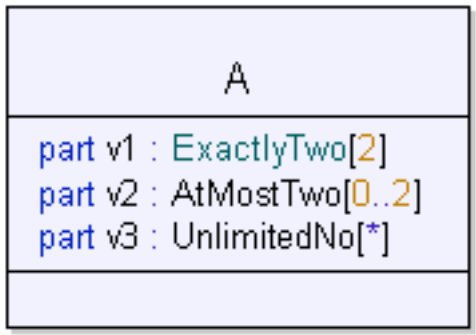



*Figure 206: Class with parts.*

The same example using definitions in a text symbol is shown in Figure 207 on page 891.

```
part ExactlyTwo [2] v1;
part AtMostTwo [0..2] v2;
part UnlimitedNo [*] v3;
```

*Figure 207: Part definitions in a text symbol.*

The intention in this example is that whenever an instance of class A is created, immediately there must be two instances of `ExactlyTwo` created since there is a fixed multiplicity of 2. No instances of the `AtMostTwo` or `UnlimitedNo` classes should be created. Instance of these classes are expected to be created later in the lifetime of the A instance.

However, whenever the A instance is terminated, all of the contained instance should also terminate.

The examples so far shows only use of passive classes. The semantics is the same for active classes that contain a separate thread of control and the classes could just as well have been active as shown in Figure 208 on page 891.

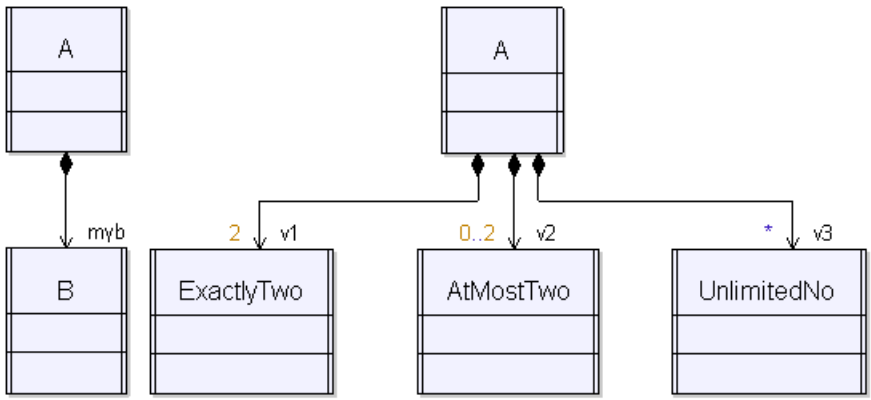



*Figure 208: Active classes with parts.*

## Composite structure vs. part-whole relationships

Composite Structure diagrams are used to show the internal structure of UML classes. The ‘internal structure’ that is shown consists of the attributes of the class and the connectors that link them together. All kinds of attributes

can be visualized in a composite structure diagram; composite attributes, non-composite attributes, attributes typed by classes and attributes typed by datatypes.

An example of this general case is Figure 209 on page 892 (note the dashed symbol indicating that '`aReference`' is not a composite part):

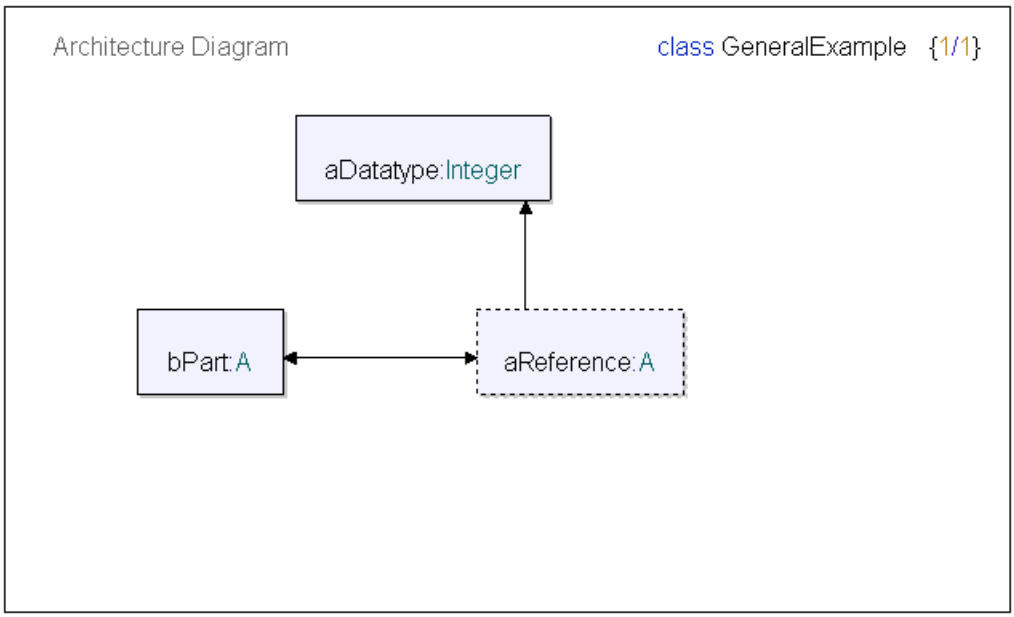



*Figure 209: Composite structure diagrams with the internal structure of a UML class.*

From a design point-of-view the composite structure diagrams provide two benefits:

- They give a visual view of the internal structure, making it easier to understand the structure of a complex application.
- They provide the 'glue' to put together components through an addressing mechanism that allows the parts to be designed independently and then graphically linked together when composing the containing entity.

The visualization should be self-evident from the example, but the addressing mechanism may need some more elaboration. It is based on the possibility to communicate via the ports of a class instead of based on the identity of the receiver. So, e.g. if a class A has a port p, then it is possible to write the following statement somewhere in the behavior code of class A:

```
output s() via p;
```

The semantic of this statement is that a signal ‘s’ is sent via the port ‘p’, forwarded via the connector that in a given context is connected to ‘p’ and finally received by whoever is connected to the other end of the connector. The identity of the receiver is completely determined by the composite structure diagram where ‘A’ is instantiated.

To make the graphical glue to work the classes should however preferably be designed according to certain rules:

- They should have ports with required and/or realized interfaces
- They should use the ports to establish communication with the outside world
- They should be designed to be self-contained and have their own thread of control. They should thus be active classes.

The implication is that a component typically consists of the following definitions:

- An active class with ports
- A set of interfaces

A simple example is shown in Figure 210 on page 894.

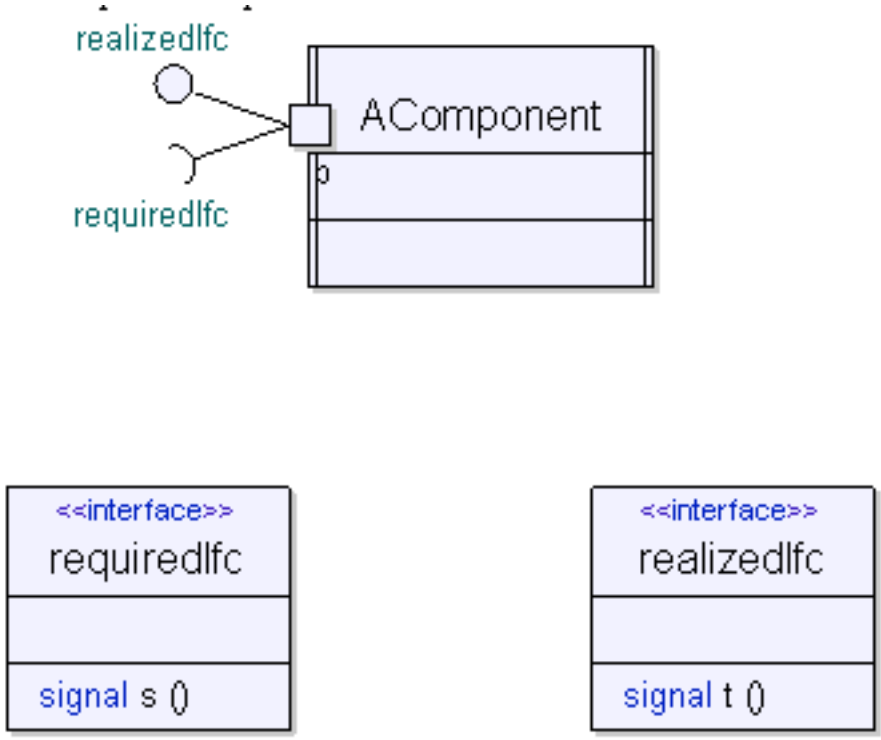



*Figure 210: Active class with ports.*

This component can now be used in a composite structure diagram (Figure 211 on page 894) as a part of a larger system.

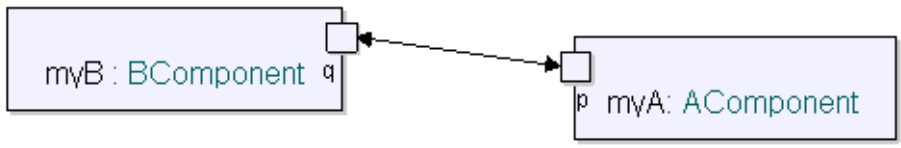



*Figure 211: Components in a composite structure diagram.*

In most cases composite structure diagrams focus on the composite attributes that define the parts of the class. This is also the reason why the diagram (slightly misleading) officially in UML also is called ‘Composite Structure diagram’.

Composite Structure diagrams can be used both for active classes and for passive classes, but since they often show loosely coupled architectural entities they are in practise mainly used to describe the top-level active parts that define the architecture of an application. Parts that communicate using asynchronous communication and that may be distributed over a network.

In the C code generators the support is restricted to this common case: Composite Structure diagrams can only be used to define the internal structure of active classes and is only allowed to contain composite parts typed by other active classes.

## Dynamically created active instances vs. part-whole relationships

In UML active instances can be created dynamically using the ‘new’ statement. Depending on the structure of the application a newly created instance can either be created as a composite part of a containing entity or it can be created outside all composition hierarchies and composite structures.

The choice depends on how the instance is intended to be used. If the corresponding class (or classes communicating with it) is coded as a component with ports etc. and relies on the ports for communication then it needs to be inserted in an internal structure to work properly.

If, on the other hand, the class and its surrounding classes do not rely on ports for communication but instead always specify the receiving instance when sending signals or calling operations then it does not need to be inserted in the internal structure of a containing entity.

In the AgileC Code Generator the composition relation is very strictly enforced. A newly created instance must either directly be inserted in a composition relation or it will be considered to be outside all composition relations. The syntax used depends on the Multiplicity of the attribute but are shown in Example 319 on page 895.

**Example 319: Attribute multiplicity**

```
A[0..1] Aref;
// Aref is a reference to one A instance

A[*] Arefs;
// Arefs is a list of reference to A instances

part A[0..1] Apart;
// Apart is a composite part for one A instance
part A[*] Aparts;
```

```
// Aparts is a composite part for a list of A instances
```

**Example 320: Instance creation**

```
Aref = new A();
// This created an A instance outside all
// composition hierarchies/internal structures
// and keeps a reference to it in Aref´

Arefs.append( new A() );
// This created an A instance outside all
// composition hierarchies/internal structures
// and added a reference to it in ‘Arefs´

Apart = new A();
// This created an A instance as a composite
// part of the containing entity
// as defined by the composite part ‘Apart´
‘
Aparts.append( new A() );
// This created an A instance as a composite
// part of the containing entity and added
// it to the composite part ‘Aparts´
```

A common situation is that you need to get a reference to a new instance that is added to a composite part. This is accomplished with the ‘offspring’ mechanism in UML as in Example 321 on page 896.

**Example 321: Reference from offspring**

```
part A[*] Aparts;
A Aref;
Aparts.append( new A() )
Aref = offspring;
// Aref will now reference the newly create instance
```

The offspring variable will always hold a reference to the latest active instance that is created.

It is recommended to not rely on that the offspring values will be unaffected by other create operations in application code. Assign offspring to an attribute or variable before creating any new instances!

Another aspect that should be noted is that the dynamic creation of instances that should be inserted into a composite structure always should be performed by the containing class, i.e. the class that owns the part attribute where the instances is inserted. A typical structure for a component is to have a behavior defined by a state machine and a set of parts that will contain the

dynamically created parts. The dynamic creation of the instances is done by the state machine code. Figure 212 on page 897 show a fairly typical structure. First the diagram containing a component definition that defines an '`AServer`' that provides on management port ('`mgm`') and a service port ('s').

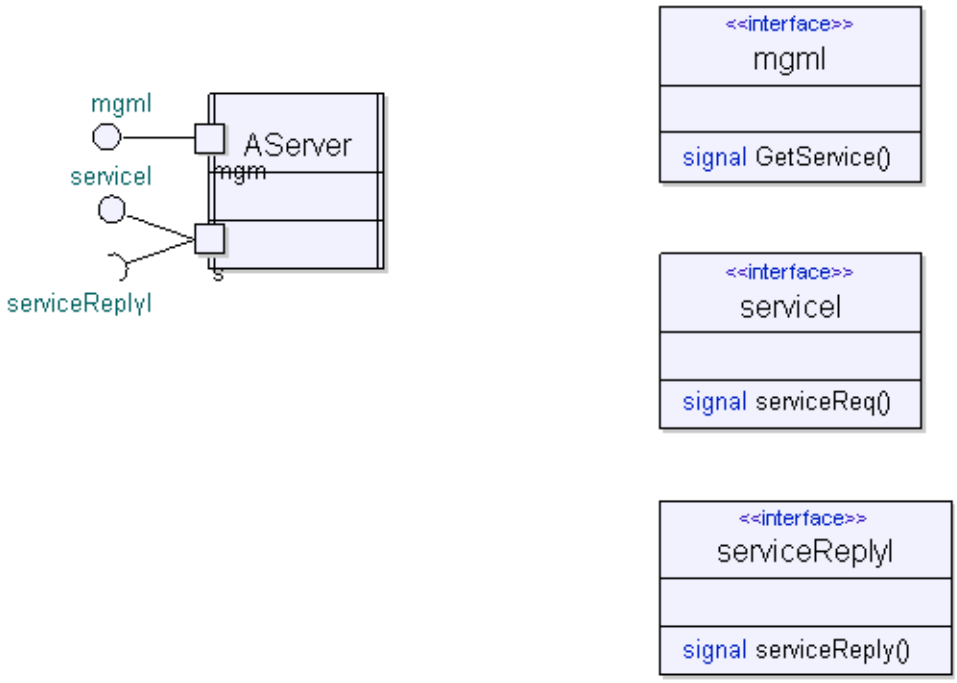



*Figure 212: Component definition with management port ('`mgm`') and service port.*

The internal composite structure of the `AServer` looks as Figure 213 on page 898.

**Note**

*The locally defined class* `LocalServer` *that is the class that implements the* `serviceI` *interface.*

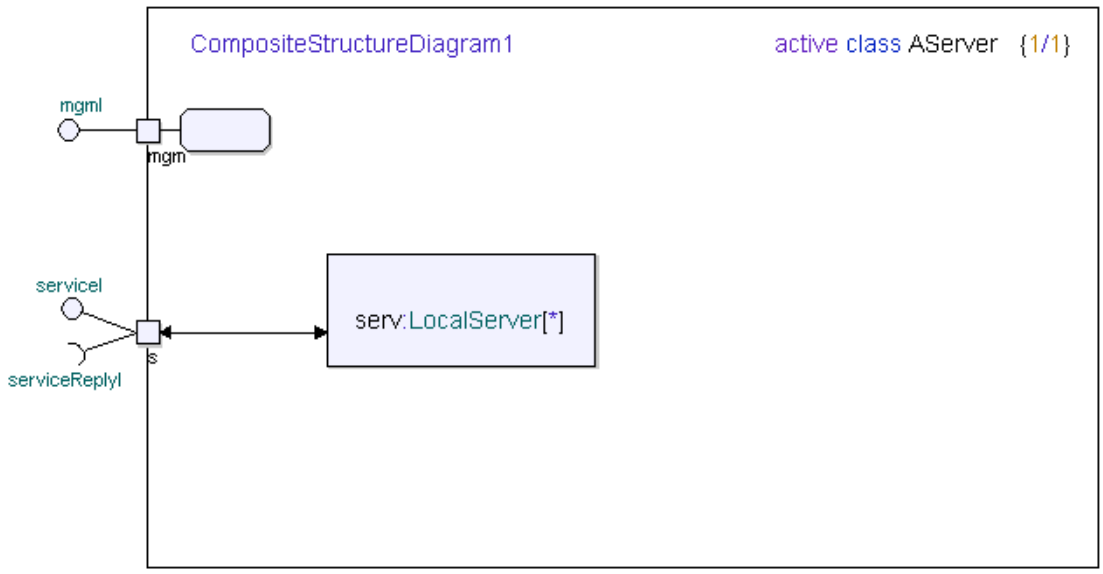



*Figure 213: Server class with one service.*

The main server class (‘`AServer`’) only supports one service. In a more realistic case there would of course be more services and `AServer` would have had more local parts. The state machine of the `AServer` class is shown in Figure 214 on page 898.

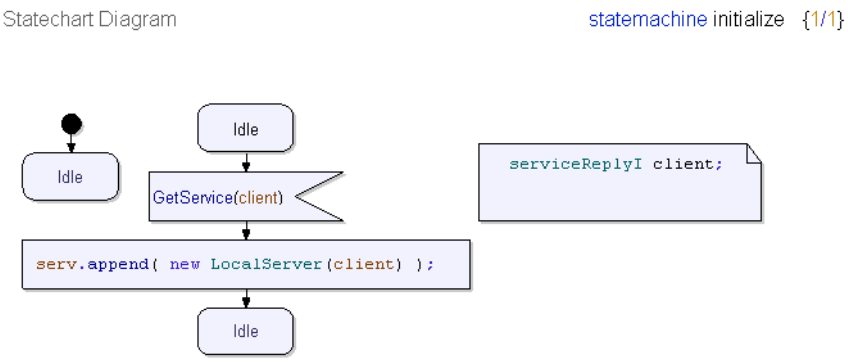



*Figure 214: State machine in the server class.*

Finally, the state machine of the very simple `LocalServer` class that implements the service is shown in Figure 215 on page 899.

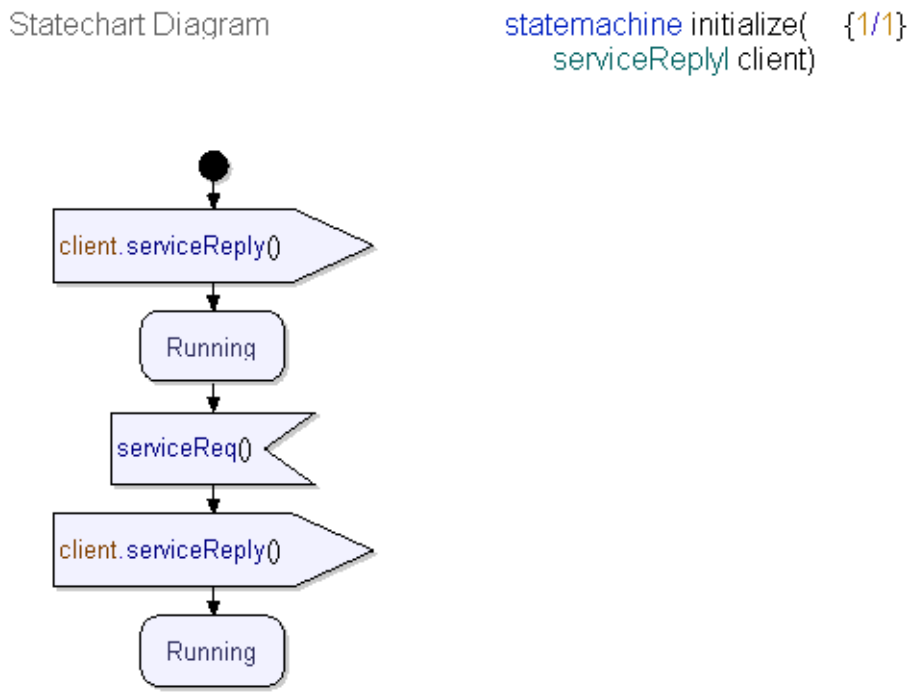



*Figure 215: State machine for the service.*

## Communicating with created instances

As described above there are two different ways to send signals or call operations on active classes:

- Using the composite structure to find the receiver
- Using explicit addressing of the receiver

There main practical benefit with using the composite structure in terms of ports, connectors etc. is that its simplifies the initialization of an application. There is no need to communicate references to the different instances that will form the executing application during an initialization step. This is particularly important for embedded applications with hard constraints on how dynamically allocated memory is used. The extreme (but fairly common) case is a situation where all instances are statically allocated already at initialization time and no dynamic memory is allocated at all during the execution. In this case addressing based on ports is very efficient.

To be able to use addressing using the composite structure combined with dynamic instantiation it is essential to remember to explicitly insert the dynamically created instances that should be accessible using the port/connector structure in the composite structure as described in the previous section. If this is not done the communication will not succeed.

In case dynamically created instances are not created as part of a composite structure they will not be accessible via the ports/connectors. Instead you need to use explicit addressing when sending signals or calling operations.

The static structure of a simple example that uses only explicit addressing is given by Figure 216 on page 900, by an extract from a class diagram.


*Figure 216:Static structure with explicit addressing.*

There are two active classes ‘A’ and ‘C’ in Figure 216 on page 900, where the ‘A’ class has a reference to the ‘C’ class. The ‘A’ class can now create an instance of the ‘C’ class and then send a signal to it, as you can see in Figure 217 on page 901, from the state machine of class ‘A’:

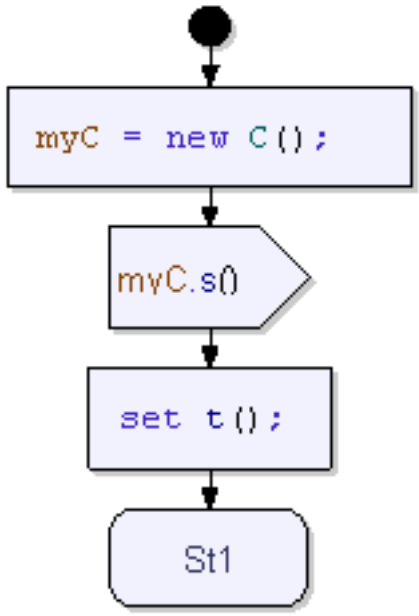



*Figure 217: State machine of class 'A'.*

In this example you do not create any ports on A and C nor do you create any composite structure. When sending the signal it is done directly based on the '`myC`' reference and not sent based on any ports.

Even though you do not define any ports on the classes in the example above it is possible to use interfaces and ports also when the ports are not used to specify the target in a signal sending or a call. By separating the interfaces from the implementation of the interfaces you get more extensible and reusable design. So, the classes above could as in Figure 218 on page 902, while keeping all behavior code.

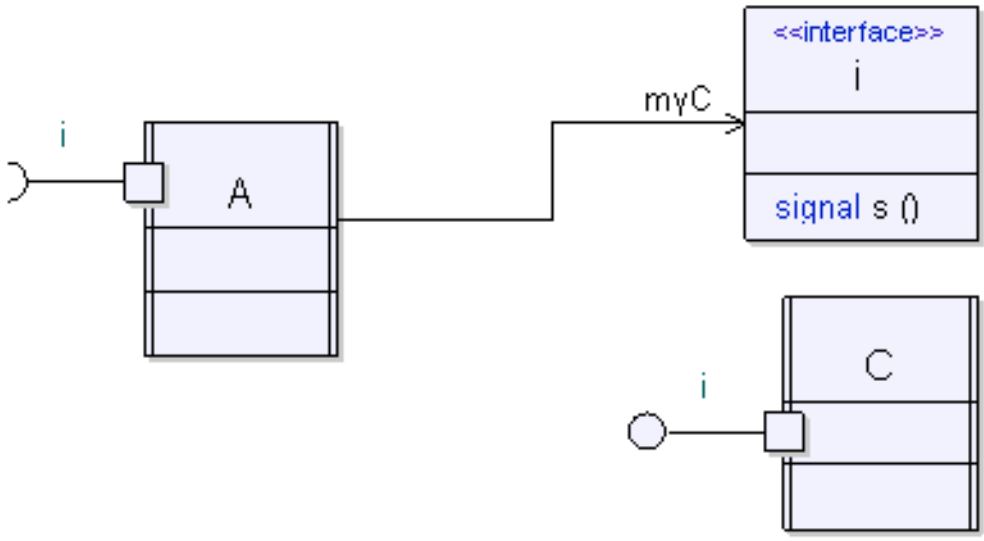



*Figure 218: Separated interfaces.*

The benefit is mainly that the ‘A’ and ‘C’ classes now can be reused as components in a composite structure. This would not have been possible in the previous example where you did not use interfaces and ports. So, if the classes are to be designed to encourage reuse it is better to use interfaces and ports.

## Iterating over parts

In many situations it is required to iterate over all instances contained in a composite attribute, i.e. a part. This can be done as in this example. Assume that you have an active class `Sys` with an internal architecture according to Figure 219 on page 903.

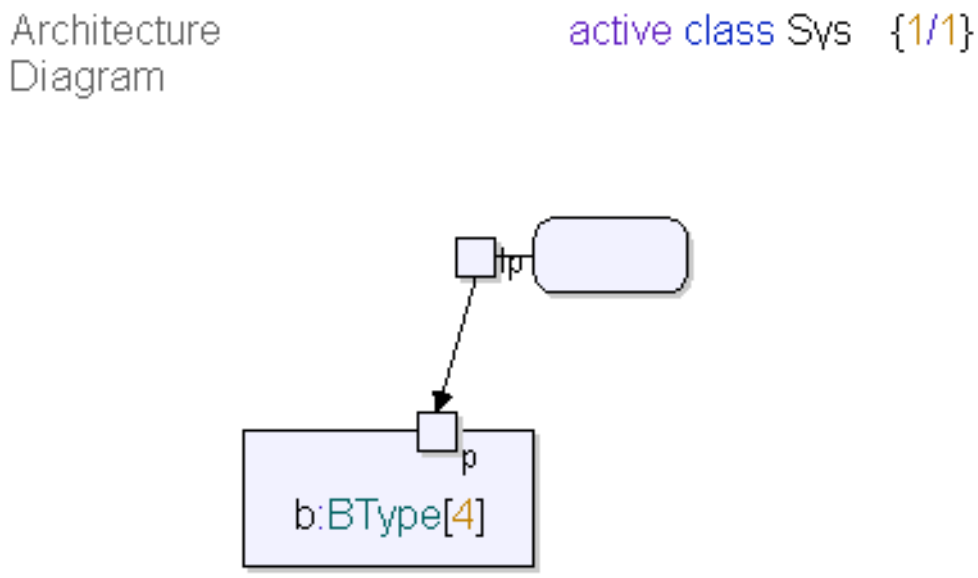



*Figure 219: Class* `Sys`.

Some minor comments on Figure 219 on page 903:

- The behavior port `lp`. This port makes the state machine of `Sys` visible in the composite structure diagram and you can connect ports on the internal parts to the state machine of `Sys` itself.
- The name and the information flows associated with the connector are omitted. This is a shorthand that can be used to avoid cluttering a diagram with too many details.

It is now possible to iterate over the instances in the b part from inside the `Sys` state machine. This can be done using the length operator and indexing as in Figure 220 on page 904.

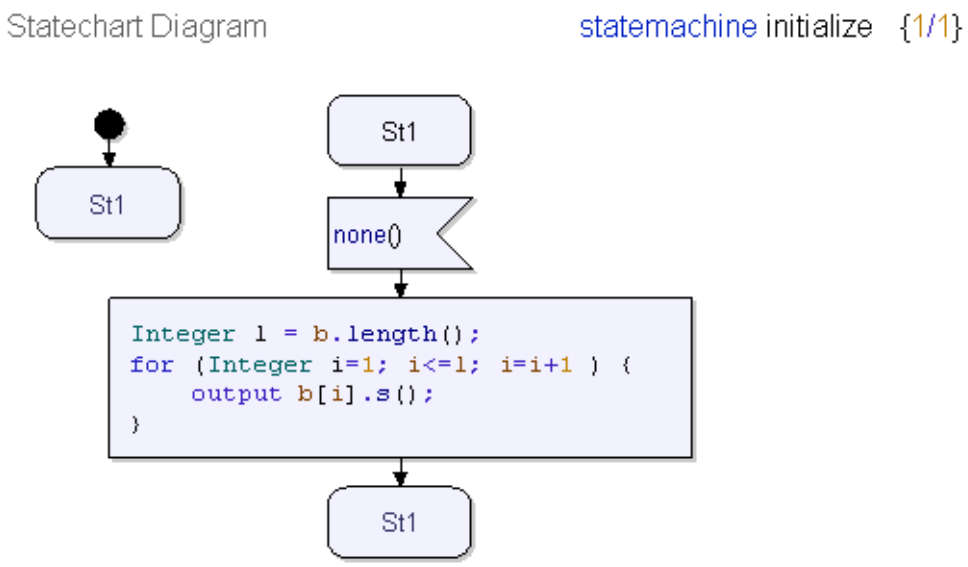



*Figure 220: Iteration over the instances.*

# Restrictions in C code generators

This section summarizes some of the most important constraints that the C code generators (C Code Generator, AgileC Code Generator) put on the UML model with respect to architecture and composition of active classes. Failure to heed these restrictions will likely lead to undesired behavior.

### Composite structure diagrams & composition

Composite structure diagrams are only allowed in active classes.

Only active classes may be used as types of the attributes shown in composite structure diagrams.

The attributes shown in composite structure diagrams must be composite parts and not references.

### Dynamic creation of composite classes

It is not possible to dynamically create instances of classes with internal architecture. More precisely this means that if an active class has a composite part typed by another active class it is not possible to create an instance of the

outer class. An example: Consider an active class A. Inside A is a composite structure diagram with a part symbol typed by B (another active class). It is now not possible to dynamically create A instances.

In practise this can be handled using a pattern with a ‘manager’ class that dynamically creates all necessary instances, see section Dynamically created active instances vs. part-whole relationships, where the ‘`AServer`’ acted as a manager of the contained ‘`LocalServer`’ instances.

### Instance termination

The only way to terminate an instance of an active class is for the instance to execute a stop action. It can not be terminated from other instances. In practise this means that active classes that should be possible to stop from the 'outside' should include a transition with a signal (e.g. called 'terminate') that performs necessary clean up actions and then performs a stop.

### Instance creation

Instances that should be inserted in an architecture, i.e. that should be added to a composite attribute, must be created by the owner of the attribute. It is thus not allowed to reference an attribute defined in a containing scope from within a nested class when creating a new instance. To get around this you can define an operator in the owner of the attribute which creates and appends the attribute and returns a reference to the newly created class. This operator can then be called in the nested class.

# Using the CPtr Type in Tau

The CPtr type is intended to provide features similar to a low-level pointer type in UML modeling. It is intended mainly for UML applications that interface with manually coded C or C++ components, but can be used in all UML application intended for C code generation. When importing external C/C++ declarations all pointers in the external code will be translated to CPtr instantiations in the UML model.

## Introduction

Consider for example the following fragment from a `.h` file:

```
int * ip;
typedef struct c {
  int i;
} * cp;
int op( cp p );
```

This will in UML be converted to:

```
CPtr<int> ip;
class c {
  int i;
}
syntype cp = CPtr<c>;
int op( cp p );
```

From a UML point of view the CPtr type can be seen either as a wrapper class that gives class properties to data types or as a low-level alternative to the usual class references in UML that provides some additional capabilities. The CPtr type is defined in the TTDCppPredefined UML package that is loaded by the CppTypes addin and provides following four operators:

```
SetValue
GetValue
GetAddress
'[]'
```

The details of how to use these operators in the UML model is described in the rest of this technical note.

## CPtr and data types

When using CPtr together with data types it acts like a wrapper class that provides the following features:

- It can be dynamically created using ‘new’ statements
- The contained value can be accessed using a ‘`GetValue`’ method
- The contained value can be changed using a ‘`SetValue`’ method
- It can be created based on an existing attribute or variable using the `GetAddress` operator
- It can be viewed as an array and indexed. This is only possible when the array is allocated in C/C++ code and is mainly intended for integrating with external C/C++ code.

**Example 322: CPtr allocation and access**

```
class c {
  Integer i;
  Integer j;
  CPtr<Integer> ip;
  CPtr<Integer> jp;
  void test() {

    /* Allocation of a CPtr, usage of SetValue/GetValue
*/
    ip = new CPtr<Integer>();
    ip.SetValue(10);
    i = ip.GetValue(); // ‘i’ is now 10

    /* Accessing an attribute using GetAddress */
    j = 20;
    jp = CPtr<Integer>::GetAddress(j);
    jp.SetValue(30); // ‘j’ is now also set to 30

    /* Viewing a CPtr as an array */
    CPtr<Integer> intArray;
    Integer len,index;
    [[
      #(intArray) = malloc(10*sizeof(#(Integer)));
      #(len) = 10;
    ]]
    for (index=0; index<len; index=index+1) {
      intArray[index] = 44;
    }
  }
}
```

## CPtr and classes

CPtr can be used together with classes to provide a low-level view on references to objects. However, since this does not give any major benefits compared to using regular references it is not recommended practice for pure UML modeling.

The main usage of CPtr and classes is thus when accessing external C/C++ code. All structs and classes in external C/C++ code will generate class definitions in the corresponding UML. Consider the following example of C code:

```
typedef struct c {
  int i;
} * cp;
int op( cp p );
```

The imported UML model will essentially contain the following declarations:

```
class c {
  public int i;
}
syntype cp = CPtr<c>;
int op( cp p );
```

This can be used from behavior code in the UML model. An example:

```
cp mycp;
int i;
mycp = new cp();
i = op( mycp );
```

## Recursive use of CPtr

`CPtr` can be applied recursively. This corresponds to pointers to pointers in C/C++ code and works both for `CPtr` types applied to data types and classes. For classes there is one level of indirection less, since a reference to a class is treated as semantically the same as a `CPtr` to the class. So, "`new c()`" will return a reference to "c" and "`new CPtr<c>()`" will return a pointer to a reference to "c".

**Example 323: Recursive CPtr**

```
class c {
   CPtr<Integer> ip;
   CPtr< CPtr<Integer> > ipp;
   CPtr<c> cp;
```

```
    CPtr< CPtr<c> > cpp;
    void test() {
      ip = new CPtr<Integer>();
      ipp = new CPtr< CPtr<Integer> >();
      ip.SetValue(11);
      ipp.SetValue(ip);
      cp = new c();
      cpp = new CPtr<c>();
      cpp.SetValue(cp);
    }
}
```

## Assignment compatibility of CPtr types

CPtr instantiations are assignment compatible and can be formulated like the following:

```
CPtr<Integer> i1;
CPtr<Integer> i2;
i1 = i2;
```

When calling operators there is an implicit assignment performed.

**Example 324: Import an external function**

```
int op(int * i);
```

**Example 325: External function in behavioral code of the UML model**

```
CPtr<Integer> ip;
ip = new CPtr<Integer>();
ip.SetValue(10);
op( ip );
```

## Recursive usage of SetValue/GetValue

`SetValue`/`GetValue` can be called recursively allowing for them to be used as they are in the following constructions:

```
class c {
  syntype ipType = CPtr<Integer>;
  syntype ippType = CPtr<ipType>;
  ipType ip;
  ippType ipp;
  void test() {
    ip = new ipType();
    ipp = new ippType();
```

```
    ipp.SetValue(ip);
    ipp.GetValue().SetValue(10);
  }
}
```

## Converting between CPtr and references

A common requirement is to be able to convert between a CPtr applied to a class and a usual reference to the class. This is fully supported in the tool and may be used in line with the following:

```
class c {
  public Integer i;
}
syntype CPtr_c = CPtr<c>;
class test {
  CPtr_c cp;
  c cr;
  void test() {
    cr = new c();
    cp = cr;
  }
}
```

The same can be done with a part as shown by the following example:

```
part c cv;
CPtr<c> cp;
void test() {
   cv.i=10;
   cp = cv;
}
```

# Threaded OS Integrations

## Overview

### RTOS

In the following section the term RTOS integration will be used frequently to refer to an adaptation of kernel source code for a specific RTOS (Real-time operating system) to allow an application generated by Tau from a UML model to run on that RTOS. This adaptation is partly prepared in the code delivered for a limited set of operating systems.

### Version differences

The difference between previous versions of threaded integrations and the ones presented here is not that large. The integration principles are almost exactly the same. The major difference is that previous versions of integrations were expressed using macros, while the new ones uses a functional interface. Another difference is that all previous versions were mixed into one header file, while in the new integration, each RTOS integration is implemented using one header (`.h`) and one body (`.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.

**Previous integrations:**

- SUN Solaris (This is a POSIX pthread integration and can be used on most UNIX-like operation system.)
- Win32

**New integrations:**

- POSIX pthreads (tested on SUN Solaris and Linux but can be used on most UNIX-like operation system.)
- Win32

### RTOS integration files

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.

### Compilation switches

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: `cc -DSCTAPPLCLENV -DTHREADED -I/some/suitable/path`

In the remaining part of this document the new threaded integrations are described in detail.

## 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 326: 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 327: 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 328: 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 329: 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: 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 contains a field of type `xThreadVars`, which should be defined in the RTOS integration.

Example: `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: 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. The 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. Together with the definition of the thread artifacts, 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:

```
#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. You need the possibility to perform a condition wait, with or without a time-out. You 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 330**

```
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 for SDL Suite. 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`.

**Example 331: 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);
```

# Application Examples

To fully assimilate and take advantage of the examples in this section, it is assumed that you have a knowledge of Tau, how to work with projects, models and diagrams, how to build applications and so forth.

The examples can all be started from within Tau if you go to the **File** menu and open the **New** dialog. Locate the **Template** tab and select the desired example to run.

## Examples with environment (EchoServer)

The examples are complete to allow "hands on" exercises, where you build the applications, run them and probably spend some time learning and understanding the supplied code to assimilate how to implement the integration between UML, C and C++ languages.

### Behavior

The application in the collection of examples models a small server, with a limited and simplistic functionality. The way the application behaves is the following:

1. For every reception of the signal `Say(Charstring, Duration)`, sent to the system from the environment, the active class Server dynamically creates an instance of the active class RequestHandler.
2. RequestHandler waits the number of time units specified by the Duration parameter.
3. The Charstring parameter is then concatenated to itself.
4. Lastly, the signal Echo(Charstring) is signal sending to the environment from the active class RequestHandler.

## Deployment and threading example

The "Deploy" example highlights the deployment and threading aspects of code generation. The AgileC Code Generator is used as the main flavor of code generator, since it is indented for situations where efficiency of the generated code is of importance.

### Behavior

The deployment example is a typical Client-Server system. The top active class “Top” contains two components “m” and “s” of the types “Master” and “Slave”, respectively (left part of Figure 221 on page 925). The master sends a request to the slave and waits for a reply. This is repeated 100 times (200 signals are sent). Before the first signal is sent the master prints the string “Starting” to `stdout`, and after the last signal is received it prints the string “Done”.

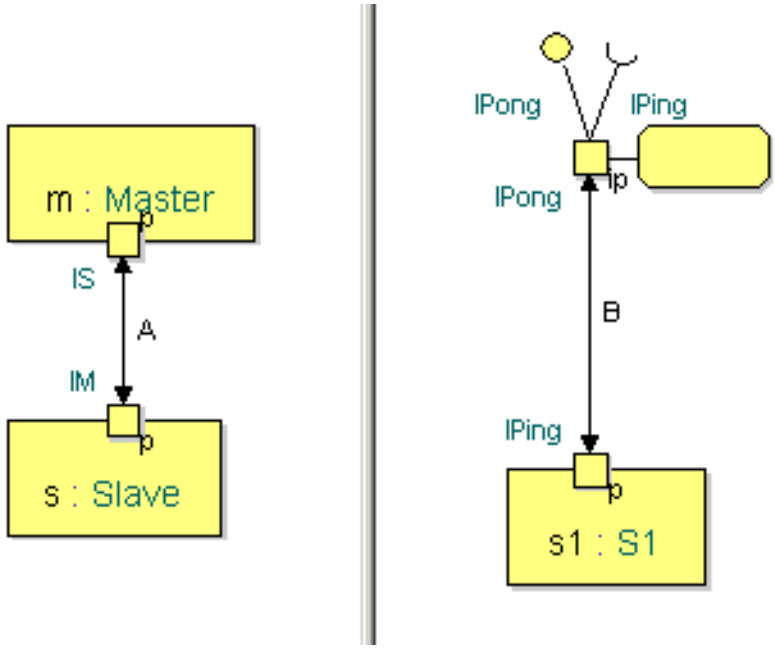



*Figure 221: Deployment in a composite structure diagrams*

The Slave class contains both a state machine and a component “s1” of the type “S1”. When the Slave class receives a request from the Master class, it initiates a signalling exchange sequence with its part “s1”. The signalling sequence is repeated 100000 times before the Slave class responds to the Master class. In total two million signals are sent in this example.

Figure 222 on page 926 demonstrates the situation in an alternative notation (squares represent active classes, circles represent state machines and lines signal flows).

**Note**

*The Master state machine and the Slave state machine communicate and, similarly how the Slave state machine communicates with the state machine in its part s1.*

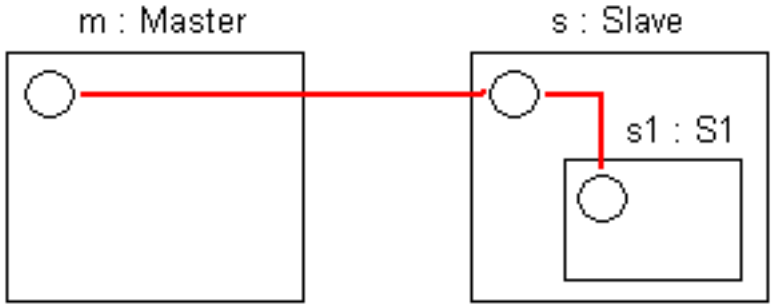



*Figure 222: Deployment alternative notation*

### Deployment into threads

The example could be deployed in many ways, depending on both the type of application to generate (Model Verifier or AgileC Code Generator application) and what kind of target system. Also the mapping of the state machines to different threads within the application can be varied.

In this example the following mappings to threads are used:

- One thread, containing all state machines
- Two threads – one thread holds the part m and one holds the part s.
- Three threads – one thread for each state machine

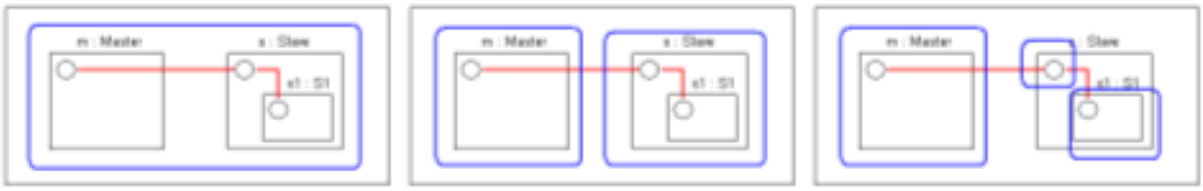


*Figure 223: Deployment mappings*

The class diagrams named “Deployment One Thread”, “Deployment Two Threads” and “Deployment Three Threads” describe these three cases.

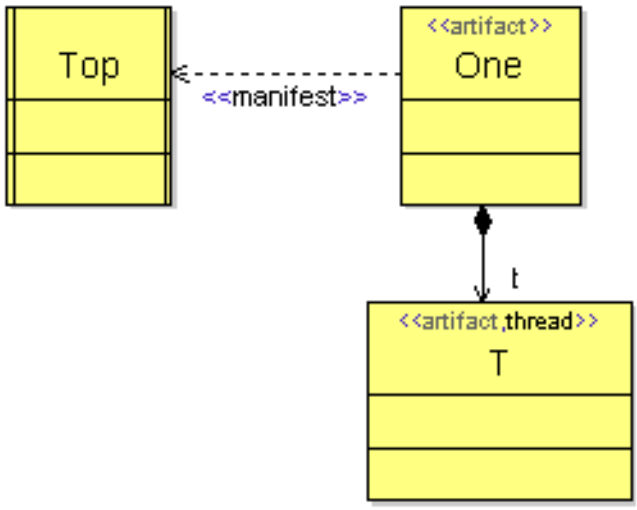



*Figure 224: Deployment into one Thread*

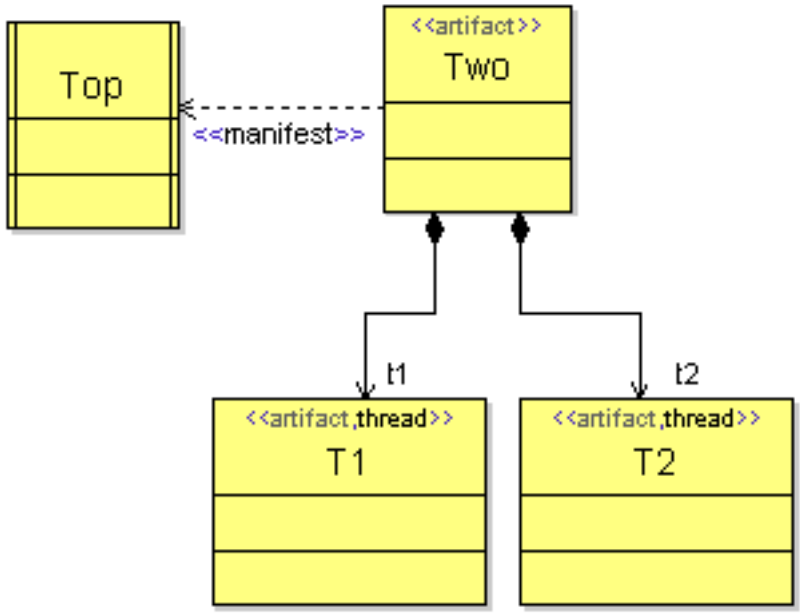



*Figure 225: Deployment into two Threads*

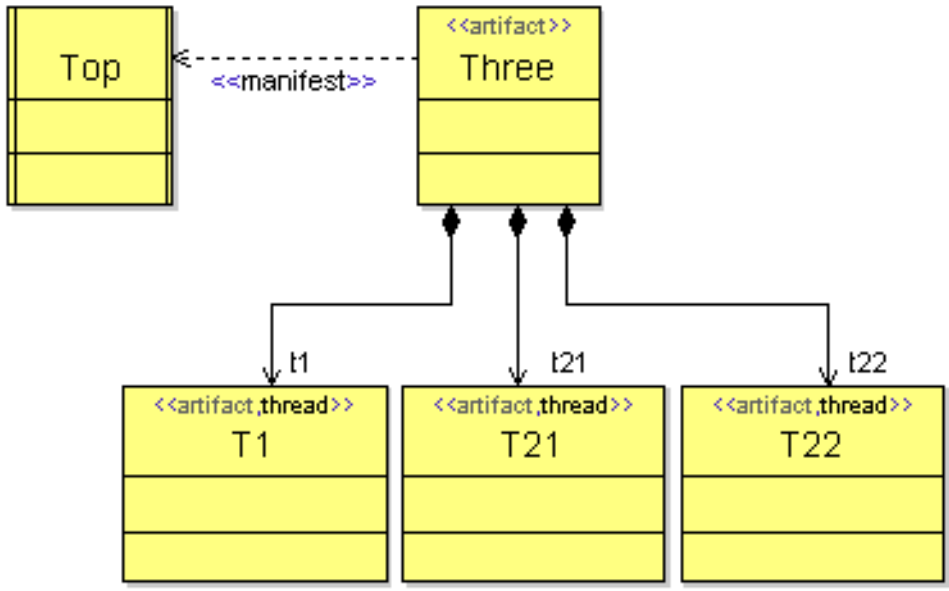



*Figure 226: Deployment into three Threads*

### Thread artifacts

Thread Artifacts are named compositions of the Build Artifacts. The Thread Artifact "T1" is used in both the two- and three-threaded cases.

Each of the thread artifacts holds a list of instance names of the state machines that the thread should contain. These names are fully qualified UML instance names relative to the "manifested" class. In this case there are the following possibilities:

- m – Denotes the part m of the Top class.
- s – Denotes the part s of the Top class.
- s.self – Denotes the state machine in the part s of the Top class.
- s.p1 – Denotes the part p1 in the part s of the Top class.

This list of instance names is edited using "Build Settings", for example Right-click on the "T" artifact and select "Build Settings". The following editor is displayed:

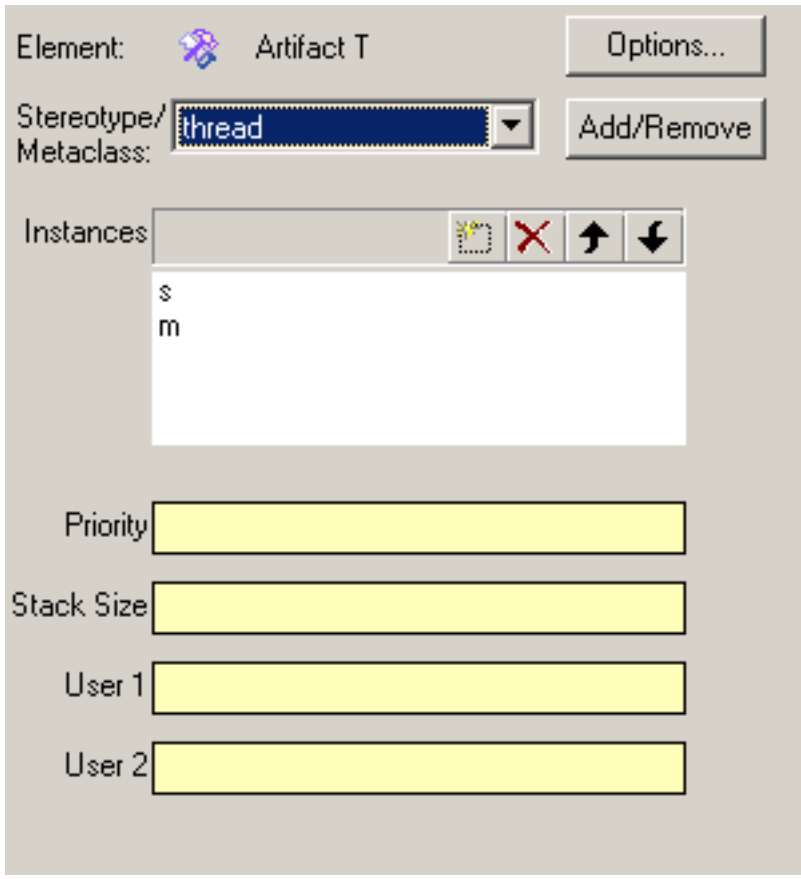



*Figure 227: Properties for thread artifact T*

In addition to the list of instances that are to run in a thread, it is also possible to set the priority and stack size for each thread. The two attributes named "User 1" and "User 2" are available for the user to specify additional properties for the thread. (In this example none of these additional values are used.)

Three more build artifacts are provided in the example:

- Single – This Build Artifact is not composed of any thread artifacts and is used to build a bare version. In this case the behavior is the same as that of the "One" thread.
- ModelVerify – This artifact is used to build a Model Verifier executable.

### Building

Each of the Build Artifacts described earlier can be built by right-click on the artifact and selecting **Build** on the shortcut menu, or from the **Build** menu using the command Build Configuration.

In this case, the class diagram named “Configuration ALL” can be used to build the four Build Artifacts “ModelVerify”, “One”, “Two” and “Three”.

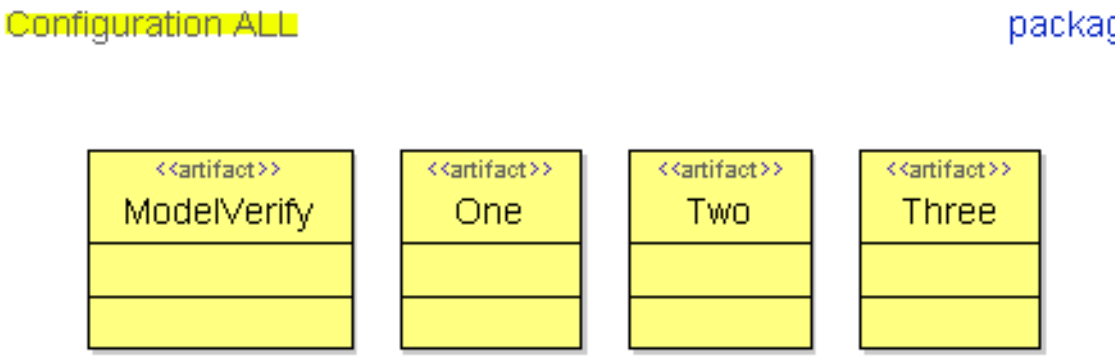



*Figure 228: Configuration diagram*

### Execution

Only the executable produced by the “ModelVerify” Build Artifact can be launched from within the Tau framework. To launch any of other executable a command shell is required.

### Execution performance

The three threading scenarios have different timing characteristics, and hence show different execution performance. In the example, signal sending is the main issue. The following execution times were measured on a 800MHz Intel CPU running Windows 2000.

| **Artifact** | **One** | **Two** | **Three** |
|---|---|---|---|
| Time (s) | 0.7 | 0.8 | 15.0 |

Although these time measurements are crude, they do highlight the additional overhead due to signal sending between threads.

# 27

# *Building Applications Reference*

This document is a reference to the build of C, C++ and Java applications from UML models.

# Interactive Build Interface

The Application Builder manifests itself through the following components: Build Artifacts, Configurations, a Build Wizard, a Build Menu and build settings contained in Stereotypes and attributes that are accessed through the Properties Editor.

## Build Artifact

A build artifact is an artifact with a build stereotype applied. The build stereotype defines which kind of build artifact it is, for example a C++ or Java build artifact. If more than one build stereotype is applied to an artifact, it is not defined which is used if a build operation is initiated on the artifact.

Build artifacts are displayed in the workspace window by using icons with a special graphical appearance to easily associate a build artifact with its function (defined by the attributes in the «Icon» stereotype). This makes it easier to distinguish between build artifacts. The figure below demonstrate the connection between the icon of a build artifact shown in the model view and the <<Icon>> stereotype on the corresponding build stereotype.

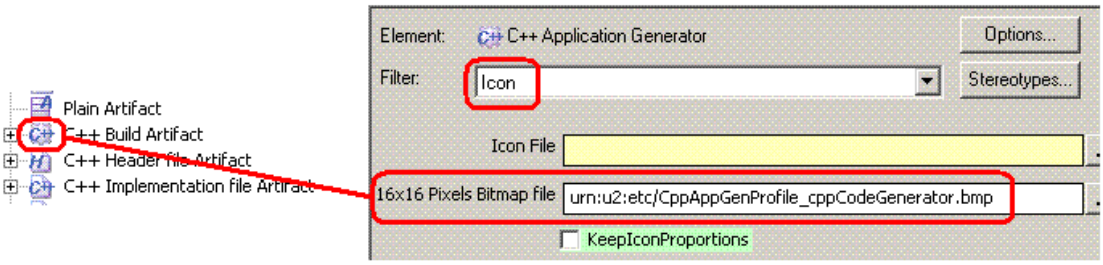



*Figure 229: Artifact Icons*

**See also**

"Using Build Artifacts" on page 833.

## Build Stereotype

The definition of a build stereotype defines a set of build operations and the build settings. The build settings are the attributes of the build stereotype and represent the settings for all tools associated with the build stereotype, such

as code generators or run-time systems. The build operations specify the various actions that can be initiated on a build artifact. The build operations appear in the build shortcut menu.

All build stereotypes inherit from the (abstract) generic build stereotype «build».

**Note**

*The generic build stereotype «build» is normally hidden in order to reduce the visible stereotypes.*

## Build Operations

There are two kinds of build operations; "dynamic" and "static".

A dynamic build operation is defined using an operation of a build stereotype. That is, the build stereotype “owns“ the definition of the operation. The name of the operation is used in the Build shortcut menu. This makes it possible to choose any name of the operation, not just “Generate”, “Build” or “Make” as mandated by the static build operations.

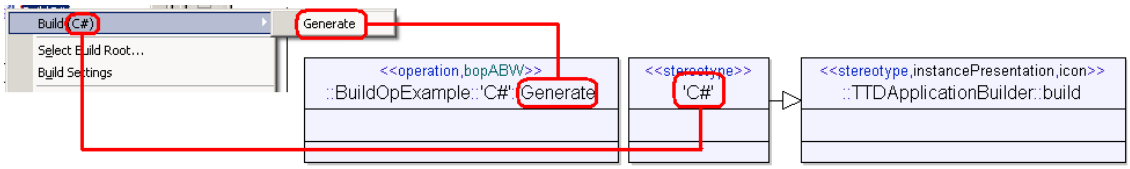



*Figure 230: Build Operation*

The figure above demonstrates the build artifact ‘C#’ with the build operation ‘Generate’ and how these names are used in the build menu.

The two stereotypes «bopAgent» and «bopABW» are used to define the actual action of the dynamic build operation. One of them must be applied to the definition of the operation. If neither is applied, the operation is not recognized as a build operation. If both are applied, the behavior is not defined.

«bopAgent» is used to specify that the action of the build operation is executed using Agents that execute within the address space of Tau. «bopABW» is used to specify that the action of a build operation is executed using an external executable in a process by itself. This means that Tau is not blocked during the execution of code generation and the code generation can be interrupted using the “Stop Build” button.

If the executable specified using «bopABW» is "ABWGen" agents are used in the same way as if «bopAgent» was used, with the important difference that they execute in an external process.

The static build operations are used to describe a static set of operations; "check", "generate", "build", "make", "clean", "launch" and "update".

**Note**

*Static build operations are deprecated. The new and more flexible dynamic build operations should be used instead. Static build operations are described here for completeness only.*

**See also**

"ABWGen" on page 2003

## Configuration

A Tau project contains at least one configuration, of which exactly one is the active configuration. The current project and the active configuration of the current project are visible in the project tool bar.

A configuration groups an arbitrary number of build artifacts. When building a configuration, all the build artifacts it contains are built in quasi-parallel fashion. The order in which they are built is described in Multiple build artifacts – configurations.

## Build Root

A build root specifies and delimits the subset of the model to build. More information about how to use build roots is available in the user's guide.

**See also**

"Using Build Roots" on page 842

## Build Type

A build type specifies which code generator is used by a Build Artifact. The available build types are:

- Model Verifier

- C Code Generator
- AgileC Code Generator
- C++ Application Generator
- Java
- Make settings
- Makefile generator

All of the build types above inherit from the generic build type **build**, which does not define any particular code generator and it is not possible to use for real builds.

**See also**

"Using Build Types" on page 843

## Build Settings

Build settings control how to build a model. Each Build Type has its individual build settings. These build settings are implemented as Build stereotypes and properties. For each Build Artifact, a number of stereotypes can be attached.

Below is a summary of the build settings available for the supported build types.

### Model Verifier settings

This build type gives you the option to specify the following settings:

- Specifying a Target Directory
- Including support for sending signals to the environment and receiving signals from the environment
- Specifying the host platform
- Which C/C++ compiler to use
- Supplying a make template file
- Option to compile in C++ mode
- Specifying to compile the generated code as an application or as a library
- Convenience settings such as verbosity and suppressing warnings.

### C Code Generator settings

This build type allows to specify the same settings as for the Model Verifier, including the following additions:

- Option to generate a threaded application
- Prefixing of generated names in C
- A user-defined (customized) run-time library can also be specified as an alternative to one of the defined libraries that are provided with the C Code Generator.

### AgileC Code Generator settings

This build type allows to specify the settings that can be found for the Model Verifier and C Code Generator, and with the following addition:

- Advanced settings for naming of generated code
- Settings for handling of processes, signals, timers
- Settings that impact the run-time performance such as dynamic memory management
- Run-time error detection in the generated code.

### C++ Application Generator settings

This build type allows to specify the settings that can be found for the C++ Application Generator. Some example settings include:

- Specifying a Target Directory
- Specifying prefixing of names.
- Options for how to layout and organize the generated code.
- How to link with the Tau Object Run Time (TOR) library.
- Which unit of time to use in a generated application which makes use of timers.
- Which port and host to use when debugging a generated C++ application with Tau.
- If changes made in generated files should be possible to round-trip back to the UML model.
- Whether code generation and/or model update should be automatically performed or not.

See Translation Options for a complete listing.

### Java code generator settings

This build type allows to specify the settings that can be found for the Java code generator. Some example settings include:

- Specifying a Target Directory
- Which Java version to use.
- Whether code generation and/or model update should be automatically performed or not.

See Java Build Artifact Settings for a complete listing.

### Make settings

This build type allows to specify the following:

- Which platform specific conventions should be used when calling "make"
- Which make file to submit as input
- Specifying user defined options to "make"

### Makefile Generator settings

This build type allows to specify the following:

- Where to put the generated make file
- Which dialect of "make" to use
- Specifying user defined section to add to the generated make file.

## See also

"Model Verifier" on page 1001

"AgileC Code Generator" on page 962

"Make settings" on page 999

"Makefile generator" on page 1000

# Build Wizard

The build wizard dialog prompts the user for the information required for a build in the following cases:

- When a build is initiated on a Build Artifact with incomplete settings. In this case the Build Root and/or build type are preset to the value of the build artifact. The values that are defined when using the wizard are then stored in the build artifact.
- When a build is initiated on a model element that is not a build artifact, and has no build artifact associated. In this case the build root is filled in.
- When a Configuration build is initiated and the list of build artifacts in the configuration is empty.

### Properties specified in the build wizard

The build wizard prompts the user to specify the following properties:

- **Manifests**

  This field displays the fully qualified name of the Build Root to build. **Manifests** specifies a relation between a Build Artifact and the element (e.g. a class or package) that is used as root for a build.
- **Set**

  Pressing the **Set** button allows to designate the build root in a class hierarchy tree, based on the current project model.
- **Build type**
- The **Build Type** choice is used to designate the Build Type. The available choices depend on which build types are enabled in the project and on the literal values available in the build type enumeration type (which is to be set as a property on a build artifact).**Add artifact to active configuration** (optional choice)

  This toggle allows to insert the artifact (which is created after the wizard has completed) into the currently active Configuration. See "Multiple build artifacts – configurations" on page 835.
- **Properties** (optional choice)
  - This button opens a modal Properties Editor where properties that are not accessible in the build wizard, such as the build artifact name, can be changed.

## File Artifact

A file artifact is an artifact with a file stereotype attached. All file stereotypes inherit from the generic stereotype «file»

These artifacts are used to specify how model elements are implemented on, or 'connected' to files. They are also used to model dependencies between UML sources and executables or libraries, to ensure a proper chain of build dependencies.

By adding a file stereotype to an artifact, it becomes specialized and is given a purpose when transforming the abstract model to a concrete implementation – i.e. when building the application.

The following file artifacts are available to use when building applications using the C++ Application Generator:

- **cppHeaderfile**
- **cppImplementationfile**
- **executable**
- **library**
  - **dynamicLibrary**
  - **staticLibrary**
- **makefile**

The following file artifacts are available to use when building applications using the Java code generator:

- **javaFile**
- **jarFile**

**Note**

*When building applications with the present version of Tau, file artifacts are used by the C++ Application Generator and the Java code generator only. The connection between model elements and files cannot be specified using file artifacts for applications generated by the AgileC Code Generator, C Code Generator and Model Verifier.*

**See also**

"Using File Artifacts" on page 836.

## Thread Artifact

A thread artifact is an artifact with the stereotype «thread» attached. Thread artifacts are stereotyped classes that represent the threads within an executable application. The Class diagram editor is used for modeling such classes.

Thread artifacts are used exclusively with the AgileC Code Generator and C Code Generator.

For the C++ Application Generator and the Java code generator threads are defined and manipulated programmatically using utilities in the TOR library.

**See also**

"Using Thread Artifacts" on page 836

## Project Tool Bar

The project tool bar consists of the Active project, Active configuration, Active tool and a Build tool bar with quick buttons.

### Active project

This box allows to switch between the projects contained in the currently loaded workspace.

### Active configuration

This box allows to change the active Configuration. The list of items available for selection contains the configurations defined in the active project. Changing the active configuration is a handy feature when managing multiple builds from a model.

### Active tool

This box defines the 'tool' used when building applications. At present time it contains one item only – **Application Builder**.

### Build tool bar

The build tool bar contains quick buttons to access the frequently used commands that order the build of a configuration:

| Command | Shortcut |
|---|---|
| Update Configuration | SHIFT F6 |
| Generate Configuration | F6 |
| Build Configuration | F7 |
| Stop | CTRL + Scroll Lock |
| Execute Configuration | F5 |

**See also**

"Working with Projects" on page 35 for a user guide to projects.

"Using Configurations for Build" on page 851 for a guide to using configurations when building applications.

"Build Menu" on page 941 for a reference to these commands.

## Build Menu

The build menu provides all of the build commands that are supported for a Configuration build.

- The most frequently used commands are also available as quick buttons in the project tool bar.
- When right-clicking a Build Artifact in the workspace window, the shortcut menu holds the build commands that are supported for an artifact build.

Build operations are divided into several ordered steps:

1. Prepare for build and check. This step is always performed (except for the Stop command). Here it is checked that the configuration and build artifacts are valid and that the model is correct and suitable for build.
2. **Generate**. This step generates code from the model and for C/C++ code generation also a make file.
3. **Build**. This step executes "make", if a make file was generated in the Generate step.
4. **Launch**. This step executes the generated executable, if the previous steps resulted in such an executable.

The steps are executed in the listed order, and repeated for each build artifact in the configuration. The last step executed depends on which build operation is invoked – how "far" the build should go.

### Stop

This command stops all build operation in progress, and terminates the currently active Model Verifier session (if any).

### Check Configuration

This command performs a semantic check on the Configuration. The semantic checker is operated in the context of the build type defined by the build artifact(s) contained in the configuration.

Check Configuration differs from the regular **Check** command in that the model is checked with respect to additional rules imposed by restrictions in the code generator, or the target language, defined by the build type. The intention is to catch such problems as early as possible in the build chain.

**Note**

*Many, although not all of the UML constraints imposed by code generator specific restrictions are found by the semantic checker. However, some constraints are found first when generating code. More information on restrictions can be found in "Restrictions in UML Support when Building C Applications" on page 951 and "Restrictions in UML Support when Building C++ Applications" on page 958.*

### Generate Configuration

This command first checks that the configuration is suitable for build (see Check Configuration.

Next, if the part of the model manifested by the Build Root of the build artifact is found semantically correct and is supported by the code generator defined in the build artifact, code is generated according to the settings defined in the build artifact. For C and C++ a make file is also generated.

### Build Configuration

This command first checks that the Configuration is suitable for build, and then generates code and make files (see Check Configuration and Generate Configuration).

After code and make files are successfully generated, the generated code is compiled and linked according to the make settings defined in the build artifact.

### Execute Configuration

This command first checks, generates and builds the Configuration (see Check Configuration, Generate Configuration and Build Configuration. If the build is successful, the resulting application is launched.

**Note**

*Only Model Verifier and C++ applications take advantage of this feature.*

### Update Configuration

This command updates the part of the model defined by the build roots contained in the build artifacts in the Configuration from the source files that are manifested by Using File Artifacts. Changes made in these source files are propagated back to the model.

**Note**

*Model update is not supported by the AgileC Code Generator, C Code Generator and Model Verifier.*

### Clean Configuration

This command performs a “make clean” operation on the make file. This operation deletes object files, libraries and executables resulting from a previous build.

### Start Model Verifier

This command launches a Model Verifier application without building it. Following the command, the user is prompted to specify a file containing a Model Verifier executable.

**Note**

*No checks are performed to verify that the application originates from the model currently loaded in the workspace. If that is not the case, or if the model has changed since it was built, incorrect or incomplete information may likely be displayed by the Model Verifier.*

# Build Shortcut Menu

The build shortcut menu (appears when right-clicking a Build Artifact or a model element that is suitable to use as Build Root) has one of two appearances.

### Build shortcut menu on build artifacts

When activated on a Build Artifact, the build shortcut menu displays the following:

**[Build type]** followed by a submenu with the available build commands. These build commands are identical to the commands on the Build Menu (for example Check, Generate, Build, Update, Clean and Launch).

### Build shortcut menu on model elements

When activated on a model element that is suitable to use as build root (a package or a class), the build shortcut menu appears as follows:

- A list of all build types that are currently enabled in the project
- Each of these build types has a submenu, with all build artifacts with matching build type in the active Configuration
- A command to create a new build artifact.

# Batch Build Interface

Tau can be invoked to build applications from the command line prompt using the command `taubatch`.

```
taubatch [Options]
```

## Input

The input to `taubatch` is specified as an option in the command, and consists of the following in combination:

- A project file (`.ttp` file), using the `-p "project file"` option
- A Configuration, using the `-c "Configuration Name"` option
- A Build Artifact, using the `-g GUID` or `-o element` option.

## Output

### Build index file

`taubatch` generates a build index file in the Target Directory, containing a list of files generated the last time this Build Type was used. This file is empty if the build failed for some reason.

The name of the build index file is:

```
build_index_" <guid-of-artifact> ".xml
```

in which the GUID of the artifact is mangled so that all tokens not in the set "0-1, @-Z, a-z, _" are replaced with a "_".

The build index file uses XML for representing the information, according to the following build index file DTD:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE BI [
  <!ELEMENT list (file)+ >
  <!ELEMENT file EMPTY>
    <!ATTLIST file name CDATA #REQUIRED>
    <!ATTLIST file time INT #REQUIRED>
    <!ATTLIST file type CDATA #IMPLIED>
]>
```

Each entry in the list describes a generated file and contains the following attributes:

| Attribute | Description |
|---|---|
| Name | The name of the file |
| Type | The type of the file, for example “debugger” for a “Model Verifier” executable. |
| Time | Time when the file was generated, in seconds, counting from 1970-01-01:00:00:00 |

### Code generator output

The output from the code generators consists of C/C++/Java source files, and possibly also make files and other useful files and reports.

**See also**

Chapter 35, C Code Generator Reference

Chapter 39, AgileC Code Generator Reference

Chapter 52, C++ Application Generator Reference

Chapter 43, Java Code Generator Reference

## Options

The options recognized by `taubatch` are the following:

### -B

Short form for `--Build`

This option specifies to order a build including semantic check, generation of code and make file and lastly execution of ‘make’ on the ‘Makefile’.

This option cannot be used in combination with any of the options -C, -G and -S

### -c “Configuration Name”

A short form for `--config "Configuration Name"`

This option specifies to use the Configuration specified by "`Configuration Name`" as input to the build. As a result of this, all of the build artifacts contained in the configuration are built.

**-C**

Short form for `--Clean`

This option performs a clean of the files generated by the 'make' utility – usually removal of generated object files, libraries and executables. Which flavor of 'make' and which 'Makefile' is used is defined in the Build Artifact used for the build.

This option cannot be used in combination with any of the options -B, -G and -S

**-g GUID**

A short form for `--guid GUID`

This option uses a build artifact referred to by its Globally Unique Identifier as input to the build.

A way to obtain the GUID in the general case is to look for it in the `.u2` file where the build artifact is stored. GUIDs for elements that are essential for the build (such as build roots) are also found in a generated table of contents (see option "-T").

**-G**

Short form for `--Generate`

This option checks and generates code and a make file (in case of C/C++) for the subset of the model manifested by the Build Root(s) defined in the input to the build.

This option cannot be used in combination with any of the options -B, -C, and -S

**-h**

A short form for `--help`

This option prints help about the available options on `stdout` and then exits `taubatch`

**-l** (lower case L)

Short form for `--listbt`

This options prints the build types available in the project file on `stdout` and then exits `taubatch`

**-o element**

Short form for `--object element`

`element` is the qualified name of the build artifact to submit as input to `taubatch`

This option is mutually exclusive with the options -g GUID and -c "Configuration Name"

**-p "project file"**

Short form for `--project "project file"`

"`project file`" is the name of the project file (`.ttp` file) to submit as input to the build.

**-r**

Short form for `--rebind-by-name`

This option forces the tool to use full name resolution when a model is read. This is usually not necessary and also causes the build to take longer time.

**-S**

Short for `--Semantic-check`

This option orders a build that is limited to checking the input for semantic correctness and that it is suitable for build.

This option cannot be used in combination with any of the options -B, -C, and -G

**-T**

Short form for `--TOC`

This option generates a file containing a table of contents of a project file, holding information about:

- The name space structure (packages and classes) in an indented list
- The available build artifacts
- The available configurations.

**-V**

Short form for `--Version`

This option prints the version of `taubatch` on `stdout` and then exits

# Examples of Using taubatch

**Example 332: Example of using the -T option**

```
$ taubatch -p PingPong.ttp -T
Telelogic Tau Application Builder

TAB1026: Table of contents
DJhxxINnrzALDYVY-Le5U27I : pingpong_artifact
L0SZSIZK9BlLsxk94E9IS*dE : PingPong
b7xI3LmvPyFLuTiXGEwHCW1I :   Match
w2rrRLYFz60L-H2dZL99rzgV :     Player2
bKPz0VVWI2FLMsNWILWqtuoI :     Player1
7MNyDEJTh40Ld5wkyIy1RczL : Agile

Configurations:
  ALL
  Default
  MyConfig
```

**Example 333: Build of a configuration and a build artifact**

To build the Configuration “MyConfig”:

```
taubatch -p PingPong.ttp -c MyConfig
```

To build the Build Artifact “Agile”:

```
taubatch -p PingPong.ttp -o Agile
```

Both MyConfig and Agile must be defined in the project file `PingPong.ttp`

# Restrictions in UML Support when Building C Applications

This section lists the restrictions in UML support when building applications using any of the C based build types.

- Model Verifier
- C Code Generator
- AgileC Code Generator

## Restrictions in C build types

### Active code generators

In order to generate code efficiently, it is recommended that each project limits the number of active code generators to one. E.g. if both the C++ Application Generator and the Model Verifier code generator are active in a project, the code generation may require longer time to complete and/or require larger memory resources.

### All C build types

The following UML constructs, or use of UML, are not supported by any of the C build types (Model Verifier, C Code Generator and AgileC Code Generator)

### Classes

- Comparison operators (‘==’ and ‘!=’) in generated C code may sometimes not work for instances of external classes. This can cause compilation errors. These operators may work in some conditions. An example is if a user-defined stand-alone operator ‘==’ for a C++ class is placed in a header file and the same header file is included into a model by the means of the CApplication stereotype. Then the operator will work for instances of the imported C++ class.
- Dynamic creation of instance of active classes that have internal structure.
- Creation of passive class is not allowed to be a standalone action.
- Passive classes may not contain operations with states.

- Passive classes may not realize any interfaces.
- Non const attributes in packages are ignored unless the package is externally defined in C or C++.
- Methods of a passive class cannot call methods of an active class.
- Multiplicity constraints are not supported for a passive attribute.

**Example 334: Multiplicity constraints for a passive attribute**

```
class A { }
part A [1..10] mypart; // constraint [1..10] is ignored
```

## Constructors and destructors

- Constructors must have a realizing method.
- Static constructors and destructors are not allowed.
- At most one constructor state machine can be present in a class.
- It is not possible to call inherited constructors from a constructor in a passive class.

Destructors are only partially supported. To work-around this:

- Active classes should be implemented to be able to receive a signal that eventually stops the state machine after required clean-up.
- In passive classes, destructors cannot call other operations.

**Example 335: Destructors cannot call other operations**

```
class B
{
  public void Foo() { }
}
class A
{
    B refB;
    ~A()
    {
      refB.Foo(); // Not supported, error TCC0924
      delete refB;
    }
}
```

## Ports

- Ports are not allowed in passive classes.

- Ports cannot be redefined.

**Operations**

- Calling of superclass operations using a qualifier is not supported.
- Calling operations on an active class from a global operation is not supported.
- Return value from value-returning operations must be handled in the left hand side of an assignment expression.
- A “delete” operation may not be applied to a part.
- Redefinition of a finalized operation is not allowed.
- Static operations cannot access non-static attributes.

**Signals**

- Parameters in received signals must not be omitted.
- Signal Sending via “all” is not supported.
- Optional parameters are not supported. Multiplicity `[0..1]` on signal parameters is ignored when generating C code.

**Operators**

- For templates Own and Oref:
  The support in the C code generators for templates Own and Oref is limited. It is not recommended to use these templates.
- Bit type:
  The relation operators <, >, <=, >= are not supported by the C code generators.
- Operation “Octet '[]' (Octet, Integer, Bit)” in Octet type:
  Assigning a new value to a bit in Octet is not supported in the C code generators.
- For operation “Octet '\=' (`Octet`, `BitString`)” in Octet type:
  Assigning an Octet from a `BitString` value is not supported in the C code generators.
- The operator power is not supported by the code generator, and should be implemented externally, if used.

**Timers**

- Timers must not be defined in packages. Timers must instead be defined in the active classes where they are used.

- Operation `active` on timers is not supported for timers with parameters. Only timers without parameters can be tested.

**Actions**

- Try Actions are not supported.
- Join, Return and Stop actions are not allowed inside loops.
- The Start Transition must have a terminating Action.
- Receiving signal “none” is not supported.
- Deep History Nextstate Action is not supported.

**Attributes**

- Static attributes are not supported – such attributes correspond to global data in C and the generated C code has no support for accessing such variables in a safe way in real-time applications.
  - If global data is needed in an application, such global data could be implemented in an active class that can accessed by all active classes, through a signal interface to read and write global data.
  - An alternative lightweight solution is to implement the static attributes as static C variables.
- Inheritance of (virtual) operations from a parent class requires the following:
  - Operation attributes must be re-declared in the (finalized) child class
  - Attributes must however not use the same name in the parent and child operation.
- Integer and enum types are not type compatible. Attributes of such types cannot be mixed in expressions and operators.

**Miscellaneous**

- The STL support is currently limited to work with the C++ Application Generator. STL can not be used with the C Code Generator. In general, any imported C++ code that uses templates can not be used with the C Code Generator.
- The C code generators do not support Function pointer as described for C/C++ import.
- Range check expression is not supported when generating C code.

- Regarding Model Verifier 2.2 to 2.3 migration, the naming convention has changed for certain data types. This will result in the `.ttdcfg` file may not work if the model contains two or more enumerated types with identical literals.
- Use of literal values or expressions containing only literal values as both arguments for a comparison operator may yield an analysis error.
- The `noScope` stereotype was introduced to solve browser grouping problems. It is not recommended for any models intended for code generation.

**Example 336: Unsupported expressions**

The following type of constructs will not work, if used for example in a decision:

```
2+2==4
0!=1
```

- Multiple inheritance is not supported.
- Dynamic creation of instance of a choice is not allowed.
- A model must contain at least one active class otherwise the application has no behavior (the application is possible to build but will not execute).
- User-defined templates are not supported.
- State expression is not supported.
- A model intended for C code generation should not depend on actors, sequence diagrams and use-cases. These UML language constructs are informal and are not supported at C code generation.
- Using the API to load a profile package (addin), using `u2::LoadLibrary` and letting the tool calculate the dependency will result in that the C Code Generator will ignore the package.
  For a profile package to be generated C code for, the profile package should be loaded using the following Tcl command instead:
  `u2::LoadFile <model> <profile pacakge> false<=loadAsProfile>`
  This will load the profile package as a file into the model. The user can then set up the required dependency to the profile package (thus forcing code generation of the package).

### AgileC Code Generator

In addition to the restrictions that are applicable for All C build types, the following UML constructs, or use of UML, are not supported by the build type that uses the AgileC Code Generator.

#### Classes

- Active classes that have internal structure can only have Multiplicity one (`[1]`).
- Declaring an infinite number of instances of active classes.
  - Although this is supported, the use of infinite number of instances is not recommended for execution performance reasons.

#### Parts

- It is not possible to add to a part anything different from the construct "new <active class name>". For instance the following is not supported:

```
active class A { }
part A [*] mypart;
A tmp;
tmp = new A();
mypart.append(tmp); // not possible
```

#### Operations

- Operations with states are not supported.
- Call of virtual or redefined operations from an operation is not supported
- Call of operations in other active classes from an operation is not supported.

#### Timers

- Timers with any other parameter type than Integer.
- Timers with more than one parameter.
- Timer duration cannot be set with Real values. Timer duration must instead be an expression of type Duration.

#### Miscellaneous

- "`any`" decision.
- Informal decisions.
- Transitions with guards without an event expression.
- Part indexes are not supported.

- Call “this”.

## Reserved words

### All C build types

These restrictions are applicable to all C build types (Model Verifier, C Code Generator and AgileC Code Generator):

The following words **must not** be used in order to name any item in UML models (such as name of classes, attributes...) if “C” applications are to be built from the model. (The model although it is correct from a UML perspective will not be accepted by the code generator.)

```
break, choice, optional, remote
```

### Model Verifier execution

The following words are accepted by the C code generators, and the result will become a working application. However the Model Verifier sequence diagram trace may report an error when encountering an entity with such a name. In such case the trace will not display the expected result. This will also apply to any name containing a hyphen '-'.

```
action, all, alt, as, before, begin, block, by, comment,
concurrent, condition, connect, create, decomposed,
empty, end, endconcurrent, endmsc, endexpr, env, exc,
expr, external, found, from, gate, in, inf, inline, inst,
instance, loop, lost, msc, mscdocument, msg, opt, order,
par, process, reference, related, reset, service, seq,
set, shared, stop, subst, system, text, timeout, tim, to,
via
```

**Note**

*This restriction applies to the words written in any case configuration, the Model Verifier sequence diagram trace is **case insensitive**, e.g. both System and SYSTEM are reserved.*

# Restrictions in UML Support when Building C++ Applications

This section lists the restrictions in UML support when using the C++ Application Generator.

The following restrictions apply. For a more detailed information on what is supported it is important to study the General Translation Rules and C++ Textual Syntax.

### Active code generators

In order to generate code efficiently, it is recommended that each project limits the number of active code generators to one. E.g. if both the C++ Application Generator and the Model Verifier code generator are active in a project, the code generation may require longer time to complete and/or require larger memory resources.

### UML restrictions

- No support for deferred events in state machines (also known as 'save' of signals)
- No support for operations in datatypes
- No support for UML reception mechanism
- The C++ code generator is usually able to handle circular include dependencies, by adding appropriate forward declarations. However, in some cases a manual resolution of such dependencies are required. In case of circular include dependencies (Class1.h includes Class2.h which includes Class1.h) compile errors can occur. This can be solved by adding a forward declaration like:

```
//<USER>
    class Class1;
//</USER>
```

- **Restrictions in roundtrip C++ support**No support for function pointers
- No support for template specialization
- No support for pointers to class members

**See also**

Chapter 51, C++ Textual Syntax

"General Translation Rules" on page 1542 in Chapter 52, *C++ Application Generator Reference*

# Restrictions in UML Support when Building Java Applications

This section lists the restrictions in UML support when using the Java code generator.

The following restrictions apply. For a more detailed information on what is supported it is important to study the Java Code Generator Reference. In general UML constructs not mentioned in that document is considered unsupported.

### Active code generators

In order to generate code efficiently, it is recommended that each project limits the number of active code generators to one. E.g. if both the C++ Application Generator and the Java code generator are active in a project, the code generation may require longer time to complete and/or require larger memory resources.

- **UML restrictions**No support for deferred events in state machines (also known as ‘save’ of signals)
- No support for operations in datatypes
- No support for UML reception mechanism
- No support for architectural constructs such as ports and connectors
- No support for parameters with default values or optional parameters
- No support for operations implemented by means of state-less state machines
- No support for syntypes

# 28

# *Stereotypes for Code Generation*

This section contains a reference to the code generator and build artifact stereotypes, listed in alphabetic order.

# Stereotypes

## AgileC Code Generator

This stereotype inherits from the stereotype «build» and contains the attributes that control the generation of C code performed by the AgileC Code Generator.

Some of the attributes used by the AgileC Code Generator have the same name and semantics as those used by the C Code Generator. They are listed below for completeness. Descriptions are available by following the references to the section about the C Code Generator stereotype.

### Code generation properties

This control groups attributes that define properties for the code generated by the AgileC Code Generator.

- **Name mangling**
  - Type: {Prefix | Suffix}
  - Default: Suffix

  This attribute controls the name mangling of generated names in C code. As the name scopes in UML and C are not the same, it is not possible to use the UML names directly in the generated C code.

  By default the AgileC Code Generator will add a suffix to the UML name and use that as an identifier in C, to ensure that the C identifier is unique. If a user has used long UML names and at the same time the C compiler uses a limited number of characters to determine if two names are the same, name clashes might occur. By using prefixes instead of suffixes such situations can be avoided.

- **Comments**
  - Type: {Sparse | Structure | Explanation}
  - Default: Sparse

  In the C code generated for state machines it is possible to specify the level of comments that are added to the code. The following levels are possible:
  - **Sparse**: Only some comments used to identify transitions are included.
  - **Structure**: As Sparse, adding a comment for each translated UML symbol.
  - **Explanation**: As Structure, adding comments giving some explanations to the code.
- **Connector name, Constant name, Type name, Literal name, Signal name**
  - Type: String
  - Default: see table below

| UML entity | C name |
|---|---|
| Connector name | `cha_%n` |
| Constant name | `con_%n` |
| Type name | `typ_%n` |
| Literal name | `lit_%n_%s` |
| Signal name | `sig_%n` |

These attributes control the prefixes or suffixes of generated C names in the Interface header file (.ifc) for connectors, constants, types, literals and signals.

`%n` is the name of the entity, `%s` is the name of the scope. Omit `%n` in order to exclude the definitions completely from the file.

- **Operators in environment header file**
  - Type: Boolean
  - Default: True

  This attribute configures if operators should be present in the Interface header file (.ifc), or not.
- **Include references to UML source as comments**
  - Type: Boolean
  - Default: False

  Enabling this feature instructs the AgileC Code Generator to include calls to trace functions in the generated code. This attribute must be set True if the feature Print UML level trace on stdout is enabled.
- **Always include** `stdio.h`
  - Type: Boolean
  - Default: False

  This attribute defines if the file `stdio.h` should be included in the include files submitted to `make`
- **Print UML level trace on** `stdout`
  - Type: Boolean
  - Default: False

  This attribute defines if a trace of important UML actions and events should be printed on `stdout`. Important actions are receiving/sending signals, create and timer actions.
- **Generate MISRA compliant code**
  - Type: Boolean
  - Default: False

  This attribute defines if the generated code should be MISRA compliant. The generated code may not be fully patronized. For examples there may be duplicated code segments rather than unconditional jumps to labels.

## Compile and link

This attribute is described in the section about the C Code Generator stereotype.

### Dynamic memory allocation

This control groups the attributes that define how to manage dynamic memory.

- **Use memory management package provided with AgileC**
  - Type: Boolean
  - Default: False

  This attribute defines to use the memory management package provided in the library, instead of the OS functions `malloc` and `free`. If this package is used, then the attribute Memory pool size should be set to an appropriate value by the user.
- **Memory pool size**
  - Type: Integer
  - Default: 8192 (bytes)

  This attribute defines the number of bytes to be used by the pool of dynamic memory. To avoid unnecessary problems this value should be a multiple of 16 (see Minimum block size).
- **Minimum block size**
  - Type: Integer
  - Default: (empty)

  This attribute defines the minimum size of a block of allocated memory. This value should be a multiple of 16, with 16 the lowest value.

### Error detection

This control groups the attributes that define the application error detections performed at run time.

- **Basic run-time error check**
  - Type: Boolean
  - Default: False

  This attribute enables the run time checks of the following basic state machine properties:
  - Check that memory is available for creation or sending of signals
  - Check that memory is available when allocating the parameters of a signal
  - Check that memory is available for creation of timer instance
  - Check that `xOutEnv()` is present when a signal is sent to the environment
  - Check when a signal is discarded
  - Check attempts to create more instances when maximum limit is reached.
- **Index checks in arrays**
  - Type: Boolean
  - Default: False

  This attribute enables the check that indexing in array is within allowed range.
- **Syntype range checks**
  - Type: Boolean
  - Default: False

  This attribute enables the check that syntype values are within allowed values.
- **Checks in predefined operators**
  - Type: Boolean
  - Default: False

  This attribute enables the check on error situations when executing predefined operators.

- **Check that the decision value matches an answer**
  - Type: Boolean
  - Default: False

  This attribute enables the check that there is a valid path out from a decision.
- **Checks for null pointers**
  - Type: Boolean
  - Default: False

  This attribute checks that pointers have not the value NULL before dereferencing them.
- **Enable checks inside the memory management package**
  - Type: Boolean
  - Default: False

  This attribute enables checks on the memory management package. This package is in turn used by setting the attribute Use memory management package provided with AgileC to True.
- **Print errors on** `stdout`
  - Type: Boolean
  - Default: False

  This attribute defines if error messages should be printed on `stdout`
- **Print errors on** `stderr`
  - Type: Boolean
  - Default: False

  This attribute defines if error messages should be printed on `stderr`
- **Print errors with error messages, not only numbers**
  - Type: Boolean
  - Default: False

  This attribute defines if error messages should be printed, or the error message number only.

- **Call a user provided function at warnings**
  - Type: Boolean
  - Default: False

  If this attribute is defined, a user provided function is called in case of a warning detected at run time:

  ```
  void xUserWarnAction (unsigned char WarningNumber);
  ```

- **Call a user provided function at errors**
  - Type: Boolean
  - Default: False

  If this attribute is defined, a user provided function is called in case of an error detected at run time:

  ```
  void xUserErrAction (unsigned char ErrorNumber);
  ```

### Environment

This control groups the attributes that define the call of the environment functions `xInitEnv` / `xCloseEnv` / `xInEnv` / `xOutEnv` at appropriate places in the generated application.

- Type: Boolean
- Default: False

Calls to each of these functions can be defined separately. These functions are to be supplied by the user and their purpose and design guidelines are described in detail in the reference to the AgileC Code Generator.

### Extra code

Type: String (multi-line)

Default: (empty)

This control specifies two attributes: **Head** and **Tail**, each of them specifying an optional user code section to be placed in the beginning or end of the file `uml_cfg.h`

## Generate environment template functions

This attribute is described in the section about the C Code Generator stereotype.

### Make template file

This attribute is described in the section about the C Code Generator stereotype.

### Operators in environment header file

This attribute is described in the section about the C Code Generator stereotype.

### Process properties

Type: Integer (positive)

Default: 5

In case there are parts with unlimited maximum number of instances, memory will be dynamically allocated at start-up to satisfy the needs for the number of initial instances, and adding the value given in this attribute. The value must be positive.

### Signal properties

This control groups the attributes that govern how signals should be handled at run-time.

- **Use priorities on signals**
  - Type: Boolean
  - Default: False

  This attribute defines if the signal queue should be sorted first in priority order and then in arrival order, or in arrival order only.
- Length of static signal queue
  - Type: Integer
  - Default: 20

  This attribute defines the length of the static signal queue. If dynamic signals are not used (see Prevent dynamic signals), then an error will occur if the signal queue is full when an attempt to send a signal is made.

- **Prevent dynamic signals**
  - Type: Boolean
  - Default: False

  This attribute is used to 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 the attribute Length of static signal queue to an appropriate value
- **Signals with dynamic parameters are never discarded**
  - Type: Boolean
  - Default: False

  This attribute defines removal of the code for freeing signal parameters in case a signal is discarded (only applicable if signals with dynamic parameters are used).

  If this attribute is set to True, and a signal with dynamic parameters is discarded, a memory leak will be introduced (memory will not be de-allocated and returned to the pool).
- **Priority for timer signals**
  - Type: Integer
  - Default: 50

  This attribute sets the priority for all timer signals.
- **Priority for startup/create signals**
  - Type: Integer
  - Default: 50

  This attribute sets the priority for all startup/create signals.
- **Default priority for signals**
  - Type: Integer
  - Default: 50

  This attribute sets the priority for all signals that do not have an explicit priority defined.
- **Signal parameter size**
  - Type: Integer
  - Default: (empty)

  This attribute sets the size of inline field for signal parameters. Parameters larger than this require dynamic memory.

**Timer properties**

- **Length of static timer queue**
  - Type: Integer
  - Default: (empty)

  For timers without parameters, this attribute defines the maximum amount of possible active timers.

  For timers with parameters this value may, in extreme cases, be too small. If dynamic timers are not used (see Prevent dynamic timers), an error occurs if the timer queue is full at the same time as an attempt to set a timer is made.
- **Prevent dynamic timers**
  - Type: Boolean
  - Default: False

  This attribute controls the use of dynamic memory for timers. This feature should normally be turned off in smaller systems, where dynamic memory is not to be used. In such a case, it is important to set Length of static timer queue to an appropriate value.

### Support C++

This attribute is described in the section about the C Code Generator stereotype.

### Suppress C level warnings

This attribute is described in the section about the C Code Generator stereotype.

**Target directory**

Type: String

Default: (empty)

This attribute controls where the files generated by the AgileC Code Generator are to be written on the file system. Both Absolute and relative path is accepted. If specified relative, the “root” is the location of the current project (`.ttp`) file.

This attribute has an empty default value, in which case the conventions for Target Directory naming and location take effect.

### Target kind

This attribute is described in the section about the C Code Generator stereotype.

### User defined kernel

This attribute is described in the section about the C Code Generator stereotype.

### Verbose mode

Type: Boolean

Default: False

This attribute controls if the AgileC Code Generator should print exhaustive reports and diagnostics in the message output area when generating code.

## build

This stereotype extends the Metaclass **Artifact** and serves as a 'base class' for all 'real' build stereotypes, which inherit from it. The stereotype «build» has no effect in practice and is not meant to be used by itself. Instead, the following stereotypes are available for practical use:

- «AgileC Code Generator»
- «C Code Generator»
- «C++ Application Generator»
- «Makefile generator»
- «Model Verifier»

The stereotype «build» is by default hidden in its profile and is documented for the sake of completeness.

### Target directory

Type: String

Default: (empty)

This attribute controls where the files, generated as the result of building using the current Build Artifact, are to be written on the file system. If the location of the file is specified as a relative path, the "root" is the location of the current project (`.ttp`) file.

This attribute has an empty default value, in which case the conventions for Target Directory naming and location take effect.

## C Code Generator

This stereotype inherits from the base stereotype «build» and contains the attributes that control the generation of C code performed by the C Code Generator.

### Advanced options

Type: String (multi-line)

Default: (empty)

This attribute is used to define additional advanced options to be passed to the C Code Generator. The options are appended after other code generation options are specified, in the order they appear in the text box.

**Note**

*By using this feature in any other way than the supported options described above, risk is that code generation options are overridden or changed in an undesirable way, resulting into unexpected behavior.*

The following options are supported:

- **Set-Signal-Number**
  This option instructs the C Code Generator to assign numbers to signals to/from the environment, in order to easier lookup the signals when sending them to the environment.

  As a result of this, a file named `<basename>.hs with information` about signal numbers assigned by the C Code Generator is produced. See "Improving performance of xOutEnv when many signals" on page 1050 for how to use this feature.

**Note**

*For this file to be correct, the complete application must be built. Partial builds may result in incorrect signal numbering.*

- **Set-SDL-Coder**
  This option instructs the C Code Generator to generate information about how data is encoded in the signals to/from the environment.

  This options results into the creation of two additional files, <base-name>_cod.h and <basename>_cod.c See "Encoding and decoding of signal parameters" on page 1040 for how to use this feature.

### Code generation properties

Type: String

Default:

| UML entity | C name |
|---|---|
| Connector name | `cha_%n` |
| Constant name | `con_%n` |
| Type name | `typ_%n` |
| Literal name | `lit_%n_%s` |
| Signal names | `sig_%n` |

These attributes control the prefix or suffix of generated C names in the System interface header file.

**Operators in environment header file**: see "Operators in environment header file" on page 975.

### Compile and link

Type: Boolean

Default: True

This attribute defines if a make file for compiling and linking the C files should be generated by the AgileC Code Generator or the C Code Generator and be automatically executed during the build process. (The options of creating and executing the makefile cannot be controlled individually.)

### Expand macros

Type: Boolean

Default: False

This attribute controls if macros in C code should be expanded and processed after code generation, to make the code more readable and easier to debug on C level. This processing is done by the C Compiler Driver utility.

**Note**

*Expanding macros is not supported with Target kind set to Win32.*

### Generate environment template functions

Type: Boolean

Default: False

This attribute controls if files with skeletons for the Environment Functions should be created by the {AgileC Code Generator | C Code Generator} when generating an application or a Model Verifier.

The System interface header file (`.ifc`), which holds an up-to-date definition of the interface to the environment is always created regardless of the value of this attribute.

### Make template file

Type: String

Default: (empty)

This attribute controls if the C Code Generator or AgileC Code Generator should use a given make template file (such a file has to be provided by the user) when creating the make file for the system. Such a template make file allows to include external code that should be compiled and linked with the generated C code. A relative path is relative to project directory, '+' signifies to use the `.tpm` file in the target directory that has the generated default name (dependant on build root name).

### Operators in environment header file

Type: Boolean

Default: False

This attribute controls whether the definition of UML operators should be present or not in the system interface header file created by the C Code Generator.

### Simulation kind

Type: SimulationKind

Default: Standard

This attribute controls the time handling for the Model Verifier. `SimulationKind` can be either:

```
Standard
Realtime
With Environment
```

`Standard` denotes a discrete simulation where the environment is controlled through the Model Verifier interface.

Given the value `Realtime` there will be a real-time clock timer delay in set timers. The delay will be one second per time unit. The macro XCLOCK is set at compile time.

`With Environment` will generate a Model Verifier with environment control also through the application API. The macro XENV will be set at compile time.

### Support C++

Type: Boolean

Default: False

This attribute controls if the code generated by the C Code Generator or AgileC Code Generator should be given properties that make it possible to compile it with C++ compilers, or if the code should be compiled using an ISO C compiler. The properties of major concern are in particular the handling of external code to be compiled and linked with the application code and run-time library.

This attribute also controls that a suitable library with the `_cpp` suffix will be used when compiling and linking the code.

### Suppress C level warnings

Type: Boolean

Default: False

This attribute controls if warning messages that are issued from the following tools should be suppressed. (Such warnings can be disregarded from in most practical cases).

- C Code Generator / AgileC Code Generator / Model Verifier
- C compiler
- C linker

### Target directory

Type: String

Default: (empty)

This attribute controls where the files generated by the C Code Generator are to be written on the file system. Both Absolute and relative path is accepted. If specified relative, the “root” is the location of the current project (`.ttp`) file.

This attribute has an empty default value, in which case the conventions for Target Directory naming and location take effect.

### Target kind

Type: TargetKind

Default: (depending on host)

This attribute controls what kind of target application will be built with the C Code Generator.

The possible values for `TargetKind` are either Win32, Solaris - cc, Solaris - gcc, Linux - gcc or Cygwin.

**Note**

*The Solaris - cc, Solaris - gcc and Linux - gcc Target kind are only supported on UNIX.*

**User defined kernel**

Type: String

Default: (empty)

This attribute defines if a user-defined run-time library should be used when compiling and linking the application, instead of using one of the provided libraries that are provided with the C Code Generator.

Such a library is identified by a directory, containing the files with definitions of C macros and compiler switches that govern how the code will be preprocessed and compiled.

**Verbose mode**

Type: Boolean

Default: False

This attribute controls if the C Code Generator should print informative reports, diagnostics and messages in the message output area, or not.

**See also**

"Supported libraries" on page 1062 in Chapter 33, *C and AgileC Runtime Libraries*.

"Library files" on page 1065 in Chapter 33, *C and AgileC Runtime Libraries*.

## C Application

This stereotype controls the generation of code when building a C application, using any of the code generators AgileC Code Generator, C Code Generator and Model Verifier.

The stereotype controls the following:

- Disabling the code generation for individual model elements.
- A flexible way for how to import external C or C++ declarations when generating C applications. Such C or C++ code could either be hand-written code or third party libraries.

### Generate C code

Type: Boolean

Default: True

This attribute controls whether an element in the UML model should be included or not in a build that uses the C Code Generator or Model Verifier build types.

The default behavior for such builds is to include all the classes and the packages that are contained in the class designated as Using Build Roots for a Build Artifact.

With this attribute you can decide to discard individual elements in the model, which allows for instance to exclude parts of the application that are not yet implemented, or that are not found to work properly.

The impact of discarding an element may result in semantically incorrect models, or inability to generate code.

### Include File

Type: String

Default: (empty)

This attribute defines that a file (typically a C or C++ header file) with external definitions should be included in the code generated for the element that this attribute applies to.

In the generated C code there will be created a C/C++ `#include` statement that includes the file with the external definitions.

**Hint**

*This String attribute designates one file at most. Should you need to include more than one header file, then you can create a header file that contains the #include statements for all the header files that are needed for the generated code to compile and link (and usually nothing else but these statements), and specify this file to be used as Include File in the Properties Editor.*

### Language

Type: langKind

Default: C

This attribute specifies which programming language is used for the external declaration. The available values for **langKind** are C and C++.

If C++ is the language used, then the attribute Support C++ for the C Code Generator stereotype (or Model Verifier, depending on the build type) should usually be set to True, for the generated C code to properly compile and link with external code, and/or for the generated executable to behave as expected.

### C name

Type: String

Default: (empty)

This attribute defines the name that the element will be given in the C code, overriding the naming scheme that is defined by the C Code Generator.

It also defines if, in the code that is generated by the C Code Generator, an element should be referred to using a given name, other than the name given in the UML model. This way, you may include external declarations, where names are given, without having to keep track of the naming scheme that is adopted by the C Code Generator.

**See also**

"Names in Generated C Code" on page 1121

"Target code expression" on page 362

## C Application Customization

This stereotype allows to customize advanced settings used for the generation of C code. The settings in this stereotype are shared by all the C build types (C Code Generator, AgileC Code Generator and Model Verifier), in order to preserve the behavior of the application that is used to debug the system on host and the application that is deployed to target.

### Priority

Type: Integer (`0..255`)

Default: (empty)

This attribute allows to specify priorities on signal definitions. The C Code Generator, AgileC Code Generator and Model Verifier have the ability to arrange the signal queue according to signal priority (in addition to the default which is to sort the queue according to arrival order).

Lower values mean higher priorities.

By default this attribute is empty, meaning that the code generators will add signals into the signal queue according to the signal arrival order.

## C++ Application Generator

This stereotype contains settings used for the generation of C++ code performed by the C++ Application Generator.

The attributes of this stereotype corresponds to the Translation Options of the C++ Application Generator.

## C++ header file

This stereotype specifies that the type definitions for the C++ code that is generated by the C++ Application Generator for the model element should be stored on a given file.

Files that manifest C++ code will be taken into account by the Makefile Generator and dependencies to these files will be present in the make file it generates.

### File name

Type: String

Default: (empty)

This attribute specifies the name of the header file on which the C++ Application Generator should store the C++ definitions generated for the model element. This name must be specified including the file extension (such as `.h`, `.hpp`).

### Precompiled

Type: Boolean

Default: False

This attribute indicates that the file that is specified exists in a precompiled version. This feature is only supported by C++ compilers on Windows.

## C++ implementation file

This stereotype specifies that the C++ code implementation code that is generated by the C++ Application Generator for the model element should be stored on a given file.

Files that manifest C++ code will be taken into account as sources by the Makefile Generator. Dependencies to these files will be present in the make file it generates.

### File name

This attribute specifies the name of the source file on which the C++ Application Generator should store the C++ code generated for the model element. This name must be specified including the file extension (such as `.c`, `.cpp`).

**See also**

"library" on page 997

## cppImportSpecification

The stereotype **cppImportSpecification** is applied on a UML package that is used as "target" for the model elements that are created as the result of C/C++ Import.

### Add source file references to enable navigation from the UML model to the C++ source

Type: Boolean

Default: True

With this attribute set to True, this option allows the user to select in the model view an element in an imported package and from the shortcut menu select **Go to source**. This will then open the source header file and navigate to the origin of the selected element.

**Always generate constant expressions within [[]]**

Type: Boolean

Default: False

When this attribute is set to True the importer will translate all constant C/C++ expressions to UML target code expressions ([[...]]). Otherwise it will try to translate expressions to UML expressions.

**C only**

Type: Boolean

Default: False

With this attribute, you can decide to execute import in C mode. In C mode, it is assumed that no C++ constructs are encountered in the definitions. If this assumption does not hold, that is if any C++ code is encountered in the input, the result of the translation is **undefined**.

When importing C and C++ header files, there should be one package assigned to store the elements resulting from each import pass, to ensure a separate handling of C and C++, and subsequently there should be separate Input header files and possibly Preprocessor settings.

**C/C++ dialect**

This area groups a set of boolean attributes, that can be individually turned to True or False, to enable the support for a given C/C++ dialect.

If no dialect at all is enabled, then the ISO C, or the C++ ISO/IEC 14882 standard is supported (depending on if the code is imported as C only or not).

- **GNU C/C++**

  Type: Boolean

  Default: False

  This attribute controls if the import should be compliant with the GNU C/C++ dialect.

- **Microsoft C/C++**

  Type: Boolean

  Default: False

  This attribute controls if the import should be compliant with the Microsoft Visual C/C++ dialect.

- **Borland C/C++**

  Type: Boolean

  Default: False

  This attribute controls if the import should be compliant with the Borland C/C++ dialect.

**Do not import definitions from included header files**

Type: Boolean

Default: True

This option is to avoid import of definitions from nested libraries that may result in a very large number of imported definitions.

If this option is set to False, all definitions from all referenced header files will be imported to the resulting package.

With this option set to True, definitions from included header files are not imported to the resulting package. This is irrespective if standard header files (`#include <header.h>`) or user header files (`#include "header.h"`) are used. If several header files should be imported, all of them must be listed among the input header files, even if they include each other. If you also use Selective import and a definition from an imported header in turn references another definition from another included header file, then this depending definition will be imported to the resulting model when the option Translation of depending declarations is set.

**Note**

*If standard header files are used, the number of imported definitions can become very big which may affect performance.*

**Generate artifacts**

Type: Boolean

Default: False

If this attribute is True the importer will generate file artifacts that represent the imported files. It will also generate a build artifact for the C++ Application Generator, which will manifest the file artifacts. The purpose of these artifacts is to facilitate the regeneration of the imported files using the C++ Application Generator, and this option is hence intended mainly when the purpose of the import is migration from C++ to UML. If the import is repeated the file artifacts will be updated to reflect the files that were imported, but the build artifact will not be touched. It is therefore safe to manually change options on this artifact to set-up the C++ code generation; these settings will not be overwritten in case of a repeated import.

### GUID algorithm

Type: `GuidStrategyKind`

Default: `Random`

This attribute controls whether GUID should be generated using a random pattern or should be named from the names defined in the C/C++ header files. Possible values for `GuidStrategyKind` is **Random** and **Name**.

### Action code strategy

Type: `ActionCodeStrategy`

Default: `DoNotImportAc` ("Don't Import Action Code")

If this attribute is set to `ImportAC` ("Import Action Code") all function bodies that are present in the imported files will be imported, using UML actions. This is typically used when the purpose of the import is to migrate a legacy C/C++ application to UML, and later regenerate it using the C++ Application Generator. But also otherwise it can be of interest to bring the function bodies to UML.

Sometimes it is more convenient to import function bodies using informal actions in the UML model. To do so set this attribute to `ImportACAsInformal` ("Import Action Code as Informal"). In this case the function body contents will be copied verbatim into the UML operation body as an informal action. This has the advantage of preserving code layout, macros etc. in the operation body which otherwise is lost during the import. However, it also means that the UML checker won't check the correctness of the operation body.

### Import char* as CPtr<char>

Type: Boolean

Default: False

This option is available in order to be able to import C/C++ `char*` as `CPtr<char>` rather than as `'char*'`.

Overriding the default behavior by turning this option to True will cause external `char*` definitions to be imported as `CPtr<char>` instead of `'char*'`. This may be useful if the model already contains character strings that have been imported using tool versions prior to 2.3.00.

### Import C++ pure virtual classes to UML interfaces

Type: Boolean

Default: True

This option allows to specify how to import C++ pure virtual classes:

- With this option set to True, C++ classes that contain only pure virtual methods, will be imported to UML interfaces
- With this option set to False, C++ pure virtual classes will be mapped to UML abstract classes

For more information see "Pure virtual member functions" on page 595.

### Import unsigned char to Octet

Type: Boolean

Default: False

With this option set to true, C++ type unsigned char will be mapped to UML type Octet. Such mapping is useful for the systems migrated from SDL Suite, since C++ type `unsigned char` is represented by the SDL type Octet, defined in the SDL Suite predefined packages for C++ support.

If this option is set to False, the C++ type `unsigned char` is mapped to the corresponding UML type "unsigned char".

### Import class pointers to UML references

Type: Boolean

Default: True

This setting defines if import of C++ pointers to classes should result into direct references to UML classes (a mapping that was introduced in version 2.3.00), or instead use the `CPtr<T>` template as was done in anterior versions.

This setting is mainly intended to provide backwards-compatibility with C++ import mapping schemes that were used in versions prior to 2.3.00, but at the expense of poorer functionality available after import (see Example 337 on page 987).

Consider the import of the following defined in C++:

```
class C {}
class D {
  C* c;
}
void op (C* par);
```

By default the import will generate the following UML definitions:

```
class C {};
class D {
  C c;
}
void op (C par);
```

By setting this attribute to False, the following UML would be generated instead:

```
class C {};
class D {
  CPtr<C> c;
}
void op (CPtr<C> par);
```

**Example 337: Visualizing association between imported classes**

Consider the following C++ definitions:

```
class a {
public:
  int x;
};
```

```
class b {
public:
 a* a1;
};

int Getx( b* pb );
```

Importing these two classes will result into two UML classes and one operation. If these definitions are dragged to a class diagram, association lines between classes b and a, and between operation Getx and class b will be drawn automatically as in Figure 231 on page 988.

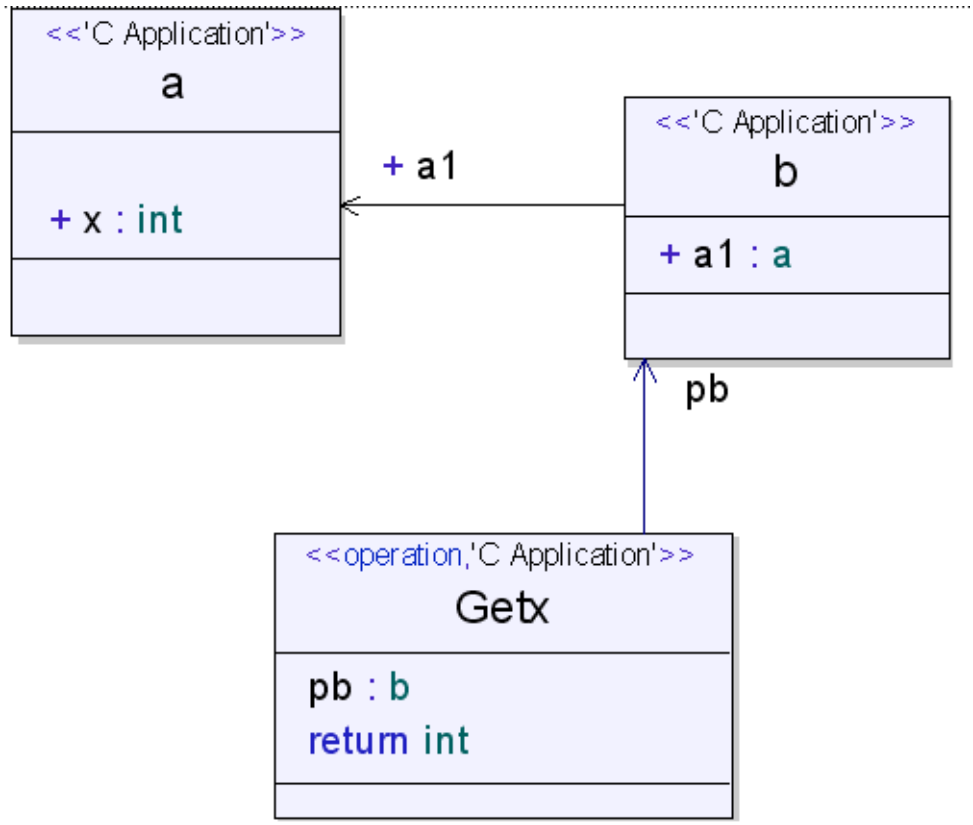



*Figure 231:*

### Input header files

Type: List of Charstring

Default: ““

This setting defines a list of input files (that is C/C++ headers) that contain the declarations that are to be processed during import and placed in the Output header file and ultimately become translated to UML.

There is a button on the toolbar that allows to insert items to the list of files to import. For each click of the **New / Insert** button, you may either

- Type in the name of the header file of your choice, or
- Click the **Browse** button that appears to the right of the newly created item, and designate the file in the subsequent **Open** dialog.

Files that should no longer be imported, are removed from the list with the **Delete** button.

The list of files is passed as a list of `#include <filename>` to the Preprocessor currently defined. Depending on technical limitations in the preprocessor, or in the C/C++ Import, conflicts resulting from for example illegal re-declarations may cause the import to fail in whole or parts. The **Move Up** and **Move Down** buttons allow you to rearrange the order in which files are imported. The importing scheme follows a top-down order, and, if the order in which definitions are imported is of significance, you may need to rearrange the list for the import to work properly.

The list of header files may also require appropriate values for the settings C only and Preprocessor.

### Options

Type: `Charstring`

Default: ""

This text box specifies options that should be passed to the Preprocessor. Typically such options could be preprocessor macros that are expanded to given values.

**Example 338: Preprocessor options**

```
/Dmymacro1/Dmymacro2
```

The syntax for defining a macro, that is `/D` in the example, depends on the actual preprocessor used (with the Preprocessor attribute).

### Output header file

Type: Charstring

Default: ""

This setting defines the output header file produced during C/C++ Import. It will consist of a `#include` statement for each of the Input header files specified to import. This output header file is the file that will be parsed and translated into UML.

Since the output header file should be included in the final C code generated by the C Code Generator, it is usually a good idea to place it in the build Target Directory. If no code generation is going to be made, the Output header file attribute can be left unspecified. (In such a case a temporary output file will be created by the tool).

### Preprocessor

Type: `Charstring`

Default: ""(An empty value is interpreted during import as "`cl`" on Windows or "`cpp`" on UNIX.)

This text field allows to specify which Preprocessor to use prior to importing the header files. The text field would typically contain the command used to invoke the preprocessor, or a suitable script that "embeds" the actual preprocessor used.

Such a command or script can be typed in directly, or be specified by clicking the **Browse** button and using the **Open** dialog to locate the file.

During C/C++ import the following preprocessors are recognized (by name):

- cl (Microsoft Visual C/C++ preprocessor) – Windows only
- cpp32 (Borland C/C++ preprocessor) – Windows only
- cpp – UNIX only
- g++/gcc (GNU C/C++ preprocessor) – UNIX (including Cygwin)

If the preprocessor you want to use is not among the ones recognized on your platform, you could write a shell script which calls the preprocessor. Be sure to give the script a name different to the names listed above. The C/C++ import will call the script with two parameters:

1. The name of the input header to preprocess.
2. The name of the file where the result of preprocessing should be stored.

**Example 3391: Shell script for gcc on Windows**

This script file calls the gcc preprocessor (from cygwin). The preprocessor is specified by `mycpp.bat` in the cppImportSpecification for the package ImportedDefinitions.

Click on the package ImportedDefinitions, choose "Import" – `mycpp.bat` is called. The log in the output tab shows:

```
gcc -E -v -I . -x c++ %1 -o %2.i
@copy %2.i %2
```

**Example 340: Shell script for user-defined preprocessor on UNIX**

An example script for UNIX platforms, `mycpp.sh`, is shown below.

```
gcc -E -v -I . -x c++ $1 -o $2.i
cp $2.i $2
```

### Selective import

Type: List of Charstring

Default: ""

This edit list may contain a list of an arbitrary number of C/C++ identifiers. These identifiers will be made available in the model by translation of the corresponding declarations.

By default this list is empty that means all C/C++ declarations that are parsed will be imported into the model. If the same identifier is entered multiple times, it will be imported only once. In the case od an selective import you should also do a Translation of depending declarations to guarantee that the resulting set of UML declarations is complete and congruent.

### Set External attributes for imported definitions

Type: Boolean

Default: True

With this attribute set to true, the external attribute will be set for imported definitions.

This option should be set to False when the intention is not to reuse the imported definitions.

### Translation of depending declarations

Type: Boolean

Default: True

With this attribute set to true, if an identifier in an import specification refers to a declaration that in turn depends on other declarations, then all these depending declarations are translated as well. This principle is applied recursively to all declarations that depend on depending declarations, thereby making sure that the resulting UML declarations are complete and congruent.

With this attribute set to false, depending declarations will not be translated automatically. In such a case, the tool cannot guarantee that the resulting set of UML declarations is complete and congruent. This is especially important when you do a Selective import. If the option "Do not import definitions from included header files" is true (default), you must ensure that you list all header files that are necessary for all depending declarations.

### Import only exported definitions

Type: Boolean

Default: False

With this attribute set to true, the C/C++ Importer will only import those definitions that are marked with the `__declspec(dllexport)` keyword. Use of this option requires a C/C++ compiler with support for the `__declspec` keyword.

## Java

This stereotype is used for defining a Java build artifact, used in order to generate and roundtrip Java source code from UML.

The attributes of this stereotype corresponds to the Java Build Artifact Settings of the Java code generator.

## Configuration

This stereotype contains settings controlling which configuration(s) a Build Artifact belongs to.

**Note**

*A build artifact may have multiple configuration stereotypes attached, one for each configuration it is contained in. This is visualized by multiple occurrences displayed in the Filter drop-down menu in the Properties Editor.*

**name**

Type: String

Default: (empty)

This attribute contains the name of the Configuration that the build artifact belongs to.

## dynamicLibrary

This stereotype inherits from the parent stereotype «library >> and is used in order to specialize a file artifact so that it manifests a dynamically linked library.

This stereotype is suitable to apply to UML packages.

File artifacts using the dynamicLibrary stereotype are managed by the Makefile Generator so that dependencies are established to the C++ source files files manifested in the UML package that the stereotype is added to.

The C++ sources that are linked to become the following:

| Platform | Resulting file |
|---|---|
| Windows | `<File name>.DLL` (Windows application extension) |
| UNIX | `lib<xxx>.so` (shared object) |

**Note**

*This stereotype is only used by the models that are built using the C++ Application Generator.*

**File name**

Type: String

Default: (empty)

This attribute designates the base name of the file that implements the dynamic library specified by the file artifact.

## executable

This stereotype is used in order to specialize a file artifact to have it manifest an executing application. It is suitable to add it to model elements that define a scope that contain all definitions and implementations required for a successful compile and link – typically a top level class.

File artifacts using the executable stereotype are managed by the Makefile Generator so that dependencies are established to the C++ source files and libraries manifested in scope defined by the element that the stereotype is added to.

The C++ objects and libraries are linked to become the following:

| Platform | Resulting file |
|---|---|
| Windows | `<File name>.EXE` (Windows application) |
| UNIX | `<File name>` (ELF Executable Library Format) |

**Note**

*This stereotype is only used by the models that are built using the C++ Application Generator.*

## file

This stereotype extends the Metaclass **Artifact** and specializes an artifact to become a File Artifact. It is a base stereotype, marked as hidden in its profile and has no practical use in its basic form.

Instead, the following specialized file stereotypes are available for practical use when modeling and building applications:

- «C++ header file»
- «C++ implementation file»
- «dynamicLibrary»
- «executable»
- «staticLibrary»

**Note**

*These file artifacts are not used when models are built using the C Code Generator, AgileC Code Generator or Model Verifier.*

### File name

Type: String

Default: (empty)

This attribute designates the base name of the file that manifests the UML element that the stereotype is applied on.

## Icon

This stereotype contains attributes that allow to tailor the appearance of graphical elements in the workspace window and in the editors.

### 16 x 16 Pixels Bitmap file

Type: String

Default: (model element dependent)

This attribute specifies which appearance is given to elements when displayed in the workspace window. It is typically taken advantage of by build artifacts and file artifacts (elements that have the «file» or «build» stereotype added, to give them an appearance that is easily associated with its function.

The string should specify a file containing a suitable 16 x 16 pixel bitmap or icon. Default values refer to files stored in the `etc` directory of the Tau installation.

The value of the attribute has no semantic impact on the model or how it is built. The impact is only 'cosmetic'.

### Icon File

Type: String

Default: (empty)

This attribute specifies which appearance is given to symbols when displayed in the editor window. It is supported by a subset of the 'important' UML symbols, such as Package, Class, Part and State.

The string should specify a file containing a suitable bitmap or icon file.

If the attribute is empty, the symbol is displayed using the factory settings – UML symbols are visualized according to the standards set by OMG.

This attribute has no semantic meaning, the effect is 'eye-candy'.

### KeepIconProportions

Type: Boolean

Default: False

With this attribute set to true, the height - width ratio is the same as in the original image.

**See also**

"Icon" on page 187 in Chapter 7, *Working with Diagrams*

## LabelPosition

### labelVertPosition

Type: {TopOutside | TopInside | VCenter | BottomInside | BottomOutside}

Default: (empty)

This attribute allows to specify how text labels for symbols should be placed vertically.

**labelHorzPosition**

Type: {LeftOutside | LeftInside | HCenter | RightInside | RightOutside}

Default: (empty)

This attribute allows to specify how text labels for symbols should be placed horizontally.

## jarFile

This stereotype is used by UML models that are managed and built using the Java technology supported by Tau. It is not used when building applications that use the C or C++ technology.

**See also**

"Java Files" on page 1336 in Chapter 42, *Java Support*

## javaFile

This stereotype is used by UML models that are managed and built using the Java technology supported by Tau. It is not used when building applications that use the C or C++ technology.

**See also**

"Java Files" on page 1336 in Chapter 42, *Java Support*

## library

This stereotype is used in order to specialize a file artifact so that it specifies a library. The concept of library in this case corresponds to a 'generic object file component' suitable to manage in an atomic way and that can be input to the linker. It does not however specify any details concerning the library properties or details about its technology.

This stereotype is suitable to apply to UML packages.

File artifacts using the library stereotype are managed by the Makefile Generator in such a way that make dependencies are defined in the generated make file, between the source files implementing the model elements and the file specified by the file artifact.

Instead of the stereotype library, the following stereotypes can be used once the library properties are better known, and the preferred library technology choice is decided:

- «dynamicLibrary»
- «staticLibrary»

**Note**

*This stereotype is only used by the models that are built using the C++ Application Generator.*

**File name**

Type: String

Default: (empty)

This attribute designates which file implements the library specified by the file artifact.

**See also**

"C++ implementation file" on page 982

## makefile

This stereotype is used internally by the tool in order to manage the generation of make files in a safe and correct way.

It is listed for the sake of completeness but is not intended to be used by the user.

Although using this stereotype may result in make files produced by the Makefile Generator that work in practice, this could be the result of 'pure luck'. Therefore this stereotype should not be used by the user, unless explicitly instructed to do so by a trusted Telelogic source.

## Make settings

This stereotype contains the settings used when executing the “make” utility. The “make” utility is run as a result of ordering a build command that involves the compilation of the generated code (e.g. Build Configuration).

The make utility can execute either:

- Transparently, operating on the make file that is generated by the Makefile Generator
- Explicitly on a make file provided by the user.

### Command

Type: String

Default: (empty)

This attribute specifies the actual “make” command to be performed instead of the default command.

### Dialect

Type: {`DEFAULT | gmake | nmake | sunmake`}

Default: DEFAULT

This attribute allows to specify which dialect of “make” is used when submitting the make file to the “make” utility, and ensuring that file paths and options are specified in the proper way.

- DEFAULT assumes that make is called ‘native’ according to the host operating system.
  - nmake for Windows hosts (using backslashes in file paths, and slashes for options)
  - gmake for Linux hosts (using slashes in file paths, and dashes for options)
  - sunmake for Solaris hosts (using slashes in file paths, and dashes for options)
- gmake and sunmake specify to use UNIX conventions when calling “make”
- nmake specifies to use Windows conventions when calling “make”

### Makefile

Type: String

Default: (empty)

With this attribute can be specified a particular make file to use as input instead of the make file that is generated by the Makefile Generator.

### Options

Type: String (multi-line)

Default: (empty)

This attribute specifies an arbitrary number of user defined options, to be used when invoking the “make” utility.

## Makefile generator

This stereotype inherits from the base stereotype «build» and contains the settings that control the contents and location of the generated make file. The generated make file is produced by the Makefile Generator as a result of ordering a build command involving code and make file generation (e.g. Generate Configuration).

### Dialect

Type: {Default | gmake | nmake | sunmake}

Default: DEFAULT

This attribute allows to override the dialect that is used when generating the contents of the make file, which may be necessary if the generated application should be easily compiled on another host computer than the one hosting Tau.

- DEFAULT will have the tool use the make dialect that is ‘native’ on the host operating system.
  - nmake for Windows hosts
  - gmake for Linux hosts
  - sunmake for Solaris hosts

- gmake instructs the tool to use the make dialect defined by GNU "gmake" utility, and the GNU C/C++ tool chain. This should be used on Linux and Solaris hosts.
- sunmake instructs the tool to use the make dialect defined by SUN workshop, and the forte C++ tool chain.
- nmake instructs the tool to use the dialect used by Microsoft "nmake"

**Target directory**

Type: String

Default: (empty)

This attribute controls where the make file generated by the Makefile Generator is to be written on the file system. Both absolute and relative path is accepted. If specified relative, the "root" is the location of the current project (`.ttp`) file.

This attribute has an empty default value, in which case the conventions for Target Directory naming and location take effect.

**User code**

Type: String (multi-line)

Default: (empty)

This attribute specifies user code to be inserted to the generated make file by the Makefile Generator. This feature allows to customize the contents of the generated make file to the user's convenience and is provided mainly for openness.

The user provided section is inserted between the "compiler macro definitions" section and the "dependencies" section in the generated make file.

## Model Verifier

This stereotype inherits from the base stereotype «build» and contains the attributes that control the generation of a Model Verifier – an application that is instrumented to support simulation, tracing and also detailed debugging of the application at UML level.

### Expand macros

Type: Boolean

Default: False

This attribute controls if macros in C code should be expanded and processed after code generation, to make the C code more readable, and make it easier for you to take advantage of a C debugger while running the Model Verifier. This processing is done by the C Compiler Driver utility.

**Note**

*Expand macros is not supported with Target kind set to Win32.*

### Generate environment template functions

Type: Boolean

Default: False

This attribute controls if files with skeletons for the Environment Functions should be created when building the Model Verifier.

The System interface header file (`.ifc`), holds an up-to-date definition of the interface to the environment is always created regardless of the value of this attribute.

### Make template file

Type: String

Default: (empty)

This attribute controls if the tool should use a given make template file (that has to be provided by you) when creating the make file used for the Model Verifier. Such a template make file allows to include external code that should be compiled and linked with the generated C code. A relative path is relative to project directory, '+' signifies to use the `.tpm` file in the target directory that has the generated default name (dependant on build root name).

### Suppress C level warnings

This attribute is described in the section about the C Code Generator stereotype.

**Support C++**

Type: Boolean

Default: False

This attribute controls if the Model Verifier that is built should be given properties that make it possible to compile it with C++ compilers, or if the code should be compiled using an ISO C compiler. The properties of major concern are in particular the handling of external code to be compiled and linked with the application code and run-time library.

This attribute also controls that a suitable library with the `_cpp` suffix will be used when compiling and linking the code.

**Target directory**

Type: String

Default: (empty)

This attribute controls where the files, generated as the result of building using the current Build Artifact, are to be written on the file system. If the location is specified as a relative path, the "root" is the location of the current project (`.ttp`) file.

This attribute has an empty default value, in which case the conventions for Target Directory naming and location take effect.

**Target kind**

Type: `TargetKind`

Default: Win32

This attribute controls what kind of target application will be built with the C Code Generator.

The possible value for `TargetKind` is set by a drop-down menu. It can be either Win32, Win32-gcc, Solaris-cc, Solaris-gcc or Linux-gcc.

**Note**

*1. The Solaris and Linux values are only supported for the UNIX version.*
*2. The value Win32-gcc is meaningful only if you have installed the GNU C compiler according to the Installation Guide. Furthermore, Win32-gcc does only support Model Verifier without any environment functions.*

**Verbose mode**

Type: Boolean

Default: False

This attribute controls if the C Code Generator should print exhaustive reports and diagnostics in the message output area when generating code for a Model Verifier.

**See also**

"Generate environment template functions" on page 975 in Chapter 28, *Stereotypes for Code Generation*

"Make template file" on page 975 in Chapter 28, *Stereotypes for Code Generation*

"Supported libraries" on page 1062 in Chapter 33, *C and AgileC Runtime Libraries*

## objectFile

This stereotype is used internally by the tool in order to manage "make" dependencies between source files and the object files generated by the C++ compiler. It is also used internally to define dependencies to targets (executables or libraries)

It is listed for the sake of completeness but is not intended to be used by the user.

Although using this stereotype may result in make files produced by the Makefile Generator that work in practice, this could be the result of 'pure luck'. Therefore this stereotype should not be used by the user, unless explicitly instructed to do so by a trusted Telelogic source.

## Source reference

The stereotype **Source reference** contains attributes that hold information about which definition an element is imported from.

The main application area is to allow a navigation to the originating C/C++ definition, by selecting **Go to C/C++ source** on the shortcut menu. Your preferred text editor for working on C/C++ source and header files will open a window on the file that is specified by the attribute File.

If technically possible and supported by the text editor, the text insertion cursor will be positioned on the line and column defined by the attributes Line and Column.

**Note**

*The attributes are given their values during C/C++ Import. These values should not be modified by the user, or the references will become erroneous.*

### File

Type: Charstring

Default: (empty)

This attribute tells which C/C++ header file contains the definition of an element that is created as the result of an import. The attribute is empty for elements that have not been created as the result of an import.

### Line

Type: Integer

Default: (empty)

This attribute refines the information about the originating C/C++ definition from which an element is imported, by telling on which line in the C/C++ header File the declaration of the definition is found during import. The start count for lines (usually 0 or 1) is depending on the Preprocessor.

The attribute is empty for elements that have not been created as the result of an import.

### Column

Type: Integer

Default: (empty)

This attribute refines the information about the originating C/C++ definition from which an element is imported, by telling on which column in the C/C++ header File the declaration of the definition is found during import. The start count for columns (usually 0 or 1) is depending on the Preprocessor used.

The attribute is empty for elements that have not been created as the result of an import.

## staticLibrary

This stereotype inherits from the parent stereotype «library» and is used in order to specialize a file artifact so that it manifests a statically linked library.

This stereotype is suitable to apply to UML packages.

File artifacts using the staticLibrary stereotype are managed by the Makefile Generator so that dependencies are established to the C++ source files manifested in the UML package that the stereotype is added to.

The C++ sources that are linked to become the following:

| Platform | Resulting file |
|---|---|
| Windows | `<File name>.LIB` (library) |
| UNIX | `lib<xxx>.a` (archive) |

**Note**

*This stereotype is only used by the models that are built using the C++ Application Generator.*

### File name

Type: String

Default: (empty)

This attribute designates the base name of the file that implements the static library or archive specified by the file artifact.

## thread

This stereotype extends the Metaclass **Artifact** and is used to specialize an artifact into a **thread artifact**. It contains attributes that allow to specify how an application should execute in separate threads.

### Instances

Type: String (multi-line)

Default: (empty)

This attribute contains a list of instances that should execute in a thread of their own. Each element in the list should be a UML fully qualified instance name.

### One thread per instance

Type: Boolean

Default: False

This attribute specifies if each instance mapped to the «thread» stereotype should execute in a separate run-time thread, or if all instances mapped to the thread stereotype should execute in the same run-time thread.

Setting this attribute to True has the same effect as creating a «thread» stereotype for each instance.

### Priority

Type: Integer

Default: (empty)

This attribute specifies which priority should be applied to the thread at thread creation. The value of the attribute is present verbatim in the generated code, in a parameter to the call of the operating system primitive that creates a thread.

If the attribute is left empty, the thread is created with default priority.

### Stack size

Type: Integer

Default: (empty)

This attribute specifies which stack size should be applied to the thread at thread creation. The value of the attribute is present verbatim in the generated code, in a parameter to the call of the operating system primitive that creates a thread.

If the attribute is left empty, the thread is created with default stack size.

**User 1, User 2**

Type: Integer

Default: (empty)

These attributes allow to define up to two integer parameters that are passed verbatim to the underlying operating system when creating a thread.

If any of these attribute is left empty, the call of the thread creation primitive uses a default value for the corresponding parameter.

# 29

# *Guidelines for Large-Scale Application Development*

This document discusses how to take advantage of the features in Tau intended for projects that create, manipulate and generate applications originating from large models. Special emphasis is given to projects that use the C Code Generator.

For a reference to the tool features, see Chapter 27, Building Applications Reference.

# Introduction

Some problem areas when designing large applications are the following:

- Dividing the model into appropriate files to enable efficient configuration and version management
- Creating a suitable model structure to minimize build times when generating code from the model.

Both of these aspects of application design are treated in this document.

# Library Builds

When developing a very large application a common strategy is to divide the application into different modules that are handled by different teams/persons and that have decently stable interfaces to each other. The idea in this situation is that each team only should care about the module it is responsible for and the interfaces to other modules. The implementation of the other modules is taken care of by other teams that should deliver libraries that correspond to stable builds of the other modules.

This approach has several benefits:

- Clearly defined responsibilities for each teams
- Clearly defined interfaces between the teams
- Reduced build times since each team only needs to build its own module.

The general scheme when working using library builds is thus that the application is divided into a set of modules, each one with a well-defined interface and each one designed to be compiled into one library. When working with a specific module only the current module is modified and only the library that is produced from this module is built.

Typically there also exists one 'main' module that defines the initialization of the application, using for example a `main()` function or by other suitable means. This module contains the code that is executed when the application starts. For each module there might also exist a test framework, that essentially is a 'main' module created for testing purposes only.

## Library artifacts

This scheme is supported in UML based on the concept of **Library Build Artifacts**. From a UML point of view a Build Artifact is an artifact that is stereotyped with a stereotype inheriting from the «build» stereotype. A typical example would be a «'C Advanced Application'», «'Model Verifier'» or «'C++ Application'» artifact.

A **library build artifact** is a Build Artifact with the 'Target' property set to 'Library' instead of 'Executable'. This means that the target file that is produced when building the artifact not is an executable file, but rather a library that should be linked together with other libraries to form an executable.

**Note**

*All library build artifacts must be of the same type. You cannot mix different build types within one project.*

A library artifact should «manifest» the model elements that should be included in the library. At present time, the tool supports only the case when the library contains all elements **in one specific package**. This is represented by having a «manifest» dependency from the library build artifact to the package itself. This is also the only supported way to create libraries when using the C Code Generator.

**Note**

*The packages that are manifested by library artifacts must be top-level packages.*

When building based on the library build artifact only the library is produced. To create a complete executable the 'main' module need to be built. In UML a 'main' module is represented by a build artifact with 'target' set to 'executable'. To identify the class that should be created when starting the application you use a «manifest» dependency from the build artifact to this selected class.

You also need to specify «include» dependencies between the build artifacts to give the tool information on what relationships exists between the libraries.

**Note**

*At present time, the «include» dependencies must be manually derived from the model and explicitly added where needed.*

The following illustrates the described concepts. In the chosen example is an application composed of two main libraries (called `Pkg1` and `Pkg2`), each containing one package with a set of definitions. The model also has one library (called `Interfaces`) containing the common interfaces, signals etc., and lastly a main module (called `LibraryExample`).

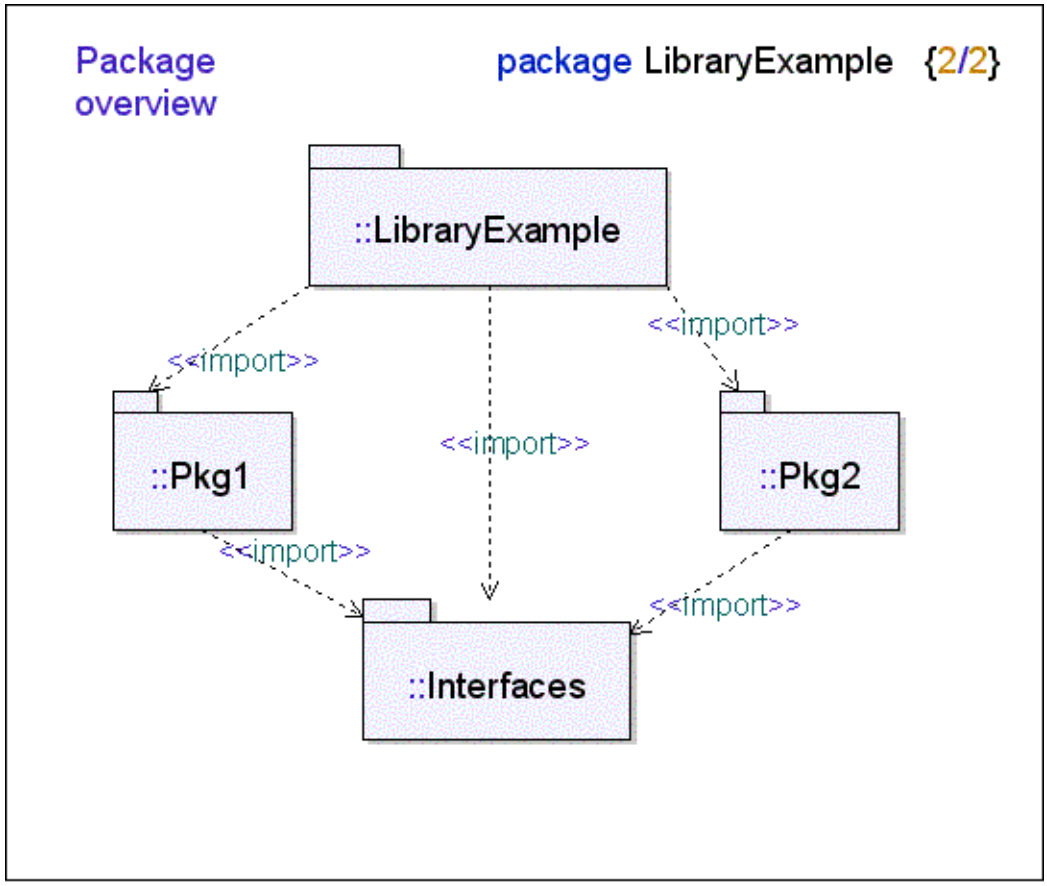



*Figure 232: Package overview of the example*

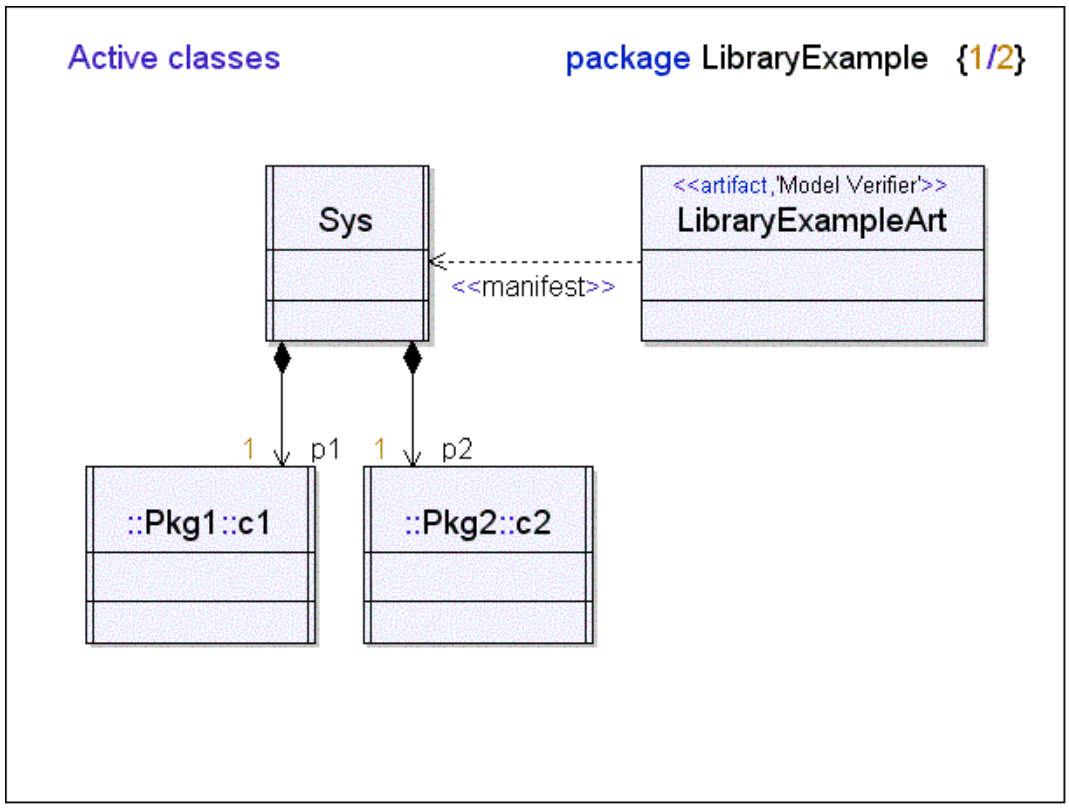



*Figure 233: System overview*

The `Interfaces` package is defined as follows:

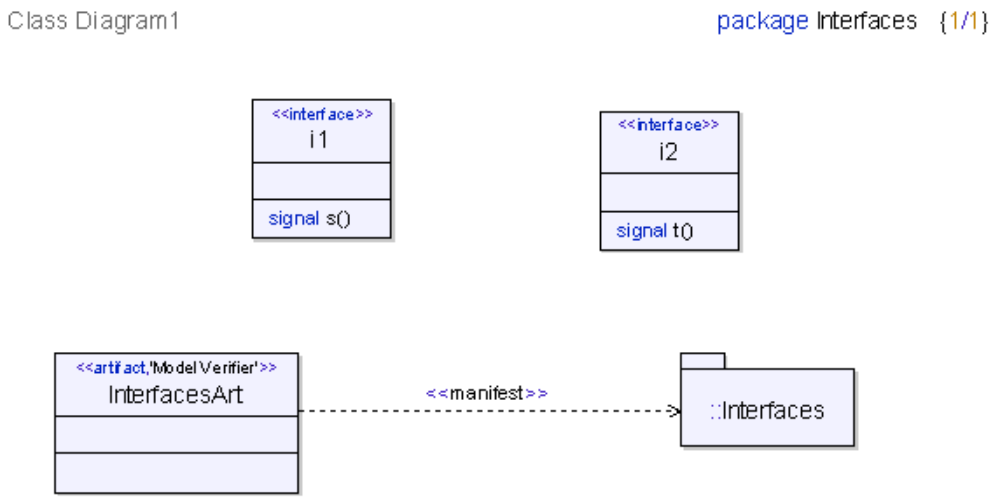



*Figure 234: Contents of package Interfaces.*

There are two interfaces, i1 and i2. There is also a **Model Verifier** Build Artifact `Interfaces` that manifests the package. If you were to check the properties of this artifact you would see that it has the Target property set to Library.

Pkg1 is defined as shown in Figure 235 on page 1014.

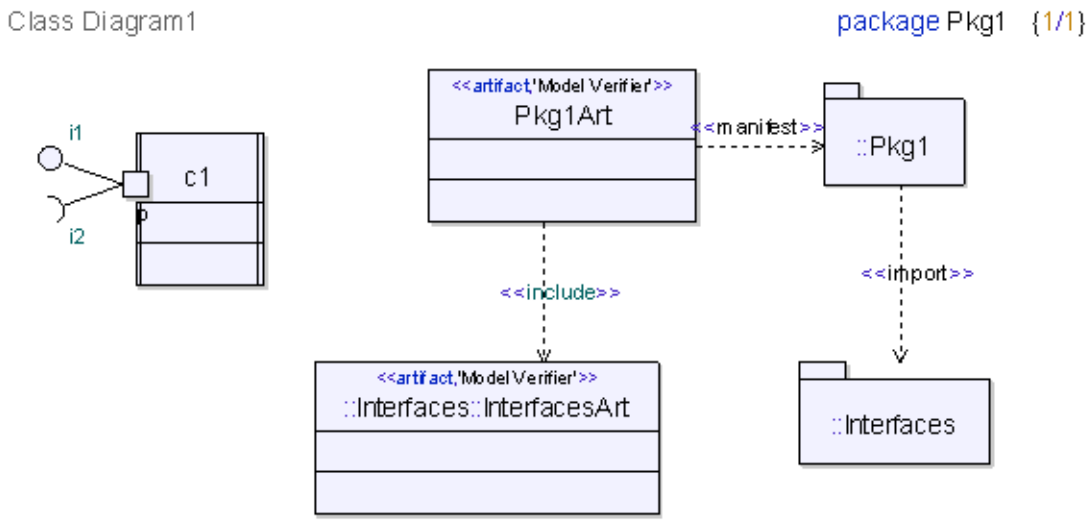



*Figure 235: Contents of package Pkg1*

`Pkg1` contains one active class called c1 and a «'`Model Verifier`'» build artifact `Pkg1Art`. The c1 class realizes the `i1` interface and requires the `i2` interface. Since these interfaces are defined in the `Interfaces` package above there should be an «`import`» dependency from Pkg1 to `Interfaces` to access their definitions. There should also be an «`include`» dependency from `Pkg1Art` to `InterfacesArt` to show that the `Pkg1Art` depends on the `InterfacesArt`.

`Pkg2` is very similar to `Pkg1`, with the difference that it contains another class as shown in Figure 236 on page 1015.

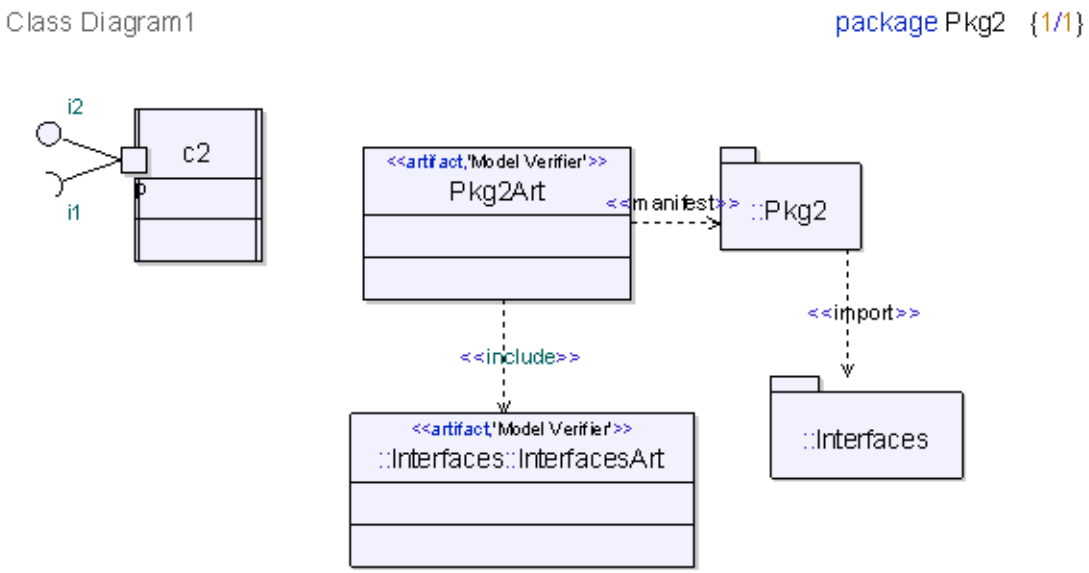



*Figure 236: Contents of package Pkg2*

In Figure 237 on page 1015 is shown the dependencies between the `LibraryBuildExample` artifact and the library artifacts it is depending on:

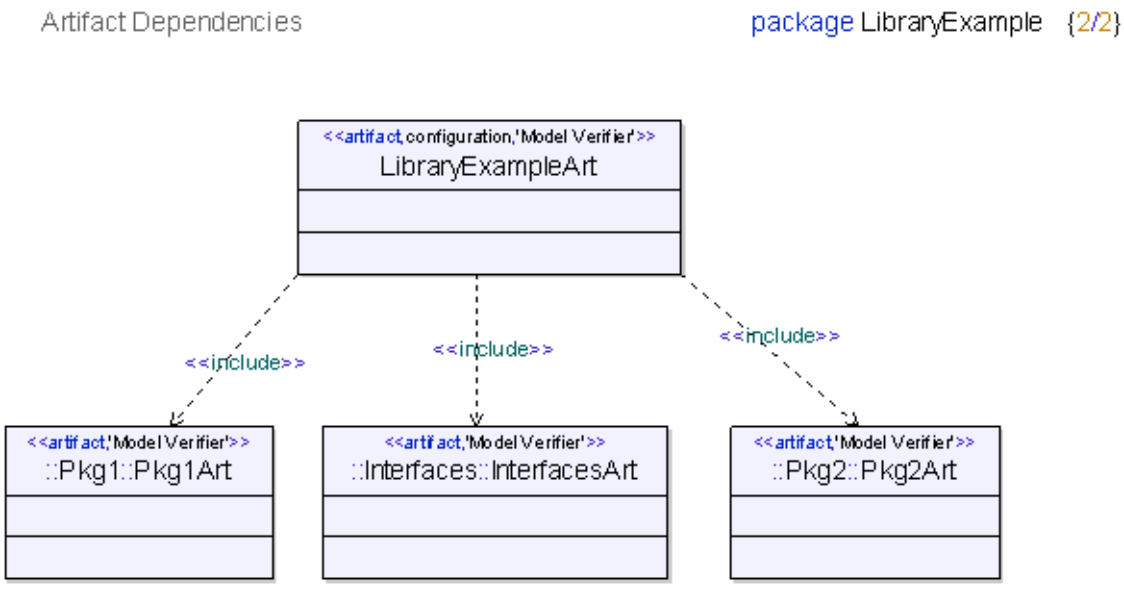



*Figure 237: Artifact dependencies*

Some aspects worth noting in this example are:

- The usage of the '`Interfaces`' package as a library module that includes the common definitions needed for the modules that contain active classes.
- Typically all of the packages would be stored in separate files. So there would be four UML files: One for each of the packages `Interfaces`, `Pkg1`, `Pkg2` and `LibraryExample`.
- There should be no library build artifact for the package in which '`main`' is placed.
- The same Target Directory should be used for all build artifacts. See "The build process" on page 1017.

## Implementation vs. signature files

To benefit from the library concept it is essential to have a well-defined interface between the modules and also to be able to separate this interface from the implementation of the module. In UML a module interface is described by a package that contains the type definition of a set of classes together with a set of interfaces, signals, data types etc. that are used in the public parts of the class definitions. In practice it is useful to store this kind of package in a separate file to have the version management of the package independent of the version management for the remaining parts of the model. In Tau it is possible to store for example a package in a separate file, by using the command Save in New File available on the shortcut menu in the Model View.

The second part of the solution is to be able to store the implementation aspects of the classes defined in the package in one or more separate files which versions can be handled independently of the file containing the module interface. In practice 'implementation aspects' consist of two parts:

- State machines and operation bodies that define the behavior of operations
- Private attributes.

In Tau you can move a state machine implementation or operation body to a separate file by dragging the body in the model view to one of the files shown in the Files folder in the Model View. Alternatively you can choose the Save in New File command for the implementation in the Model View similarly as for packages.

For the example in the previous section the natural implication would be to store the state machine implementations of `c1` and `c2` in separate files.

## The build process

To build the complete application you need to first build each of the libraries and then finally build the main module. But, before you do this, you should make sure to set up the Target Directory property of the different build artifacts to a common directory, for example "`mytargetdir`". This will ensure that the C compiler will be able to find the libraries during the linking phase.

For the example above you would give four build commands, first to build libraries for the Interfaces, `Pkg1` and `Pkg2` modules and finally to build the executable as described by the `LibraryExampleArt` artifact.

**Note**

*The libraries must be build in a 'bottom-up' order, i.e. before a specific package is built, all packages it depends on must be built.*

## Restrictions in the C code

The library based build has many advantages but there are some restrictions that are necessary to understand to be able to take advantage of the library build supported by C Code Generator. These restrictions are described below.

### Container types and library builds

When generating code for an entity, for example a class, the C Code Generator also generates a number of utility types for the class. Some examples include code for pointers to the class and code corresponding to an implicit collection type for the class used wherever the model contains an attribute typed by the class that has multiplicity >1. Other examples include types used wherever a template container type is instantiated with the class as a parameter, for example when a Bag or Array of the class is used as the type of an attribute.

If library builds are not used, the code generation includes a global analysis of the way the class is used and if template instantiations are found the corresponding types are generated in C code. However, if library builds are used

it can not be made sure that all possible instantiations are known when generating code for the library module. So, it is not possible to perform a global analysis of how the class is used.

Instead code is generated only for the following cases:

- Pointers to the class
- String instantiations.

The implication is that it is possible to use the class as the type of for example an attribute both with multiplicity 1 and a multiplicity >1. It is also possible to use a String of the class. However, all other attempts to use a template collection type instantiation with the class as parameter will generate an error since there will be no C code generated for the corresponding C type.

Fortunately there is a simple way to force the tool to generate the C code that corresponds to a particular template type instantiation. You can do this by simply defining a syntype that contains the template instantiation you want and that is stereotyped by «`CollectionType`». So, for example if you define a class C and want to use a `Bag<C>` as the type of for example an attribute then you must define a syntype definition "«`CollectionType`» `syntype Bag_C = Bag<C>;`" somewhere in his model. The most natural place to define this syntype is usually at the same location as where the class C is defined but any location in the model that is visible from where the `Bag<C>` definitions are used is also suitable.

The collection type restriction also applies to datatypes, even though the discussion in this section has been about classes.

### Recursive package import/access

The C Code Generator has a restriction with respect to how «import» and «access» dependencies can be used: It is not allowed to have recursive «import»/«access» dependencies. This applies both to a situation where library builds are used and when they are not used.

### Library packages and inheritance of active classes

If you have a library package with an active class and would like to use this active class for example in another library package, the class can only be instantiated. It is not possible to define another class that inherits the original. So all active classes that inherit each other must be defined in the same library package.

The reason for this is that, in order to generate code for inheritance, it is necessary to know the details on the states and transitions of the inherited class and this is typically hidden in the implementation part of a class.

# Managing File Size Using <<noScope>> Packages

An important aspect when developing a large model is to be able to support a large team that can concurrently work on the same model. The general strategy used in Tau to support this situation is to store the model in more than one file and then have different team members work on different files. The most common unit that is used for file splitting is to have packages in separate files and to have implementations of for example classes in separate files. However, in some cases it is useful to have other mechanisms, most often to make it possible to divide the model into more files. This section describes a method for how to accomplish this based on <<noScope>> Packages.

**Note**

*Since a «noScope» package does not form a scope unit, it is not allowed to have «import» or «access» dependencies to or from «noScope» packages.*

# Improving Build Performance

## <<bindByGuid>> packages

When generating code based on a model, the code generation is composed of several phases. One of the main phases is the name resolution phase, during which all references in the model are resolved. Tau can do this resolution in two modes, based on the unique identifiers that are generated for the model elements or based on the name and qualifiers of the elements and their position in the scope hierarchy. Normally Tau will use both mechanisms when resolving a name to be able to detect inconsistencies and propagate name changes. However, this will take extra time when loading the model and, in particularly for library packages that are seldom changed, it is not necessary.

If you apply the «`bindByGuid`» stereotype to a package you will force the tool to avoid the name based resolution for the entities in a package and only rely on resolution based on a unique identifier. This will not have any impact on what happens when interactively working with the elements in the package, name change propagation and similar features will still work. It will only affect what happens when loading the package from a file, however since this is exactly what you are doing when you generate C code for a model it can in some situations be an efficient way to improve the performance of the C code generation.

# 30

# *Requirement Traceability in Generated Code*

The C, C++ and Java code generators of Tau support the annotation of generated code with references to DOORS requirements. This is a means to obtain traceability from generated C, C++ or Java definitions to requirements to which these definitions are related.

This document describes how to utilize this feature.

# Introduction

When developing complex applications requirements management and analysis is often a key to success. If the requirements are vaguely specified or poorly analyzed the risks are high that the application that shall meet the requirements will not do so.

A requirement management and analysis tool such as Telelogic DOORS greatly helps in the requirement analysis phase of a project. When moving into the design phase where Tau is used for modeling and implementation, it is possible to establish references from design elements to the corresponding requirements. Such references are known as traceability links, and can be established using the integration between DOORS and Tau. See Working together with DOORS for more information about the capabilities of this integration.

During application development the traceability links between the Tau model and the DOORS requirements are mainly used for navigation purposes. However, when generating code from the model this information can also be used for annotating the generated code with references to the DOORS requirements. Having such annotations in the generated code helps answering common questions like the following:

- Have all requirements been implemented by the generated application?
- Which impact will a new requirement have on the generated code?
- Does one single C, C++ or Java definition realize multiple requirements?
- Is the implementation of one requirement localized to the same source file or is it spread out over multiple files?
- Is the implementation of one requirement localized to the same module (library or executable) or is it spread out across the entire application?
- Which impact will the removal of an implemented requirement have on the generated code?

Requirements references in generated code also makes the generated code more readable, and are sometimes even required when validating the code against certain standards or certification programs.

# The U2ReqTrace Add-in

The feature of annotating generated code with requirements references is available in the form of a Tau add-in called U2ReqTrace.

The intended workflow when using this add-in is the following:

1. Requirements are defined using Telelogic DOORS, and are saved in one or many formal DOORS modules.
2. When designing and implementing the UML model these formal DOORS modules are imported into Tau. See Importing requirements for more information on how to do this.
3. Traceability links are established between UML model elements and DOORS requirements. See Requirement relations for more information on how to do this.
4. The U2ReqTrace add-in is activated.See Activating add-ins for information about how to activate an add-in.
5. The `<<requirementRefAnnotation>>` stereotype is applied on the build artifact. Appropriate Options are set as tagged values for this stereotype, using the Properties Editor.
6. A C, C++ or Java application is generated from the application. A definition in the C/C++/Java code that corresponds to a UML definition for which a traceability link exists will get an annotation in the form of a source code comment. It contains a reference to the DOORS requirement, and also the requirement text. Which information from the DOORS requirement that will be printed can be customized by Options to U2ReqTrace.

Although these steps are presented sequentially here, it is common to perform them in an iterative manner. If a new requirement is added at a later stage, the DOORS module can be reimported and traceability links can be added to the new requirement (either from new or existing UML model elements). Then the code can be regenerated (fully or partially).

## Annotation Formatting

Generated code is annotated by means of source code comments. By default these annotations are printed just before the C/C++/Java definition that has a requirement reference. It is also possible to have all annotations generated at either the beginning or end of a source file (see Options for more information).

The formatting of the annotation comments depend on where they are generated. A comment that is generated just before the annotated definition has the following format:

```
/*** Realizes requirement /<DB>/<P>/<M>#<ID>
    <Req Heading>
    "<Req Short Text>"
    "<Req Object Text>"
**********************************************************/
```

A comment that is generated at the beginning or end of a file has the following format:

```
/**** Requirements realized by definitions in this file
****
************************************************************
*
<Def Name>(<Line>) : /<DB>/<P>/<M>#<ID>
    <Req Heading>
    "<Req Short Text>"
    "<Req Object Text>"
**********************************************************/
```

When generating C++ or Java code double-slash comments (//) will be used instead of C comments (/* */).

The “variables” used above have the following meaning:

| | |
|---|---|
| <DB> | The name of the DOORS database |
| <P> | The path to the DOORS formal module |
| <M> | The name of the DOORS formal module |
| <ID> | The requirement ID |
| <Req Heading> | The requirement heading text |
| <Req Short Text> | The short text of the requirement |

| | |
|---|---|
| <Req Object Text> | The object text of the requirement |
| <Def Name> | The name of the annotated definition |
| <Line> | The line number where to find the generated definition in the file |

**Example 341: Formatting of requirement annotation comments**

A comment printed just before the annotated definition could look like this:

```
//// Realizes requirement /DOORS
Database//Tau2_7_Requirements/U2ReqTrace#6
//  Support both C, C++ and Java code generators
//  "Requirement traceability should be supported both for
the C, C++ and Java code generators"
////////////////////////////////////////////////////////
/
class U2ReqTracer {};
```

A comment printed at the beginning or end of a file could look like this:

```
///// Requirements realized by definitions in this file
////
////////////////////////////////////////////////////////
/
//U2ReqTracer(12) : /DOORS
Database//Tau2_7_Requirements/U2ReqTrace#6
//  Support both C, C++ and Java code generators
//  "Requirement traceability should be supported both for
the C, C++ and Java code generators"
////////////////////////////////////////////////////////
/
```

## Note

*When using the U2ReqTrace add-in for a C++ build artifact that uses roundtripping, the annotation comments will be printed inside GENERATED tags since the information they contain does not correspond to anything in the UML model. This ensures that roundtripping can be used also on annotated files.*

## Options

Generated code will be annotated with requirement references if the `<<requirementRefAnnotation>>` stereotype is applied on the build artifact that is used when generating the code. This stereotype is defined in the ReqTraceability profile package that gets loaded when the U2ReqTrace add-in is activated.

The following options can be set as tagged values for the `<<requirementRefAnnotation>>` stereotype, using for example the Properties Editor:

- **Annotate Code**
  This option must be enabled for the code to be annotated. If it is disabled the other options are not relevant. This option is useful if you want to temporarily disable the printing of requirement references while still keeping the values of all the other options.
- **Text Size Limit**
  This option specifies the maximum number of characters to print for any DOORS requirement text. By default the limit is set to 255 characters. If you want the entire text to be printed, set this option to 0.
- **File Placement**
  This option allows you to control where in the generated file the requirement references are printed. The default value is “Before Definition” which means that requirement references will be printed immediately before the corresponding C/C++/Java definition. It is also possible to set this option to “File Header” (all requirements are printed at the beginning of the file) or “File Footer” (all requirements are printed at the end of the file). Note that the formatting of the printed annotations depends on this option. See Annotation Formatting for more information and examples.
- **Print Headings**
  This option controls whether the heading of the requirement should be printed. It is enabled by default.
- **Print Short Text**
  This option controls whether the short description (short text) of the requirement should be printed. It is enabled by default.
- **Print Object Text**
  This option controls whether the long description (object text) of the requirement should be printed. It is enabled by default.

## Usage

The U2ReqTrace add-in has no specific commands in the user interface. It is completely integrated with the C, C++ and Java code generators and is automatically invoked after the generation of C/C++/Java code. More precisely it is invoked after the insertion of the code generation result package that these code generators produce.

The following message is printed in the Build log when U2ReqTrace has annotated a generated file:

```
Successfully annotated "C:\CGTest\MyFile.h" with
requirements references.
```

In case any errors occur, these are also printed as messages in the Build log.

Note that U2ReqTrace requires a connection to DOORS in order to extract the requirement information. If DOORS has not been started when generating the code, it will be launched automatically.

## API Access

The U2ReqTrace is implemented as an agent. It can therefore be accessed from the public Tau APIs using the InvokeAgent API method. The agent is defined in the ReqTraceability profile and is called `AnnotateGeneratedCodeWithRequirementsReferences`.

It is also possible to integrate U2ReqTrace with any custom code generator. As can be seen in the profile the agent is triggered on the Insert cross reference file tool event. Hence it will work for any code generator that can produce a cross-reference file (also known as a code generation result package) on the same format as used by the C, C++ and Java code generators. U2ReqTrace works by analyzing the result package contents, extracts DOORS information for generated definitions that have traceability links to DOORS requirements, annotates the generated code with source code comments containing the extracted information, and finally updates the information in the result package if necessary (line numbers may need an offset to account for inserted comments).

# *UML for C Code Generation*

The chapters listed under UML for C Code Generation describe how a UML project is turned into a C application using the C Code Generator and the AgileC Code Generator.

# 32

# *Environment Functions for C Applications*

This section describes how you should proceed in order to design the interface between application code generated with the C Code Generator and the environment to the system.

An overview of the architecture of a C application is presented, and how it is interfaced with its environment.

Guidelines and design recommendation for the environment functions are then given. The intended use is to provide guidelines and hints how to include the functions into the application code in order to achieve good results.

**See also**

In case you need to look up reference information about the operation principles for the C Code Generator, please refer to the following chapters:

C Code Generator Reference

C Code Generator Run-Time Model

C Code Generator Symbol Table

C Code Generator Macros

# Introduction

An application generated using the C Code Generator can be viewed as consisting of three parts:

- The generated code that implements the system
- The physical environment of the system
- The environment functions, where you connect the system with the environment of the system

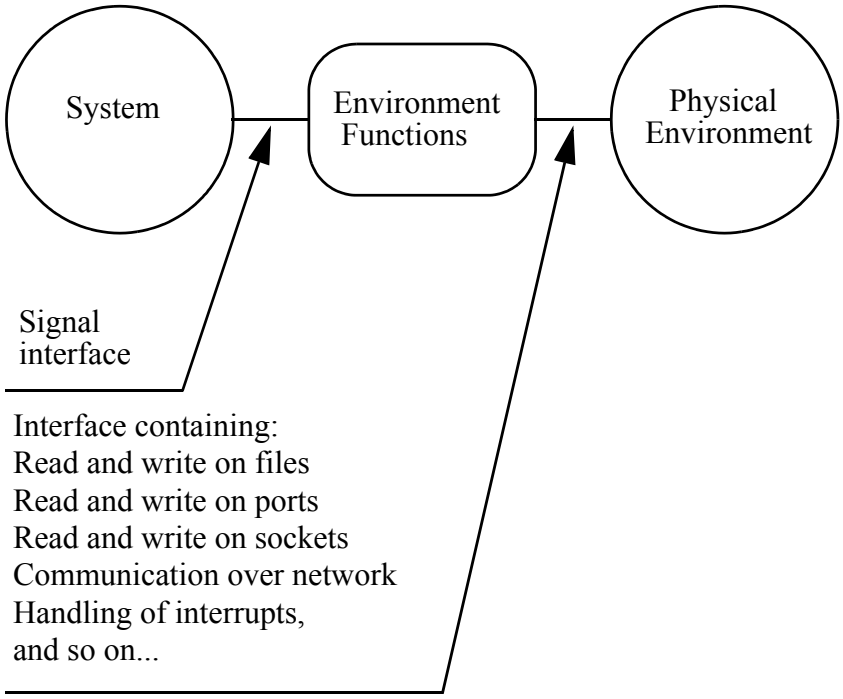



*Figure 238: Interface between application and environment.*

### Generated code

The system behaves as a state machine, in which transitions are executed in priority order, signals are sent from an instance of an active class to another instance, initiating new transitions, timer signals are sent, and so on. These are examples of internal actions that only affect the execution of the system.

A system communicates with its environment by sending and receiving signals.

### Physical environment

The physical environment of an application could consist of an operating system, a file system, some hardware devices, a network of computers etc. In this “real world” other actions than just signal sending are required. Examples of actions that an application has to perform are: to read or to write on a file, to send or receive data over a network, to respond to interrupts signals, to read and to write information on hardware ports or on sockets.

### Environment functions

The environment functions are the place where the two worlds, the system and the physical environment, meet. Here, signals sent from the system to the environment can induce all kinds of events in the physical environment, and events in the environment can cause signals to be sent into the system.

You have to provide such environment functions, as the C Code Generator has no knowledge of the physical environment, or of the actions that should be performed.

### Distributed applications

In a distributed system, an application might consist of several communicating systems. Each system will then become one executable program. It might execute either as an operating system task, communicating with other operating system tasks, or it might execute in a processor of its own, communicating over a network with other processors. There may also be combinations of these cases. For the sake of simplicity, the operating system tasks or processors nodes are called communicating over a network. In the event of communicating operating system (OS) tasks, the network will be the media for the inter-process communication provided by the OS.

Generating an application consisting of several nodes communicating over a network is possible using the C Code Generator. However, you will then need 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 C Code Generator from UML models. As long as a node can communicate with other nodes, it may be implemented using any technique.*

The Pid values (references to instances of active classes), are a problem in a distributed world containing several communicating systems. It is desired that, for example, “Sender.Output” works, even if Sender refers to an instance of an active class in another system. To cope with this, 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 instance is resident, while the other component in the Pid value is a local identification of the instance within its node.

### Additional advantages

The split of an application into the system and the environment functions has additional advantages. It separates the logical decision to perform the action (for example the decision to send a signal to the environment) from the implementation details of the action (for example handling the signal in the environment functions).

This kind of separation reduces the complexity of the problem and also allows separate testing. Furthermore, it allows the development in parallel of the logical behavior (implemented in the system) and the interface towards the environment (implemented in the environment functions). When the signal interface between the system and its environment is defined, it is possible to proceed with both development activities in parallel, and easily integrate the results at the end.

### Building the application

How to build the Application, i.e. generate code from the UML model, compile and link the code using the appropriate code generator settings, compiler options and so forth is described in Chapter 27, Building Applications Reference.

### Simulating and debugging the application

When an application is developed, it is appropriate to start using the Model Verifier to simulate and debug the UML model on your host computer, without any environment, in order to verify that the code from the UML model behaves as expected. In such a simulation mode, the environment is provided through the Model Verifier user interface, which allows you to receive signals from the environment, and to observe and trace the signals that are sent to the environment.

### Target application

Your next task is to include the required support to interface the application with the environment, and then to compile and link the application for your target environment.

**See also**

C and AgileC Runtime Libraries

# Essentials About Generated C Code

This section describes briefly what you need to know about how signals and instances of active classes are represented in the generated C code. The symbol table, which is a representation of the static structure of the system, is also discussed to help you understand how to proceed when interfacing the generated code with the environment functions. The information provided in this section is used when the environment functions are described later in this guide.

An exhaustive reference documentation to the generated C code is available in: C Code Generator Run-Time Model and C Code Generator Symbol Table.

## 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.

A general type definition `xSignalRec` for a signal without parameters and for a pointer to such a signal, `xSignalNode`, are given below. These types can 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;
```

Such 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 instance of an active class. 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 signal sendings. 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 instance.
- `Sender`. The sending 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 system and contains, among other things, one node for each signal type that is defined within the system.
- `Prio`. The priority of the signal.

In the generated code there will be types to represent the parameters of the signals according to the following examples:

**Example 342: Generated C code for signal definition**

Assume the following signal definitions:

```
signal S1(Integer);
signal S2;
signal S3(Integer, Boolean, OwnType);
```

then the C code below is generated:

```
typedef struct {
  SIGNAL_VARS
  SDL_Integer Param1;
} ySignalPar_z0f_S1;
typedef ySignalPar_z0f_S1 *yPDP_z0f_S1;

typedef struct {
  SIGNAL_VARS
  SDL_Integer Param1;
  SDL_Boolean Param2;
  z09_OwnType Param3;
} ySignalPar_z0h_S3;
typedef ySignalPar_z0h_S3 *yPDP_z0h_S3;
```

where `SIGNAL_VARS` is a C macro defined in `scttypes.h` That macro is expanded to the common components in a signal struct.

**Note**

*The signal S2 has no type definition, which is correct since no types are generated for signal without parameters.*

---

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.

**Example 343: Generated C code for structure signal parameters**

```
<<struct>> class MyStruct {
  Integer x;
  Integer y;
}

signal MySig1( MyStruct );
```

Generated C code:

```
typedef struct z_paraext_H0_MyStruct_s {
  SDL_Integer x;
  SDL_Integer y;
  z09_OwnType Param3;
} z_paraext_H0_MyStruct;

typedef struct z_paraext_H0_MyStruct_s
*z_paraext_H1_ptr_MyStruct;

typedef struct {
  SIGNAL_VARS
  z_paraext_H1_ptr_MyStruct Param1; /* Note: this is a
pointer to structure */
} ySignalPar_z_paraext_S2_MySig1;

typedef ySignalPar_z_paraext_S2_MySig1
*yPDP_z_paraext_S2_MySig1;
```

---

By default, in UML all structure and class attribute have reference aggregation, and signal MySig1 will contain a reference to structure MyStruct, which will result in a pointer in the generated C code. If this is the case and reference is really needed, then when sending such signals from environment, memory allocation and deallocation for signal parameters should be handled manually by user.

If you need to pass MyStruct parameters by value, signal parameter should be defined with composite aggregation like this:

signal MySig1( part MyStruct );

This can be done either manually in the text diagram, or in the Properties menu for signal parameter by choosing “Composition (part)” value for “Aggregation” drop-down list.

### Representation of UML data types in C

In the section “Translation of Data Types” on page 1100 is a reference to the C representation of UML data types.

### Encoding and decoding of signal parameters

To implement an automatic handling of signals including its parameters (like coders/decoders), run-time information about the layout of data in the signals is needed. The C Code Generator generates such information if `Set-SDL-Coder` is specified in the Advanced options for the C Code Generator.

With this option on, the C Code Generator will generate two files for you: `<basename>_cod.h` and `<basename>_cod.c` The C Code Generator also automatically updates the generated makefile in order to include compile and link of the generated `<basename>_cod.c` file.

**Note**

*The coder library that is present in the Tau installation is **not** supported to be used in any way as a framework for coders/decoders.*

### Representation of signals outside interfaces

A signal which is defined in a top-level package (or its sub-packages) and is present in an interface inside the same package (or in sub-packages) will be the same signal in the generated C code.

When there is no common signal definition in a package, signals with identical names defined in different interfaces are treated as different signals in generated C code.

**Example 344: Signal defined in top-level package** ——————————————

The signal “`S`” below will be of the same signal definition in both interface `I1` and `I2`.

```
package signals {
    signal S;
    interface I1 {
        signal S;
    }
    active class C <<import>> dependency to p1 {
        active class C1 {
            port in with I1;
        }
        part C1 C1;
        part p1::C2 C2;
    }
    package p1 {
        active class C2 {
            port out with I2;
        }
        interface I2 {
            signal S;
        }
    }
}
```

## Types representing instances of active classes

A Pid value is a struct consisting of two components, a global node number, which is an integer (usually generated by Function xGlobalNodeNumber) 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 system that the instance of an active class belongs to, while the local Pid value identifies the instance within the system. The local Pid pointer value is only valid within this system and must therefore not be referenced outside the 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 systems. You can also define your own Pid values in non-UML defined parts of the application and still communicate using signals.

The variable SDL_NULL, which represents a null value for instances of type Pid and which is defined in the run-time library and available through the file scttypes.h, contains zero in both the global node number and the local Pid component. The global node number should be greater than zero in all Pid values except SDL_NULL.

### 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 system. The symbol table contains, among other things, nodes that represent signal types, active classes, connectors and operations. The C type that is used to represent for instance 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 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 ports in the top class are also of interest to refer to. In a similar way as for signals such nodes may be referenced using the name yChaN_ followed by the port name with prefix.

## Environment Functions

A 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 system, the C Code Generator 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 between 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 task using some OS primitive.

You must provide the following functions to handle the environment of the system:

- `xInitEnv` that is called during the startup of the application, to properly initialize the environment
- `xCloseEnv` that is called when the application terminates, in order to ensure that the environment is "closed" in appropriate way, such as closing files and sockets.
- `xOutEnv` that manages the signals sent from the system to the environment
- `xInEnv` that manages the signals sent to the system from the environment.
- `xGlobalNodeNumber` that handles issues that arise if you have multiple communicating systems.

These functions are discussed more in detail in the section "Guidelines for Environment Functions" on page 1047.

## Function skeletons

There are two ways to create a skeleton for the environment functions:

- The normal method is to generate a skeleton by having the Application Builder generate the environment functions for you. An advantage with the generated environment functions is that the C Code Generator knows about the signal interface to be implemented in the environment functions, and can therefore insert code or macros for all signals in the interface. To calculate this information by yourself is not easy.

  In a few simple cases you will obtain executable environment functions by just adjusting the macros in this generated file, but in the general case you should use it as a skeleton and augment it to fit your needs. Remember to save the file with another name so that it is not overwritten when code is generated next time.
- You work on a copy of the file `sctenv.c` from the directory:
  `.../sdlkernels/include`
  This file also includes some trace features that may be used to trace the execution. This trace can, however, only be used if you adapt the source code for the run-time library (included in the C Code Generator) to compile a new object library using the appropriate compiler macros.

## System interface header file

This section is valid both for the C Code Generator and the AgileC Code Generator.

The **system interface header file** contains code for the objects that are defined on the system level. 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 can also be viewed as the environment header file.

The default name of the generated interface header file is `<system_file_name>.ifc`.

### Contents of the system interface header file

The system interface header file has the following contents and structure:

- Macros for all constant attributes that are translated to macros.
- All type definitions generated from passive classes and syntype declarations.
- External definitions of all constant attributes 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 on the system level, there will be definitions of the types `ySignalPar_SignalName` and `yPDP_SignalName`, that is of the types used to represent a signal.
- For each remote operation (that can be sent to or from the environment), code is generated exactly as for the signals named `pCALL_procedurename` and `pREPLY_procedurename`.
- For each connector and port defined in the system diagram there will be `extern` definitions for the `xChannelIdRec` representing the connector.
- Together with these definitions, macros that simplify the translation of names to C names are also generated.
- Optionally, information about signal paths from the application to the environment. See "Deducing signal path to the environment" on page 1049.

### Names of UML objects in C

Due to differences in naming rules in UML and C, prefixes or suffixes are generated by the C Code Generator to make C identifiers unique. These prefixes or suffixes, however, may change when you update your system 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. The system interface header file is regenerated each time you regenerate code for the system. It is not necessary to make any changes in the environment functions, as the interface names are static.

A reference section to naming and prefixing conventions is available in section "Names in Generated C Code" on page 1121 in Chapter 35, *C Code Generator Reference*.

### Prefix

In the generated code for the system, except the `.ifc` file, all UML name are prefixed or suffixed with an incremental number to make the names unique in C. This, however, makes the C names impossible to determine and they might change between two code generations if for example a new definition is inserted. In the `.ifc` file the UML names have a predefined prefix according to the table below:

| UML entity | C name |
|---|---|
| connectors | `cha_%n` |
| constants | `con_%n` |
| data types | `typ_%n` |
| literals | `lit_%n_%s` |
| signals | `sig_%n` |

where `%n` is replaced by the UML name of the entity and `%s` is replaced by the UML name of the data type.

These prefixes are controlled via the Code generation properties in the Properties Editor.

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.

**Example 345: Macro in the system interface header file**

If a signal called Sig1 is defined in the system, the following macro is created:

```
extern XCONST struct xSignalIdStruct ySigR_z5_Sig1;
#ifndef sig_Sig1
#define sig_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 will ensure that static names are generated for the following types:

- Constant attributes in packages (both translated to macros and variables).
- Passive classes and syntype declarations. If the passive class is translated to an enumeration type, all the literals are available directly in C using their UML names.
- `xSignalIdNode` representing signals. (No `ySigN_` prefix).
- `xChannelIdNode` representing connectors and ports. (Use prefix `xIN_` or `xOUT_` to access the incoming or outgoing direction of the connector).
- The `yPDP_SignalName` pointer type. This type may be referred to using the name `yPDP_SignalName`, where `SignalName` is the UML name.

**Note**

*You must always generate the system interface header file before editing or generating the environment functions.*

## Signal number file

**Note**

*This section is valid for the C Code Generator only.*

The signal number file is optionally generated by the C Code Generator when building the application (see "Advanced options" on page 973*)*. It contains information about signal numbers (assigned by the C Code Generator) and the signal names these numbers originate from. How to take advantage of this feature is described and illustrated in subsection "Improving performance of xOutEnv when many signals" on page 1050.

The signal number file is named `<basename>.hs`

### Signal parameter layout file

**Note**

*This section is valid for the C Code Generator only.*

# Guidelines for Environment Functions

The file containing the environment functions should have the following structure:

```
#include "scttypes.h"
#include "systemfilename.ifc"

void xInitEnv XPP((void))
{
}

void xCloseEnv XPP((void))
{
}

void xOutEnv (xSignalNode *S)
{
}

void xInEnv (SDL_Time Time_for_next_event)
{
}

int xGlobalNodeNumber XPP((void))
{
}
```

## Functions xInitEnv and xCloseEnv

These functions handle initialization and termination of the environment.

```
void xInitEnv ( void );

void xCloseEnv ( void );
```

In the implementation of these functions, you must 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 as inline C code: `[[SDL_Halt]]`

`SDL_Halt` is part of the run-time library and has the following definition:

```
void SDL_Halt ( void );
```

`xInitEnv` will be called before the system is initialized, which means that no references to the system are allowed in this function.

## Function xOutEnv

Each time a signal is sent from the system to the environment, the function `xOutEnv` will be called. The function has the following prototype:

```
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 contains the signal type, the sending and receiving instances and the parameters of the signal.

The types used to represent signals and instance of active classes were presented earlier in this chapter, in sections "Types representing signals" on page 1037 and "Types representing instances of active classes" on page 1041.

The parameter of `xOutEnv` is an address to a `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 function `xOutEnv` you may use the information in the signal that is passed as parameters to the function. First it is suitable to determine the 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`:

```
(*S)->NameNode == Sig1
```

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 instance, while `Receiver` is either a reference to a specific instance of an active class in the environment, or has the value `xEnv`. `xEnv` is a Pid value that is used to represent the generic "environment active class" without specifying an explicit instance.

Receiver will refer to the "active class" `xEnv` if the Pid expression in an addressed signal sending refers to `xEnv`, or if the signal is sent without direct addressing and the environment is selected as receiver in the scan for receivers.

Remote operation calls to or from the environment should in the environment functions be handled by two signals, `pCALL_procedurename` and `pREPLY_procedurename`.

### Deducing signal path to the environment

In some situations it is useful to know the path (outgoing port) from the application for a given signal, when writing the `xOutEnv` function. By compiling the application with the compilation switch XPATH_INFO_IN_ENV_FUNC this information becomes available. Switching these kind of flags on is done by changing the make.opt file and setting the sctUSERDEFS flag correctly, see Library files.

The `xOutEnv` function will then have the prototype:

```
extern void xOutEnv (xSignalNode *, xChannelIdNode);
```

**Note**

*When using this feature, remember to update the xOutEnv function implementation according to this definition.*

Information about outgoing path is found in the generated `.ifc` file.

**Example 346: Outgoing path in the generated system header file**

```
#ifndef XOPTCHAN
extern XCONST struct xChannelIdStruct yChaR_c_0;
extern XCONST struct xChannelIdStruct yChaRR_c_0;
#define xIN_c (&yChaRR_c_0)
#define xOUT_c (&yChaR_c_0)
#endif
```

`xIN_c` and `xOUT_c` are macros that can be used to refer to the incoming and outgoing directions of the path (“c” in this case). In the function `xOutEnv` `xOUT_c` is the direction of interest in this case. To test if the signal to be treated has been conveyed to the environment through the path “c”, a test like the following can be used:

```
void xOutEnv(xSignalNode *SignalOut, xChannelIdNode Path)
{
  ...
  if (Path == xOUT_c) ...
  ...
}
```

### Improving performance of xOutEnv when many signals

In the normal case signals are identified by pointers (of type `xSignalIdNode`). This means that finding the type of a signal in `xOutEnv` has to be performed using a sequence of `"else if"` C statements. If the `xOutEnv` function should treat a fairly large number of signals this will become inefficient. By identifying signals by their numbers, a `switch` statement can be used instead, which improves execution performance.

To take advantage of the signal number feature, you have to perform two operations:

1. Specify **Set-Signal-Number** in the Advanced options for the C Code Generator, which results into the creation of a Signal number file
2. Compile the application with the compilation switch XUSE_SIGNAL_NUMBERS

The compilation switch will include the signal numbers into the `xSignalIdStruct`. This means that in `xOutEnv` a switch according to the example below can be used:

```
switch ((*SignalOut)->NameNode->SignalNumber) {
  case SN_signalname1 : ....
  case SN_signalname2 : ....
  ....
}
```

### Guidelines for the xOutEnv function

You can write the `xOutEnv` function as you wish. The structure presented below may be seen as an example of how to design `xOutEnv` functions. You may also want to design the function so that it takes advantage of signal numbers rather than signal names, see "Improving performance of xOutEnv when many signals" on page 1050.

**Example 347: 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

The function xInEnv is used to make it possible to receive signals from the environment and to send them into the system. In a Bare integration this function is called repeatedly during the execution of the system. In a Threaded integration the `xInEnv` function is executed in a thread of its own, and should thus never return. During execution `xInEnv` should scan the environment to see if anything has occurred which should trigger a signal to be sent to an instance of an active class within the system.

```
void xInEnv (SDL_Time Time_for_next_event);
```

To implement the sending of a signal into the 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. These functions will be described below.

The parameter `Time_for_next_event` is not meaningful in a Threaded integration, as the function is only called once. The function should contain an infinite loop:

```
function xInEnv
{
  while (1) {
    Wait_for_something_to_do;
    /* Handle different signals */
  }
}
```

where "`Wait_for_something_to_do`" is some user defined statements that hang the execution of this thread until there is some signal to be sent. For example `xInEnv` can wait on a semaphore. This semaphore is then posted in interrupt routines or in other external code when a signal should be sent into the system.

In a bare integration the parameter `Time_for_next_event` will contain the time for the next event scheduled in the system. The parameter will be either of the following:

- 0 or Now, which indicates that there is a transition (or a timer output) that can be executed immediately.
- Greater than Now, indicating that the next event is a timer output scheduled at the specified time.
- 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 signal sending, and return as fast as possible if Time has passed `Time_for_next_event`.

If Time has not passed `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 (a signal sent to the system) or until Time has passed `Time_for_next_event`.

**Note**

*Good practice during debugging of bare integrations is to return from the* `xInEnv` *function as fast as possible to ensure that it will behave properly. Otherwise, the keyboard polling, that is, typing* `<RETURN>` *in order to interrupt the execution, will not work as expected. Returning as fast as possible when there is nothing to do introduces "busy waiting", which is something that should be avoided in a resulting application.*

### Function xGetSignal

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 an instance of an active class within the system, or the value `xNotDefPId`. The value `xNotDefPId` is used to indicate that the signal should be sent without direct addressing, while if a Pid value is given, it is treated as a signal sending with direct addressing. Pid values for instances of active classes in a system cannot be calculated, and have to be captured from the information (sender or parameter) carried by signals coming from the system.
- `Sender`. Sender should either be a Pid value representing an instance of an active class in the environment of the current system or the value `xEnv`. `xEnv` is a Pid value that refers to a generic "environment active class", which is used to represent the general concept of system environment, without specifying an explicit instance in the environment.

### Function SDL_Output

The run-time library function `SDL_Output` is called from within the PAD functions, to implement the sending of signals.

The function SDL_Output takes a reference to a signal instance and sends the signal after it has identified the receiver.

When SDL_Output identifies the receiver of a signal to be an instance of an active class that is not part of the current system, SDL_Output will call the xOutEnv function.

```
void SDL_Output (
                xSignalNode   S
  xSigPrioPar(int             Prio),
                xIdNode       ViaList[] );
```

- S. This parameter should be a reference to a signal instance with all components filled in.
- Prio. This parameter sets the priority of the signal where 0 is the highest and 255 the lowest. Use the xSigPrioPar macro to specify this parameter to ensure that your code will work without modifications even if you run a kernel configuration without signal priorities.
  xDefaultPrioSignal is defined to be the default signal priority. Please note that there should be no comma between the S and the Prio parameters when using the xSigPrioPar macro.
- ViaList. This parameter is used to specify if a VIA clause is or is not part of the signal sending statement. The value (xIdNode *)0 (a null pointer), is used to represent that no VIA clause is present. See Example 349 on page 1055.

This is sufficient information to be able to write the code to send a signal.

**Example 348: C code to send a signal from the environment**

A signal S1, without parameters, should be sent from xEnv into the system without an explicit receiver. The code will then be:

```
SDL_Output(
  xGetSignal(S1, xNotDefPId, xEnv)
  xSigPrioPar(xDefaultPrioSignal),
  (xIdNode *)0 );
```

If S2, with two integer parameters, should be sent from xEnv to the 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 xSigPrioPar(xDefaultPrioSignal),
(xIdNode *)0 );
```

To introduce a via list in the signal sending takes a variable, which should be an array of `xIdNode`, contains references to the symbol table nodes representing the current connectors in the via list. A variable like this is hence needed:

```
ViaList xIdNode[N];
```

where N should be replaced by the length of the longest via list you 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 connector (its symbol table node) in the via list, component 1 is a reference to the second connector, and so on. The last component with a reference to a connector 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 connectors, C1 and C2.

**Example 349: Via list of two connectors.**

```
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

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 system, then the appropriate code to send a signal (Example 348 on page 1054) should be executed.

The example below shows an `xInEnv` function for a bare integration. In a Threaded execution model the infinite loop and the wait statement discussed above should be added.

**Example 350: 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)
      xSigPrioPar(xDefaultPrioSignal),
      (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 xSigPrioPar(xDefaultPrioSignal),
      (xIdNode *)0 );
  }
  /* and so on */
}
```

This basic structure can be modified to suit your 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` statement can be introduced last in the `if` statements. This means that only one signal is sent during a `xInEnv` call, which reduces the latency.

# 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 systems that constitutes an application. If the application consists of only one system, then this number is of minor interest although it still needs to be set. The global node number is used in Pid values to identify the node (OS task or processor) that the instance of active class belongs to. Pid values are thereby universally accessible and you may, for example, in a simple way make “Sender.Output” work between instances of active classes that are present in different systems.

When an application consisting of several communicating systems is designed, you have to map the global node number to the current OS task or processor, in order to be able to transmit signals addressed to non-local instances of Pid to the correct OS task or processor. This will be part of the `xOutEnv` function.

## Functions xMainInit and xMainLoop

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 the instances of active classes during state 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 instance in the system will be represented by a PAD function, which is called when an instance of the current instance set is to execute a transition.

How to use the `xMainInit()`, and `xMainLoop()` functions within the `main()` function is best shown with an example.

**Example 351: Start up, and endless loop**

```
void main ( void )
{
 xMainInit();
 xMainLoop();
}

void xMainInit ( void )
{
  xInitEnv(); /* Init of internal data structures in the
run-time library */
  yInit();
}

void xMainLoop ( void )
{
  while (1) {
    xInEnv(...);
    if ( Timer output is possible )
      SDL_OutputTimerSignal();
    else if ( transition is possible )
      Call appropriate PAD function;
  }
}
```

### Stopping the execution

As discussed previously, the function `xMainLoop` contains an endless loop. The appropriate way to stop the execution of the program is to call the run-time library function `SDL_Halt`. The call of this C function should normally be included in an appropriate action, using inline C code.

The `SDL_Halt` function has the following structure:

```
void SDL_Halt ( void )
{
  xCloseEnv();
  exit(0);
}
```

# 33

# *C and AgileC Runtime Libraries*

This chapter is a description of the implementation and interface to the run time libraries used by the C Code Generator and the AgileC Code Generator. The libraries are supplied in C source code.

In this document are also discussed and explained how to create new libraries, from the default set that is delivered with the C Code Generator and the AgileC Code Generator.

Lastly is discussed how to adapt the libraries to comply with the requirements and conventions used by the C compiler that you will use to compile and link your applications.

# Runtime Libraries

### Run-Time library directory structure

The structure of files and directories used for the C Code Generator runtime libraries is depicted below. The root directory for this structure is named `sdlkernels` and it includes all the files that implement the run-time library that should be compiled and linked with the code generated by the C Code Generator.

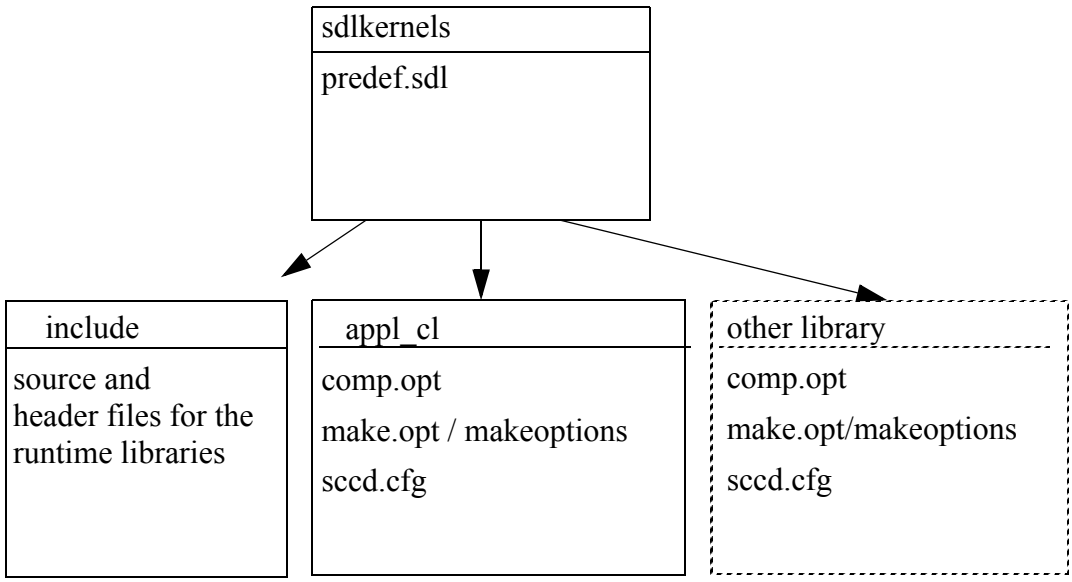



*Figure 239: Directory structure of the runtime libraries.*

### sdlkernels

In the `sdlkernels` directory the file `predef.sdl` is found. It contains the definitions of predefined sorts and is used by the C Code Generator when checking the syntax and semantics of the system prior to generating C code.

A run-time kernel is defined by the source code in the include directory, with the options from one of the Runtime libraries applied.

### include

In the include directory the source code files for the run-time library and the Model Verifier are found. These files are listed and explained in "Included source and header files" on page 1070.

### Runtime libraries

In parallel with the `include` directory there are a number of directories, each one containing the files `comp.opt` and `makeoptions` (or `make.opt` on Windows). By using the compile and link settings defined by these files, both the generated application code and the files contained in the include directory are compiled in such a way that the executable code is given the properties that are specific for the target application of your choice.

- The `comp.opt` file determines the contents of the generated makefile and how "make" is called.
- The `makeoptions / make.opt` file describes the properties of the library, such as the compiler used, compiler options, linker options, and so on.
- The `sccd.cfg` file is a configuration file for the C Compiler Driver utility, which is used to make the generated C Code easier to read for you.

### Name conventions for directories

The directories that contain the compiler and library properties follow a naming scheme to abbreviate a verbatim description of the properties of the generated code.

The syntax that is adopted for the naming scheme is:

```
<"dbg" | "appl" | "applt"> || <compiler> || [_cpp]
```

- `dbg`: indicates that the result will be a Model Verifier suitable for debugging your application on host
- `appl`: indicates that the result will become a bare application.
- `applt`: indicates that the result will be a threaded application.
- `compiler`: this abbreviation indicates which compiler and linker will be used.
- `_cpp`: this suffix indicates that the code is to be compiled in C++ compatible mode.

**Example 352: Compiling an application for Windows**

The `appl_cl` directory in Figure 239 on page 1060, is an example of such a directory.

By using the `comp.opt` and `make.opt` files in that directory, you will compile and link an application for Windows.

---

When building the application using the Application Builder, those abbreviations will instead be substituted by more explicit texts in the graphical user interface of the Properties Editor.

## Supported libraries

Below is a table with libraries that are available in the installation, each one to be used together with any of the run-time and environment integration models Bare or Threaded.

**Note**

*The settings that are defined in the attributes for the stereotypes used for the build types C Code Generator and Model Verifier designate a directory with the corresponding library.*

### Runtime libraries defined by attributes

The table below is a summary of all combinations for each of the attributes that define the settings for the build types C Code Generator and Model Verifier, and the resulting run-time library that will be used when compiling and linking the code. It can be noted that the support for OS integrations uses a deployment model described in the AgileC Code Generator.

| Build Type | Support C++ | Target kind | Threading model | Resulting run-time library |
|---|---|---|---|---|
| AgileC Code Generator | False | Solaris Forte | Bare | `agilec/agilec_appl_cc` |
| AgileC Code Generator | True | Solaris Forte | Bare | `agilec/agilec_appl_cc_cpp` |
| AgileC Code Generator | False | Solaris Forte | Threaded | `agilec/agilec_applt_cc` |
| AgileC Code Generator | True | Solaris Forte | Threaded | `agilec/agilec_applt_cc_cpp` |

| Build Type | Support C++ | Target kind | Threading model | Resulting run-time library |
|---|---|---|---|---|
| AgileC Code Generator | False | Solaris gcc | Bare | `agilec/agilec_appl_gcc` |
| AgileC Code Generator | True | Solaris gcc | Bare | `agilec/agilec_appl_gcc_cpp` |
| AgileC Code Generator | False | Solaris gcc | Threaded | `agilec/agilec_applt_gcc` |
| AgileC Code Generator | True | Solaris gcc | Threaded | `agilec/agilec_applt_gcc_cpp` |
| AgileC Code Generator | False | Win32 | Bare | `agilec/agilec_appl_cl` |
| AgileC Code Generator | True | Win32 | Bare | `agilec/agilec_appl_cl_cpp` |
| AgileC Code Generator | False | Win32 | Threaded | `agilec/agilec_applt_cl` |
| AgileC Code Generator | True | Win32 | Threaded | `agilec/agilec_applt_cl_cpp` |
| C Code Generator | False | Solaris Forte | Bare | `appl_cc` |
| C Code Generator | True | Solaris Forte | Bare | `appl_cc_cpp` |
| C Code Generator | False | Solaris Forte | Threaded | `applt_cc` |
| C Code Generator | True | Solaris Forte | Threaded | `applt_cc_cpp` |
| C Code Generator | False | Solaris gcc | Bare | `appl_gcc` |
| C Code Generator | True | Solaris gcc | Bare | `appl_gcc_cpp` |
| C Code Generator | False | Solaris gcc | Threaded | `applt_gcc` |

| Build Type | Support C++ | Target kind | Threading model | Resulting run-time library |
|---|---|---|---|---|
| C Code Generator | True | Solaris gcc | Threaded | `applt_gcc_cpp` |
| C Code Generator | False | Win32 | Bare | `appl_cl` |
| C Code Generator | True | Win32 | Bare | `appl_cl_cpp` |
| C Code Generator | False | Win32 | Threaded | `applt_cl` |
| C Code Generator | True | Win32 | Threaded | `applt_cl_cpp` |
| Model Verifier | False | Solaris Forte | Bare | `dbg_cc` |
| Model Verifier | True | Solaris Forte | Bare | `dbg_cc_cpp` |
| Model Verifier | False | Solaris gcc | Bare | `dbg_gcc` |
| Model Verifier | True | Solaris gcc | Bare | `dbg_gcc_cpp` |
| Model Verifier | False | Win32 | Bare | `dbg_cl` |
| Model Verifier | True | Win32 | Bare | `dbg_cl_cpp` |
| Model Verifier | False | Win32-gcc [a] | Bare | `dbg_cyg` |
| Model Verifier | True | Win32-gcc [b] | Bare | `dbg_cyg_cpp` |

a. See restriction regarding use of Target kind with Model Verifier
b. See restriction regarding use of Target kind with Model Verifier

**See also**

Predefined Stereotypes and Attributes

## Library files

This section describes the contents of the directories where the libraries are implemented. The following topics are covered:

- The file `comp.opt`
- The files `makeoptions / make.opt`
- The file `sccd.cfg`
- The `makefile`
- The make template file

### comp.opt

This file determines the details of the generated makefiles, 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) 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, $(sctCODER_OBJS).
%a : the archive file, libstcoder$(sctLIBEXTENSION)
```

**Example 353: 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
```

## makeoptions / make.opt

This file has the following structure:

**Example 354: make.opt on UNIX**

```
# #
sctLIBNAME     = Simulation
sctIFDEF       = -DSCTDEBCOM
sctEXTENSION   = _smd.sct
sctOEXTENSION  = _smd.o
sctLIBEXTENSION=  _smd.a
sctKERNEL      = $(sctdir)/../INCLUDE
sctCODERDIR    = $(sctdir)/../coder

#Compiling, linking
#Take advantage of the C Compiler Driver
sctSCCD        =
sctCC          = $(sctSCCD) cc
sctCODERFLAGS  = -I$(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 libraries
# The following names of importance are defined:

sctLINKKERNEL  =
sctLINKKERNELDEP =
sctLINKCODERLIB =
sctLINKCODERLIBDEP =
```

The information to the right of the equal signs should be seen as an example. The environment variables set in the `makeoptions` (`make.opt`) file should specify the following:

- `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 C Code Generator, 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 or library.
- `sctKERNEL`. Directory of the run-time library source code.
- `sctCODERDIR`. The directory for the source code of the coders or decoders.
- `sctCC`. This defines the compiler to be used.
- `sctCODERFLAGS`. Compilation options needed to compile the coder or 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` and `sctpred.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.

- `sctUSERDEFS`. In make template files the `sctUSERDEFS` flag can be used for example to set extra include paths for the compilation. This variable is invoked with `sctCPPFLAGS` in the standard kernels.
- `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 library source files. It will be used in the link command in the generated makefile.
- `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 makefile.
- `sctLINKCODERLIBDEP`. Used to implement the dependencies to recompile the coder library when it is needed.

### sccd.cfg

This is the CCD Configuration File for the C Compiler Driver utility, which can be used to expand C macros and make the generated C code easier to read and suitable for debugging on C level.

### makefile

The generation and execution of the makefile is automatically handled by the C Code Generator and Application Builder. The content of the makefile is best shown by an example.

**Example 355: generated makefile on UNIX, for a system named "example" ——**

```
# 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)
```

When "making" such a generated makefile for your system, the following sequence of actions takes place:

3. The `makeoptions` (`make.opt`) file is included in the directory referenced by the environment variable `sctdir`.
4. "Make" will then use the variables `sctIFDEF`, `sctLINKKERNEL`, `sctCC`, `sctCPPFLAGS`, `sctCCFLAGS`, `sctLD`, and `sctLDFLAGS` to compile and link the application code generated by the C Code Generator (in this case `example.c` in Example 355 on page 1068)
5. In the code Generated by the C Code Generator, `#include` statements are provided to include the `.c` and `.h` files that are present in the include directory, at the suitable place in the generated code. These files, that among other implement the run-time kernel, will hence become automatically compiled and linked with the application code.

**Note**

*To guarantee the congruency between the code that is generated for the system and the run-time kernel, the* `makeoptions (make.opt)` *file is used when generating the makefile compiling the C files included in the library and the generated application code. Non-congruency in this sense between the run-time library and the application code will make the result unpredictable!*

### make template file

A make template file is useful to easily have external code compiled and linked with the application code and run-time library. Such code could be the environment functions or external libraries. The make template file should follow the syntax adopted by the current "Make" used on your computer.

`USERTARGET` is a "hook" that is used to specify additional make targets that should be added to the makefile that is generated by the C Code Generator.

`USERLIBRARIES` is used to specify additional libraries that should be linked into the application.

The use of the Make template file is controlled via the code generator stereotype properties of the Build Artifact.

**Example 356: A make template file (using nmake)**

Consider the environment functions in the files `example_env.c` and `example_env.h`

The makefile to use as template would then look like this:

```
USERTARGET = C:\example\example_env$(sctOEXTENSION)
USERLIBRARIES = -lm

C:\example\example_env$(sctOEXTENSION):
C:\example\example_env.c C:\example\example_env.h
   .$(sctCC) @<<
   .$(sctCPPFLAGS) $(sctCCFLAGS)
   .$(sctIFDEF)
   ./FoC:\example\example_env$(sctOEXTENSION)
   .C:\example\example_env.c
<<
```

## Included source and header files

The code and definitions for the runtime libraries used by the C Code Generator is available in a number of files. All of these files can be found in the directory `...\sdlkernels\include\` and their contents is briefly described in the subsections below.

### sctda.c

This file contains the functions that implement the command interpreter and command executor for the requests sent by the Model Verifier.

### sctadacom.c

This file implements the communication layer used for the interaction between the Model Verifier and the application. It is uses TCP/IP sockets for this implementation.

### sctadacom.h

This header file contains definitions used by `sctadacom.c`

### sctdamsg.c

This file implements the message layer used for the interaction between the Model Verifier and the application. It is built on top of the communication layer, which is implemented in `sctadacom.c`

### sctdamsg.h

This header file contains definitions used by `sctdamsg.c`

### sctdamsgcode.h

This header file contains definitions used by `sctdamsg.c`

### sctlocal.h

This file contains type definitions and `extern` declarations of variables and functions that are used only in the run-time kernel. This file is not included in the generated code.

### sctos.c

In this file are placed functions that represent dependencies to hardware, operating system and compilers.

The functions required for an application to work on target are:

- A function to read the clock
- A function to allocate memory.

### sctpred.c

The functions implementing the operations defined in the predefined data types can be found in this file.

**sctpred.h**

This file contains type definitions and `extern` declarations handling the predefined data types (except Pid, which is in `scttypes.h`). This file is included in generated code via `scttypes.h`.

**sctsdl.c**

In this file the implementation of operations can be found, together with the functions used for scheduling. This file contains among others functions for:

- Handling and reporting dynamic errors
- Operations, such as signal sending, create, stop, nextstate, set, reset, together with help functions for these activities
- Initialization and the main loop (the scheduler).

**scttypes.h**

This file contains type definitions and `extern` declarations of variables and functions. This file is included by `sctsdl.c, sctpred.c, sctutil.c, sctda.c, sctos.c`, and by each C file generated by the C Code Generator. This file should normally be included in any user-written environment file to allow use of data types, operations and signal primitives.

**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.

To move a generated C program plus the run-time library to a new target platform (including a new compiler), the major changes are to be made in this file. You must also write a new section in `scttypes.h` to describe the properties of the new compiler.

# Creating user-defined (customized) libraries

Users may want to create new libraries, which will allow to give the generated code customized properties at compile and link time. The process of creating a new library is not supported in the graphical framework. It involves

using suitable tools available on the host computer (shell, text editor, make, compiler, linker...) and running the application to verify that the behavior is the expected one.

The process should be done in the following sequence.

1. Start by copying the contents of one of the existing directories with supported libraries (`appl_cl`, `applt_cl`...), or any other library that you have created previously. You may want to start with a library which already has properties that are close to the intended effect, to reduce the amount of work in the next steps.
2. Next, modify the contents of the file `comp.opt` and verify that executing the file results in the expected results:
   - That the correct `makeoptions / make.opt` file is included
   - That the correct compile script is used
   - That the correct link script is used
   - That the correct command to run make is used
   - That coder libraries are correctly built.
3. Next, modify the contents of the file `makeoptions / make.opt` and modify the environment variables, C macros and compiler flags to adequate values.
   - A reference to the C macros that are used when preprocessing the generated C code and the definitions in the `include` directory is available in Chapter 38, C Code Generator Macros.
   - The section that defines the creation of coders and archives is usually not needed to modify.
4. Should you want, you may also want to customize the C Compiler Driver (optional).
5. Build the application using the `comp.opt` and `makeoptions / make.opt` files, and test that it behaves as expected.

**Note**

*If you create new versions of the library, make sure that the included files and the generated code are both compiled with the same compilation switches. If not, you will experience unexpected or undefined behavior in the compiled application!*

# Adaptation to Compilers

This section covers how to change the source code to adapt it to a new environment. This could mean moving the code to new target hardware and OS, or using a new compiler.

There are two parts of the source code that might need changes:

- In `scttypes.h` there is a section defining the properties of different compilers, where a new compiler can be added.
- 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.

## Compiler definition section in scttypes.h

In `scttypes.h` the properties of the compiler is recognized by the compiler or 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, you must configure it yourself by writing a file `user_cc.h`, which can be placed in the Target Directory.

The compilers above are:

| Compiler | Description |
|---|---|
| __linux | gcc on Linux |
| __sun | different compilers on SUN |
| __hpux | different compilers on HP |
| __CYGWIN__ | gcc on windows, for more information go to `http://sources.redhat.com/cygwin/` |
| QNX4_CC | QNX |
| __BORLANDC__ | Borland C/C++ compiler on Windows |
| _MSC_VER | Microsoft Visual C/C++ 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 (ULONG_MAX != UINT_MAX)
```

```
#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.

## Modifications in the file sctos.c

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 macros, the corresponding function or functions are removed from `sctos.o` and have to be supplied by the user instead:

```
XUSER_ALLOC_FUNC
XUSER_FREE_FUNC
XUSER_HALT_FUNC
XUSER_CLOCK_FUNC
XUSER_SLEEP_FUNC
XUSER_KEYBOARD_FUNC
```

### xAlloc

The function `xAlloc` is used to allocate dynamic memory and is used throughout the run-time 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`.

If you want to estimate the need for dynamic memory you may introduce statements in `xAlloc` to record the number of calls and also the total requested size of dynamic memory. A few things should be noted.

- A program compiled to become a Model Verifier will require more dynamic memory than a program compiled to become an application, so estimates should be made while using the appropriate compilation switches.
- A call of `calloc` will actually allocate more memory than is requested, in order to make it possible for the C run-time system to deallocate and reuse memory. The size of this additional memory is compiler dependent.
- If you want to handle the case when no more memory is available at an allocation request you 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 print an appropriate message before terminating.

### 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 if no other use of dynamic data has been introduced by you, this function will not be used.

The parameter of the `xFree` function is the address of the pointer to the allocated memory.

**Example 357 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`.

### SDL_Clock

The function `SDL_Clock` returns the current time, read from a clock somewhere in the OS or hardware. The return value is of type SDL_Time. 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, 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.

**Note**

*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 likely cause a severe system malfunction.*

### SDL_Time

`SDL_Time` 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;
```

`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.

### xSleep_Until

The function `xSleep_Until` is given a time value, of type SDL_Time and should suspend the executing until this time is reached. Then it should return.

This function is used only when real time is used (the switch XCLOCK is defined) and when there are 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 of the system that should be part of a (larger) distributed application. If environment functions are available for the system this function should be implemented there.

### 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 Model Verifier (when `XMONITOR` is defined).

The `xCheckForKeyboardInput` function is used to implement the possibility to interrupt the execution of state transitions by typing `<Return>` and to handle the polling of the environment when the program is waiting at the command prompt in the Model Verifier.

# 34

# *Dynamic Memory Management in C Code Generator*

This section describes how to allocate and de-allocate dynamic memory in the code produced by the C Code Generator. It explains how dynamic memory can be reused using de-allocation and avail lists and how to estimate the total need of dynamic memory for an application.

**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.*

# Dynamic Memory Size Requirements

Dynamic memory is used for a number of objects in a run-time model for applications generated by the C Code Generator. These objects are:

- Instances of active classes
- Signal and timer instances
- Instances of operations in active classes
- `Charstring`, `OctetString`, `BitString`, and `ObjectIdentifier` attributes
- Attributes of `String`, `Bag`, general `Array`, and general `PowerSet` types.
- Attributes of other user-defined datatypes, where you have decided to use dynamic memory.

To help to estimate the need for memory for an application, information about the size of these instances and the number of created instances is needed. The size information given is true for generated applications, that is, applications that do not, for example, contain the Model Verifier command-line interpreter. 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`.

## Active classes

Each instance of an active class is represented by two structures that will be allocated on the heap. In `scttypes.h` the type `xLocalPIdRec` is defined and in generated code `yVDef_ProcessName` structures are defined:

```
typedef struct {
   xPrsNode      PrsP;
}  xLocalPIdRec;

typedef struct {
   xPrsNode       Pre;
   xPrsNode       Suc;
   int            RestartAddress;
   xPrdNode       ActivePrd;
   void (*RestartPAD) (xPrsNode  VarP);
   xPrsNode       NextPrs;
   SDL_PId        Self;
   xPrsIdNode     InstNameNode;
   xPrsIdNode     TypeNameNode;
   int            State;
   xSignalNode    Signal;
```

```
    xInputPortRec  InputPort;
    SDL_PId        Parent;
    SDL_PId        Offspring;
    int            BlockInstNumber;
    xSignalIdNode  pREPLY_Waited_For;
    xSignalNode    pREPLY_Signal;
/* parameters and attributes of active classes */
} yVDef_ProcessName;
```

To calculate the size of the structures above it is necessary to know more about the components in the structures. 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:

$$\text{Size}_{\texttt{xLocalPIdRec}} = \text{Size}_{\text{address}}$$

$$\text{Size}_{\texttt{xPrsRec}} = 15 \cdot \text{Size}_{\text{address}} + 7 \cdot \text{Size}_{\texttt{int}}$$

The size of `yVDef_ProcessName` is the size of the `xPrsRec` plus the size of the attributes and parameters in the active class. Any overhead introduced by the C system should also be added. The size of the parameters of the state machine and its attributes is dependent on the declarations in the active class.

For each instance set of an active class in the system the following number of structures of a different kind will be allocated:

- There will be one `xLocalPIdRec` for each instance created. These structures will not be reused, as they serve as identification of instances of active classes that have existed.
- There will be as many `yVDef_ProcessName` structures as the maximum concurrently executing instances of the instance set of the active class (maximum number during the complete execution of the program).

The `yVDef_ProcessName` structures are reused by having an avail list. Such a struct is placed in the avail list when the instance of the active class it represents performs a stop action. There is one avail list for each type of active class. When an instance should be created, the run-time 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.

**Note**

*If the compilation switch* `XPRSOPT` *is defined then,* `xLocalPIdRec` *is reused together with the* `xPrsRec. xLocalPIdRec` *contains an additional* `int` *component.*

## Signals

Signals are handled in much the same way as active classes. A signal instance is represented by one struct:

```
typedef struct {
   xSignalNode   Pre;
   xSignalNode   Suc;
   int           Prio;
   SDL_PId       Receiver;
   SDL_PId       Sender;
   xIdNode       NameNode;

   /* Signal parameters */
} ySignalPar_SignalName;
```

This struct type contains one component for each signal parameter. The component types will be the translated versions of the types of the parameters.

This means that it is possible to 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 `ySignalPar_SignalName` struct is thus equal to the size of the `xSignalRec` plus the size of the parameters.

For each signal type in the system the following number of data areas will be allocated:

- There will be as many `ySignalPar_SignalName` structures as the maximum number of signals (during the complete execution of the program) of the signal type that are sent but not yet received.

The `ySignalPar_SignalName` structures are reused by having an avail list, where such structures are 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 in-

stance should be created, for example during a signal sending operation, the run-time 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, but taking into account that each timer also contains an additional SDL_Time component.

## Operations in active classes

Active classes and operations in active classes have much in common in terms of memory allocation. An operation in an active class 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 attributes */
} yVDef_ProcedureName;
```

The struct type contains one component for each formal parameter or attribute. The component types will be the translated version of the 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 an operation without attributes 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 attributes defined in the operation.

For each type of operation in the system the following number of data areas will be allocated:

- There will be as many `yVDef_ProcedureName` structures as the maximum number of concurrent calls (during the complete execution of the program) of the operation. Concurrent calls occur both when an operation calls itself recursively within one instance of an active class, and when several instances of the same active class calls the same operation during overlapping times.

The `yVDef_ProcedureName` struct is reused by having an avail list, where this struct is placed when the operation instance executes a return action. There is one avail list for each operation type. When an instance of an operation should be created, that is, at a call operation, the run-time 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.

## Predefined data types

The predefined type Charstring is implemented as `char *` in C and thus requires dynamic memory allocation. The predefined data types `BitString`, `OctetString`, and `ObjectIdentifier` are also implemented using dynamic memory.

The implementation of the sorts `Charstring`, `BitString`, `OctetString`, and `ObjectIdentifier` 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.

`Charstring`, `BitString`, `OctetString`, and `ObjectIdentifier` are also handled correctly if they are part of structures 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.

# Implementation of Memory Management

The allocation and de-allocation 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. 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.

The `sctos.c` file which is described in detail in "Modifications in the file sctos.c" on page 1076 in Chapter 33, *C and AgileC Runtime Libraries*.

## Functions for allocation and de-allocation

### xAlloc

```
void* xAlloc(xptrint Size)
```

The function `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

```
void xFree(void** P)
```

The function `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 zero**.

### Implementation aspects

The `xAlloc` and `xFree` functions are usually implemented using some version of the C standard functions for allocation (`malloc`, `calloc`) and de-allocation (`free`). If the default implementation in `sctos.c` is not sufficient for your needs, other implementations can be supplied, as long as the interface is fulfilled.

**Note**

*If you provide an alternative implementation, then make sure that your own xAlloc sets data to zero before returning the address to the data, and that xFree sets the pointer to zero before returning!*

### Memory Fragmentation

Memory fragmentation is a phenomenon occurring when a program allocates and de-allocates data areas (of different sizes) in some "random fashion". Then, small chunks of memory here and there are lost, since their sizes are too small to fit an allocation request. This can lead to a slowly increasing demand for memory for the application.

To prevent memory fragmentation avail lists, which do not require de-allocation of memory to be implemented, are used in almost all circumstances.

De-allocation of memory is only applicable for data types, and is used for variables of the following types:

- `Charstring`,
- `BitString`,
- `OctetString`,
- `ObjectIdentifier`,
- Types created by `String` (not #STRING) and `Bag` template
- Types created by `Array` template, if the index type is such that an array in C cannot be used. (General array)
- Types created by `PowerSet` template, if the component type has the same property as for the index type in general arrays.

This means that if attributes of the above mentioned types are not used, and you have not introduced the need for de-allocation of memory, no memory de-allocation will occur at all. In this case it is unnecessary to implement the `xFree` function.

### Trace of memory needs

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.

# 35

# C Code Generator Reference

This chapter is a reference manual to the C Code Generator and describes the principles that govern the code generation from UML to C. Notably, the following is described:

- Operation principles
- Implementation of run-time semantics (time, scheduling, signalling...)
- Translation of the predefined data types in UML
- Translation of Tau proprietary data type extensions to the predefined UML datatypes
- Passing of parameters to operations
- Generic functions on data types
- Names in generated code.

# C Code Generator Operation Principles

The C Code Generator is used by the build types **C Code Generator** and **Model Verifier**.

**See also**

"Using Build Types" on page 843 in Chapter 26, *Building and Code Generation Overview and Examples*

## C Code Generator options and settings

The C Code Generator takes advantage of Using Build Artifacts for the settings that are global to the application being built. An example of such a setting is specifying the target and the run-time library to compile and link with the generated code.

The build type C Code Generator also allows you to apply options to individual elements in the model, such as excluding model elements from a build, and solving naming and language related issues that arise when including declarations and definitions of external C and C++ code.

## Launch of C Code Generator

### Interactive mode

To launch the C Code Generator from the graphical user interface, the following sequence of actions should take place:

1. Create a Build Artifact that contains appropriate settings for the current build.
   - The stereotype «C Code Generator» contains the attributes that define the settings for the C Code Generator.
2. Apply a Build Root to the build artifact. A build root designates the top-level active class in the model to build.
3. If needed, tag any element that should be excluded from the build, and apply suitable naming and language related attributes to the elements that are defined in external C or C++ code.
   - Use the stereotype «C Application» for this.
4. Select the **C Code Generator** build type for the build artifact.

5. To generate the code:
   – Right click the build artifact, and select **Build** on the menu.

As a result of the actions above, the following operations take place without need for human intervention:

1. The part of the model that is specified as Build Root is transformed to an intermediate representation. The internal representation is expressed in an SDL-like syntax.
   – The elements that have the attribute Generate C code set to false are discarded.
   – Including external declarations and subsequent naming issues is handled by settings that are available in the «C Application» stereotype.
2. The exported internal representation is checked by the C Code Generator to be semantically correct, before C code is generated.
3. The C Code Generator generates C code
   – Code generation is taking into account the settings specified for the build.
   – The code generation translation rules are described in main sections in this document.
4. The generated C code is written on files in the Target directory.
5. A makefile is created by the C Code Generator. This makefile ensures that the generated code and run-time library that has been specified in the build artifact are properly compiled and linked. External code can also be added to the compile and link scheme, by specifying a make template file in the build artifact.
   – These settings are managed by the stereotype «C Code Generator»
6. The generated makefile is executed as the final step in the build.

### Batch mode

The C Code Generator can be operated in batch (command-line) mode, obeying the same principles and with the same variety of options as for an interactive build. The only difference is that messages are written on `stdout` instead of on the output message area.

To launch the C Code Generator from the command line prompt, the `taubatch` command is used.

**See also**

"Supported SDL" on page 633 in Chapter 17, *SDL Import*

"Interactive Build Interface" on page 932 in Chapter 27, *Building Applications Reference*

"Batch Build Interface" on page 945 in Chapter 27, *Building Applications Reference*

# Implementation of Run-Time Semantics

## Time

An application generated by the C Code Generator 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 simulation/debugging sessions using the Model Verifier, means that the time in the simulation has no connection with the real time. Instead the discrete event simulation technique is used. This technique is based on the idea that the current value for the simulation time (Now) 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 are transitions in state machines, 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, then the timer output will occur immediately. The simulation time will be increased by one hour, but you do not have to wait one hour.

### Real time

If the **Real-time** option for Simulation kind is used, then there will be a connection between the clock in the executing program and the real time. If you give a Go command in the example above, you 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.

If the next transition is a timer output scheduled in the future, (more than a second from now), the command Next Transition will run the simulation for one second and then pause.

## Scheduling

The behavior of the active classes in the system is implemented by state machines, which execute transitions that consist of actions like actions, decisions, signal sending, calls of operations, etc. 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 a signal sending operation occurs.

A transition is always executed without any preemption, unless you are debugging the application and manually rearrange the ready queue (using an appropriate command provided by the Model Verifier) and then execute another transition. The interrupted transition can afterwards be executed to its end.

UML 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, the Model Verifier, which also obeys this execution strategy, is not directly suited for simulation of "timing effects", that is, situations where the time or order of actions in different instances of active classes is of vital importance.

**Example 358: Scheduling a hazardous situation**

Suppose an instance of an active class, say A, that sends two signal instances during the same transition, one to an instance of an active class, B, and one to an instance, C. During the corresponding transitions to B and C, a signal instance is sent to an instance of an active class, D.

If the behavior of the application is dependent on the order in which the signal instances are received in the input port of D, this is a hazardous situation where the execution speed of the instances of active classes and any delay of signals will determine the behavior.

---

## The ready queue

The ready queue contains all the instances of active classes that have received a signal that can cause a transition, but have not yet completed that transition. The ready queue is ordered according to priority and the insert time.

The priority of an active class is the priority of the signal that will cause its next transition. The ready queue is first sorted according to priority (higher signal priority value = lower priority) and then according to insert time.

### Ready queue priority

An instance of an active class will be inserted last among the instances with the same priority. An instance of an active class will never be inserted before the instance that is currently executing, as preemptive scheduling is not used.

- If an instance of an active class sends a signal to another instance, which immediately can receive the signal, the receiving instance will be inserted into the ready queue last among the instances of active classes with the same priority, but never before the currently executing instance.
- If the state machine that implements the behavior of an instance of an active class is currently executing a nextstate, immediately can continue to execute another transition, then the instance will be inserted last into the ready queue among the instances with the same priority. This means that it can remain as first instance in the ready queue.
- If the state machine receiving a timer output immediately can execute a transition as response to the received signal, the instance will be inserted last.

## Public attributes

The translation scheme used by the C Code Generator for a public attribute is to make direct access to the memory area for the attribute. The address of the attribute is calculated by a utility function in the kernel and this address is then directly used to access or change the public attribute.

**Note**

*This scheme does not work in a threaded application to access attributes in other threads. Inside a thread there are no problems. Either the underlying OS might prevent accesses of memory between threads, or there is a risk for accessing the memory at the same time from different threads which will cause problems. In a threaded application it is recommended that sharing a variable is performed using a set and a get operation. In that way the accesses and updates of the shared variable will be synchronized and update problems are prevented.*

## Guards and guards on triggered transitions

Guard and **guard on triggered transition** are additional concepts in UML. One model for these concepts could be to use repetitive signal sending, to have the expressions recalculated repeatedly. This model is however not suitable during simulation or debugging sessions, and definitely not acceptable in an application. Therefore it is used an implementation strategy closer to the described behavior of the concepts.

### Implementation

First, distinguish between those guards and guards on triggered transitions that are dynamic and those that are static.

- Static guards contain expressions with a given value, and do not provide any implementation problems or any execution overhead, except that the corresponding expressions have to be calculated at nextstate operations.
- Dynamic guards contain expressions that can change their value when the corresponding state machine is waiting in the state. The expression contains a part that can change its value, even though the state machine does not execute any statements. an example of this is when using public attributes or Now in a guard expression.

Dynamic guards have to repeatedly be recalculated. The strategy selected for these expressions is to recalculate them after each transition or timer output performed by any state machine, and additionally also before the Model Verifier command line interpreter is entered within a transition.

Each state machine waiting in a state containing a dynamic guard executes an implicit nextstate operation between each transition or timer output performed by other state machines.

## Constant attribute

A Constant or read-only attribute is implemented either as a C macro (`#define`) or as a C variable.

- To be translated to a macro, the following must apply for the expression defining the value of the attribute:
  - It must be of one of the predefined sorts (Integer, Real etc.).
  - It may only contain literals and operations defined in the predefined sorts and other constant attributes that are possible to calculate when generating the code.
- All other constant attributes in packages are implemented as C variables given their values at program start up.

The reason for raising this question is because it is relevant to the implementation of Array and PowerSet. There are two different implementations for each of these concepts.

- An array 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 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 are used, otherwise the linked list is used.

If a constant attribute that is translated to a C 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, then 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 constant attributes

External constant attributes can be used to parameterize a system and thereby also a generated program. The values that should be used for the external constant attributes can either be read by the generated program during start up, or included as macro definitions into the generated code. The C Code Generator can handle both these cases – it is not necessary to select which way should be used for each attribute until the program is compiled.

### Using a C macro definition for specifying the value of attribute

To use a macro definition in C to specify the value of an attribute in package, perform the following steps:

1. Declare the attributes to be external, and write the corresponding macro definitions to a file.

**Example 359: Macro definitions**

UML package declarations of external constant attributes:

```
const Integer extern attribute1;
const Real extern attribute2;
```

The attribute names are the UML names without any prefixes and with all characters that are not letters, digits or underscores removed.

```
#define attribute1 3
#define attribute2 3.14
```

It is up to the user to ensure that the defined values (`3`, `3.14`) are compatible with the type (`Integer`, `Real`) of the attributes.

---

2. Select the package that the attribute(s) belong to.
3. Open the properties editor and select C Application in the **Filter** drop-down menu. If **C Application** is not present it should be activated with the Customize dialog, from the Add-Ins tab.
4. Enter the name and path of the file with the macro definitions in the Include File field. The path can be either an absolute path or a path relative to the Target Directory.

**Note**

*When an application is created, macro definitions should be used for all attributes in packages, as the function for reading attribute values stored on file is not available.*

### Reading values of attributes at program start up

The other way to supply values of external constant attributes in packages is to read the values at program start up. If there are any external constant attributes that do not have a corresponding macro definition, it is possible to choose between supplying the values of the remaining attributes from the keyboard or to use a file containing the values.

When the application is started, the following prompt appears in the Model Verifier:

```
External file :
```

- Press `<Return>` in the Model Verifier Console 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 you choose to read the values from the terminal, you will be prompted for each value. In the other case you should have created a file containing the attribute names and their corresponding value according the following example:

**Example 360: Values at program startup**

```
attribute1 value1
attribute2 value2
```

The attributes may be defined in any order.

## Value returning operation call

In the C Code Generator, value returning operation calls (in datatypes, passive classes or active classes) are implemented by inserting an extra call just before the statement containing the value returning operation call. The result from the call is stored in an anonymous variable, which is then used in the expression.

**Note**

*The value returning operation calls are transformed to ordinary calls, by adding a new IN/OUT parameter for the operation result, last in the call.*

## Arbitrary value operator (any)

There are two different applications of `any`. It is possible to write

- `any` in a decision
- `any (SortName)` within an expression.

**Note**

*The* `any` *operator should be used only for simulation purpose (with the Model Verifier), not for applications generated by the C Code Generator!*

### any in decision

any in a decision should only be used for debugging and simulation purposes, and is implemented by a question giving you a possibility to select the path to follow.

### any (SortName) in expression

`any (SortName)` within an expression, is implemented using a random number generator to draw a random number of the given type.

**Note**

*The operation call* `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.*

**See also**

"Arbitrary value (any) expression" on page 360 in Chapter 8, *UML Language Guide*

# Translation of Data Types

## General

### Implementation of C definitions

The implementation for the C types and C macros supplied by the run-time library used by the C Code Generator for the predefined and Tau proprietary extensions to UML are found in the file `sctpred.h`, except for the Pid sort which is handled in the file `scttypes.h`.

In all examples in the following sub-sections, the prefixes, which are added by the C Code Generator to the names in C, are not shown. (These prefixes are added to make sure that no name conflicts occur in the generated program. For more information about prefixes and suffixes

**See also**

"Names in Generated C Code" on page 1121 in Chapter 35, *C Code Generator Reference*.)

### Initial values

Initial values will be assigned to all attributes which do not have an initial value specified in UML, such attributes will be set to 0 by using a `memset` to 0.

**Note**

*The C Code Generator does not permit naming of literals using name class literals or character strings.*

## CPtr

This template, as well as the templates Own and ORef, represent pointers with different properties. They are all translated to pointers in C.

CPtr has the following operations (from the definition):

```
public <<External="true">> void SetValue( T);
public <<External="true">> T GetValue();
template<type T1> public static <<External="true">> T1 GetValue(CPtr<T1>);
template <type T1>public static <<External="true">> CPtr<T1> GetAddress(T1
entity);
public <<External="true">> T '[]'(CPtr<T>, Integer);
public CPtr<T> '\+'(CPtr<T>, Integer);
```

```
public CPtr<T> '\-'(CPtr<T>, Integer);
public static <<External="true">> void free(CPtr<T>);
```

Supported only for `CPtr<char>` or `char*`:

```
template<type T1> public static <<External="true">> CPtr<T1> malloc(Integer);
```

## Array

Instantiations of the array template are handled by the C Code Generator with the following restriction: The component and index sort may be any sorts that the C Code Generator can handle, but must not directly or indirectly refer to the array type itself (see also the section describing the handling of Struct.

If the index sort is a discrete sort, with one closed interval of value, then the UML array is translated to a struct containing an element which is an array in C. The discrete sorts are:

- Character
- Boolean
- Octet
- Bit
- A sort that is considered as an enumeration type
- Syntype of any Integer, Character, Boolean, Octet, Bit or enumeration type. The subtypes may only have one range condition that specifies a closed interval of values.

If the index sort is not one of the sorts in the list above, a UML 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 361: Array**

```
syntype MyInt= Integer constants( 1..10 );
  class Arr {
      Array <MyInt, Real> MyArray;
 }
```

is translated to:

```
typedef SDL_Integer MyInt;
typedef struct {
 SDL_Real A[10];
```

```
} Arr;
```

## Bag

The Bag template is similar to PowerSet. However, bags may contain several elements with the same value. A bag is translated to 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.

## Charstring

The Charstring type is typically used to represent text strings. A Charstring is represented as a `char*` in C.

**Note**

*For Charstring types, an extra character is inserted by the code generator at index 0. This character is used internally by the run-time system and should be disregarded from when addressing Charstring variables in C at application level.*

## Choice

Choice is used to express a union with implicit tag.

**Example 362: Choice**

```
choice Str {
  Integer a;
  Boolean b;
  Real c;
}
```

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 `Present` component is automatically set by the C Code Generator when a component in the choice is given a value.

**Note**

*During simulations and debugging sessions with the Model Verifier, it is tested at execution time that a component "is present" when an attempt is made to access it. A run-time error message is printed if this is not the case.*

## Enum

A sort containing a literal list, is seen as an enumeration type. See Example 363 on page 1103. Such a type is translated to `int`, together with a list of 'defines' where the literals are 0, 1, 2, and so on.

**Example 363: Enumeration type**

```
enum EnumType {Lit1, Lit2, Lit3}
```

is translated to:

```
typedef XENUM_TYPE EnumType;
#define Lit1 0
#define Lit2 1
#define Lit3 2
```

The macro `XENUM_TYPE` is defined in the file `sctpred.c` as:

```
#ifndef XENUM_TYPE
#define XENUM_TYPE int
#endif
```

This means that all enum types will become `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`.

## ORef

This template, as well as the template Own, represent pointers with different properties. They are all translated to pointers in C.

## Own

See ORef.

## PowerSet

Instantiations of PowerSet is handled by the C Code Generator with the following restriction: The component sort may be any sorts that the C Code Generator can handle, but must not directly or indirectly refer to the PowerSet type itself.

There are two translation schemes for PowerSet. If the component sort fulfills the conditions for index sorts mentioned in "Array" on page 1101, an array of 32-bit integers is 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 members of the set, is used to represent the PowerSet.

## String

Instantiations of String are handled by the C Code Generator with the following restriction: The component sort may be any sort that the C Code Generator can handle, but must not directly or indirectly refer to the String type itself.

Strings are translated to linked lists containing one element for each element in the string value.

## Struct

A passive class is translated to a `struct` in C, as can be seen in Example 364 on page 1104.

**Example 364: Struct**

```
class Str {
 Integer a;
 Boolean b;
 Real c;
}
```

is translated to:

```
typedef struct {
  SDL_Integer a;
  SDL_Boolean b;
  SDL_Real c;
} Str;
```

The components of a struct may be of any sort that the C 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 no longer be valid.

### Syntype

A syntype can be defined for any sort that the C Code Generator can handle, giving a new name for 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 syntype attribute 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 an attribute is handled as a dynamic error in run-time. If the syntype is 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.

## Parameter Passing to Operations

For performance reasons the data types have been divided into two groups:

- Simple, small types that are passed as values
- Structured, larger types that are passed as references (addresses).

### Types passed as values

The following types are 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 template Own, ORef.

Passed as values are also:

- Any syntype of a type in the list above
- Types that inherit a type in this list.

## Types passed as addresses

The following types are passed as addresses (structured types):

- `BitString`
- `OctetString`
- `ObjectIdentifier`
- `Struct` types
- `Choice` types
- Instantiations of template `PowerSet`
- Instantiations of template `Bag`
- Instantiations of template `Array`
- Instantiations of template `String`.

Passed as addresses are also:

- Any syntype of a type in the list above
- Types that inherit a type in this list.

**Note**

*For types represented as pointers (Charstring, including any syntype of Charstring, Own, ORef), the pointers, not the addresses of the pointers, are passed as parameters.*

## Parameter passing

The parameter passing for operations implemented in C works as follows:

### In parameters

- Passed as a value in C if the type is listed in "Types passed as values" on page 1105. This means that the parameter type in C is the same type as in UML.
- Passed as an address in C if the type is listed in "Types passed as addresses" on page 1106. This means that the C parameter is `(UML_type *)` if the type in UML is `(UML_type)`.

### In/Out parameters

- Parameters are always passed as addresses, i.e the C parameter is `(UML_type *)` if the type in UML is `(UML_type)`.

### Operation result

If the type of the result of the operation is listed in "Types passed as values" on page 1105, the C function result type will be the same as in UML.

If the result type is listed in "Types passed as addresses" on page 1106, two things are changed.

- The C result type will be `(UML_type *)`, i.e the result will be an address.
- An extra parameter is inserted last in the C function. This parameter is also of type `(UML_type *)` and is used as a location to store the result of the function. At a call of an operation, a "dummy" variable should be passed as the actual parameter. The C function can then use this to store the result of the operation and should return the variable again as result.

**Example 365:** ———————————————————————————

Consider a struct datatype, struct1.

```
class c {
  int X;
```

```
  struct1 Y;
}
```

The C prototypes for these operations 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**

*As IN parameters are passed as addresses for structured types, changing such a parameter inside the operation might have undesired effects. An attribute passed as an actual parameter is then also changed. If you want to change the formal parameter copy it first to a local attribute.*

# 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, the size of the type, the component types for structured types and the 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.

For a detailed reference to the contents of the type info nodes, see "Type definitions of type info nodes" on page 1182 in Chapter 37, *C Code Generator Symbol Table*.

Every passive class or syntype introduced in UML 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 UML has access to an assignment macro `yAssF_typename`.

**Example 366: 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 run-time library) 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 UML.

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 the following tasks:

- 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.
- 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.
- 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 template 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 *);
```

A `GenericAssignSort` function must consider the following questions in order to handle the objects correctly.

- How should the object be copied?
- What is the status of the newly created object?
- What should be done with the old value referenced by the left hand side variable?

These topics will be discussed below.

### Copy of objects

This operation is important to consider, since 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.

### Status of new objects

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 exist:

- **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.

- **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 operation.

**Old value**

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 has access to an equal macro `yEqF_typename` and a 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.

### Charstring and Own

Special treatment of Charstring and instantiations of the Own template 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 that is implemented as a pointer, or that contains a pointer that references to memory that is automatically handled, 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

The Make operation is available for the struct type, the ObjectIdentifier type and the instantiations of the templates 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, Own, ORef, and any syntype 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 initializer, 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 template Own.

- 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 templates `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.

### Copy

An implicit copy operator has been inserted for every user-defined type. It takes a value and returns a copy of that value. For all types that are not Own pointers or contain Own pointers, this operator is meaningless as it just returns the same value. For Own pointers or for structured values containing Own pointers, the copy function, however, copies the values referenced by the Own pointers.

The implicit copy operator is only implemented for user-defined types. For predefined data types the copy operator is not implemented as it is not meaningful.

# Generic Functions for Operations in Predefined Templates

The generic function for the operations in the predefined templates follow the general rules for operations with a few exceptions:

- A type info node is needed as a parameter, as the C function can handle all instantiations of a certain template.
- Parameters of template 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 PowerSet 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 Bag and PowerSet 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. `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 definition of parameters.

## Limited string

Generic functions are available for limited strings. Limited strings are strings that use a maximum 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 definition of parameters.

# Optimizations

## Removing unused operations

When implementing a system, you do not always use all available UML operations. The C Code Generator removes the declarations of unused operations, thus minimizing the code size of the generated application. Unused operations that are removed are:

- Operations in predefined data types, for example substring, concatenate, calculate length on Charstring, etc.
- Operations defined in the predefined templates String, Array, PowerSet, Bag
- Special operations (and help functions) like assign, equal, default, make, extract, modify, free.

The C Code Generator performs the following steps to optimize the code:

1. Every C function that implements an operation is surrounded by a `#ifndef` definition.

**Example 367 The `#ifndef` definition**

```
#ifndef XNOUSE_AND_BIT_STRING
  /* function implementing the operation */
#endif
```

2. During the code generation, the usage of the operations in the transitions in state machines that implement active classes is recorded.
3. The dependencies between different operations are updated. For instance, an equal operation for a struct type may depend on equal operations for all its component types.
4. For each operation that is found to be unused, a `#define` definition is generated that removes the code for that operation. All the defines are placed in a file called `auto_cfg.h`

**Example 368 The #define command**

```
#define XNOUSE_AND_BIT_STRING
```

# Names in Generated C Code

This section is valid both for the C Code Generator and the AgileC Code Generator.

## Prefixes and suffixes in generated C names

When a UML name is translated to an identifier in C, a prefix or a suffix is normally added to the name given in UML. This prefix is used to prevent name conflicts in the generated code, as UML has other scope rules than C and also allows 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 UML to have a sort, an attribute and an operation with the same name defined in an active class. So the purpose of the prefix or suffix is to make each translated UML name to a unique name in the C program.

It is recommended to use suffix as it makes the code more readable. A reason to use prefix instead is if the used compiler only looks at a limited number of characters when parsing the names of identifiers. This makes it necessary to have the unique part in the beginning of the name.

### Names in generated code using suffix

A generated name for a UML object contains three parts in the following order:

1. The UML name stripped from characters not allowed in C identifiers
2. An underscore ‘_’
3. 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.

### Names in generated code using prefix

A generated name for a UML 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 UML name stripped from characters not allowed in C identifiers

## Sequence of characters

The sequence of characters that make the name unique is determined by the position of the declaration in the 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 "root".

**Example 369**

Given, say, a sort is defined as the 5th declaration in an inline active class with parts 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 370: Generated names in code**

Examples of generated names:

| Name | Position of the Declaration | Generated Name |
|---|---|---|
| S1 | 10th declaration in the system | S1_9 or z9_S1 |
| Var2 | 3rd declaration in the active class, which is the 5th declaration in an inline active class with parts, which is the 15th declaration in system | Var2_e42 or 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.

---

# 36

# C Code Generator Run-Time Model

The run-time model governs the applications generated by the C Code Generator, and controls the scheduling, signalling and execution of instances of active classes in the application.

This chapter provides information helping you to understand the principles that control the run-time execution of the application code. Such information is useful, should you need to proceed with debugging of your applications on C level.

The chapter also describes the C data structures used for representing the various objects that exist in the system.

# 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 connectord in Sequence
     Diagram 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 instance.

  The input port is a double 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 xSignalIdNode that represents this signal type. If the signal is in the avail list Pre is 0.
- **Prio** The priority of the signal.
- **Receiver** is used to reference the receiver of the signal. It is either set in the signal sending statement, or calculated (signal sending without direct addressing).
- **Sender** is the Pid value of the sending instance of an active class.
- **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 ensure compatibility of signal with earlier versions of the C Code Generator. This component is not used in normal cases and should not be present.
- **EnvChannel** is used to identify the outgoing connector in Sequence diagram trace.
- **GlobalInstanceId** is used in the Sequence diagram trace as a unique identification of the signal instance.

A signal without parameters is represented by a xSignalStruct, while for signals with parameters a struct type named ySignalPar_z_<package>_<number>_SignalName and a pointer type referencing this struct type (prefixed with yPDP_) are defined in the 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 371:**

```
typedef struct {
    SIGNAL_VARS
    SDL_Integer  Param1;
    SDL_Boolean  Param2;
} ySignalPar_z_<package>_<number>_sig;
typedef ySignalPar_z_<package>_<number>_sig
*yPDP_z_<package>_<number>_sig;
```

These types would represent a signal `sig(Integer, Boolean)`.

---

As all signals start 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` are to be accessed.

## Allocation of data areas for signals

In `sctsdl.c` there are two functions, `xGetSignal` and `xReleaseSignal`, where data areas for signals are handled:

```
xSignalNode xGetSignal(
  xSignalIdNode  SType,
  SDL_PId        Receiver,
  SDL_PId        Sender )

void xReleaseSignal( xSignalNode  *S )
```

`xGetSignal` takes a reference to the `xSignalIdNode` identifying the signal type and two Pid values (sending and receiving instances of active classes) 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 `xSignalIdNode` for the signal type provides the size information needed to correctly allocate the `ySignalPar_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`, `TimerSet`, `TimerReset`)
- In a number of places in the library:
  `SDL_Create`
  `SDL_SimpleReset`
  `SDL_Nextstate` (to handle guards of triggered transitions)

The function `xReleaseSignal` is used by:

- `SDL_Nextstate`
- `SDL_Stop`, in both cases to release the signal that initiated the transition.

## Detailed layout of signal parameters

Detailed layout information about signal parameters is made available on request, if `Set-SDL-Coder` is present in the C Code Generator Advanced options. The information is stored on two files generated by the C Code Generator: `<basename>_cod.h` and `<basename>_cod.c` These files should be present in the compile and link scheme.

The `<basename>_cod.c` file contains struct definitions of types `tSDLSignalInfoS` and `tSDLSignalParaInfoS` that describe the structure of the signals. These types are defined in the run-time library file `sctpred.h`

## Sending and receiving signals

In this subsection the signal handling operation is outlined. More details will be given in the section treating instances of active classes ("Send and receive of signals" on page 1142).

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 instance.

If the receiver is not already in the Ready queue (the queue containing the instance of active classes 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 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, to find the next signal in the input port that can cause a transition. Signal instances may then be saved or discarded.

### PAD function

There is no specific input function, instead this behavior is distributed both in the run-time 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 state machine is called. The **PAD function** is the C function where the

behavior of the state machine of an active class is implemented; this function is generated by the C Code Generator. The assignment of the signal parameters to local 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.

### Signal number file

Many real time operating systems require that signals/messages are represented with an integer value. Each signal is assigned an integer value. The signal number file is named `<basename>.hs`

See also "Improving performance of xOutEnv when many signals" on page 1050 for a guide how to use signal numbers.

## Timers and operations on timers

### Representation of timer

A timer with parameters is represented by a type definition, in exactly the same way as for a signal definition ("Data structure representing signals and timers" on page 1126). At run time, all timers that are set and that 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 ySignalPar_timername and yPDP_timername types in generated code exactly as a signal ("Sending and receiving signals" on page 1129), except that SIGNAL_VARS is replaced by TIMER_VARS.

### Timer queue

During its lifetime, a timer has one of 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. As the typedef for xSignalRec and xTimerRec are identical, type casting between xTimerNode and xSignalNode types is possible.

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 of active timers
- **TimerTime** is the time given in the Set operation.

The timer queue is represented by the component xTimerQueue in the variable xSysD:

```
xTimerNode xTimerQueue;
```

The variable is initialized in the function xInitKernel in sctsdl.c. When 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 double 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.

The xTimerRec structures are allocated and reused in the same way as signals.

### Handling of timers

Timers are handled in:

- `Timer` definitions
- `Timer` outputs
- `TimerSet` and `TimerReset` operations.

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. 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`).

Two functions are used to represent `TimerReset`. `SDL_SimpleReset` is used for timers without parameters and `SDL_Reset` for timers with parameters.

```
void SDL_Reset( xSignalNode  *TimerS )

void SDL_SimpleReset(
  xPrsNode       P,
  xSignalIdNode  TimerId )
```

`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 instance of an active class. It then releases the signal instance given as parameter. This signal is only used to carry the parameter values given in the `TimerReset` 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 TimerReset action the 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 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 signal sending using the function SDL_Output.

It can be checked if a timer is active by using a call to the function SDL_Active. This function is used in the generated code to represent the operation active.

```
SDL_Boolean SDL_Active (
  xSignalIdNode TimerId,
  xPrsNode      P )
```

**Note**

*Only timers without parameters can be tested. This is a restriction in the C Code Generator.*

There is one more place where timers are handled. When an instance of an active class instance performs a stop action, all timers in the timer queue connected to this instance are removed. This is performed by calling the function xRemoveTimer with the first parameter equal to 0.

# Active Classes

## Data structure representing active classes

An instance of an active class is represented by two structures, an xLocalPIdRec and a struct containing both the general data for the active class and the local variables and formal parameters of the instance (yVDef_ProcessName). The reason for having both the xLocalPIdRec and the yVDef_ProcessName will be discussed under "Create and stop operations" on page 1139.

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

#define PROCESS_VARS \
   xPrsNode       Pre; \
   xPrsNode       Suc; \
   int            RestartAddress; \
   xPrdNode       ActivePrd; \
   void (*RestartPAD) (xPrsNode  VarP); \
   XCALL_ADDR \
   xPrsNode       NextPrs; \
   SDL_PId        Self; \
   xPrsIdNode     InstNameNode; \
   xPrsIdNode     TypeNameNode; \
   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.

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 instance of an active class.
- `InstNr` of type `int`. This is the instance number of the current instance of an active class, which is used in the communication with the user in the Model Verifier and in dynamic error messages.
- `GlobalInstanceId` is used in Sequence diagram traces to have a unique identification of the instance of an active class.

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 instance of an active class in the ready queue ("Ready queue" on page 1137).

- `RestartAddress` of type `int`. This component is used to find the appropriate symbol to continue execute from.
- `ActivePrd` of type `xPrdNode`. This is a pointer to the `xPrdRec` that represents the currently executing operation called from this instance. The pointer is 0 if no operation is currently called.
- `RestartPAD`, which is a pointer to a function. This component refers to a PAD function that implements the state machine that realizes the dynamic behavior of an active class. `RestartPAD` is used to handle inheritance between active classes.
- `CallAddress` of type `int`. This component contains the symbol number of the operation call currently executed by this active class.
- `NextPrs` of type `xPrsNode`. This component is used to link the instance of an active class either in the active list or in the avail list for this active class. The start of these two lists are the components `ActivePrsList` and `AvailPrsList` in the `IdNode` representing the current active class.
- `Self` of type `SDL_PId`. This is the `Pid` value of the current instance of an active class.
- `InstNameNode` and `TypeNameNode` of type `xPrsIdNode`. These are pointers to the `PrsIdNode` representing the instantiation and active class type of the current running instance.
- `State` of type `int`. This component contains the int value used to represent the current state of the instance of an active class.
- `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 instance of an active class, or that caused the transition that is currently executed by the instance of an active class.
- `InputPort` of type `xInputPortRec`. This is the queue head in the double linked list that represents the input port of the instance of an active class. 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 instance of an active class. A “static” instance of an active class has parent equal to `NULL`. (Static means that the number of instances is fixed and all instances are created at start-up)
- `Offspring` of type `SDL_PId`. This is the `Pid` value of the latest created instance of an active class. An instance of an active class that has not created any instances has offspring equal to `NULL`.

- `BlockInstNumber` of type `int`. If the active class is part of an active class with its typed attribute defined as a composition, this component indicates which of the instances that it belongs to.
- `pREPLY_Waited_For` of type `xSignalIdNode`. When an active class is waiting in the implicit state for the `pREPLY` signal in a "remote" procedure call, this component is used to store the `IdNode` for the expected `pREPLY` signal.
- `pREPLY_Signal` of type `xSignalNode`. When an instance of an active class receives a `pCALL` signal, that is accepts an 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 concept Sender.
- `Trace_Default` of type `int`. This component contains the current value of the trace defined for the instance of an active class.
- `GRTrace` of type int. This component contains the current value of the graphical trace defined for the instance of an active class.
- `MSCETrace` of type `int`. This component contains the current Sequence diagram trace value for the instance of an active class.
- `InTransition` of type `xbool`. This component is true while the instance of an active class is executing a transition and it is false while the instance of an active class is waiting in a state. The Model Verifier user interface needs this information to be able to print out relevant information.

## Ready queue

The ready queue is a double linked list with a head. It contains the instances of active classes that can execute an immediate transition, but which has not been allowed to complete that transition. Instance of active classes are inserted into the ready queue during signal sending 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 is an object that does not represent any instance, but is inserted only to simplify the queue operations. It is referenced by the `xSysD` component:

```
xPrsNode   xReadyQueue;
```

This component is initiated in the function `xInitKernel` and used throughout the run-time 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 is to have all queues of interest (the ready queue, the timer queue, and the input ports) always sorted in the correct order. 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 an instance that can execute
            a transition ) {
    remove the signal causing the transition
      from input port;
    set up Sender in the instance to Sender of
      the signal;
    execute the PAD function for the state machine;
  }
}
```

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 Model Verifier user interface
- Calling the `xInEnv` function
- Handling real time or simulated time

- Delay execution up to the next scheduled event
- Handling guards or guards on triggered transitions that need to be recalculated.

## Create and stop operations

An instance of an active class is, while it is active, represented by the two structures:

- `xLocalPIdRec`
- The `yVDef_ProcessName` struct.

These two structures 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 240 on page 1139 shows how the `xLocalPIdRec` and the `yVDef_ProcessName` structures representing an instance of an active class are connected.

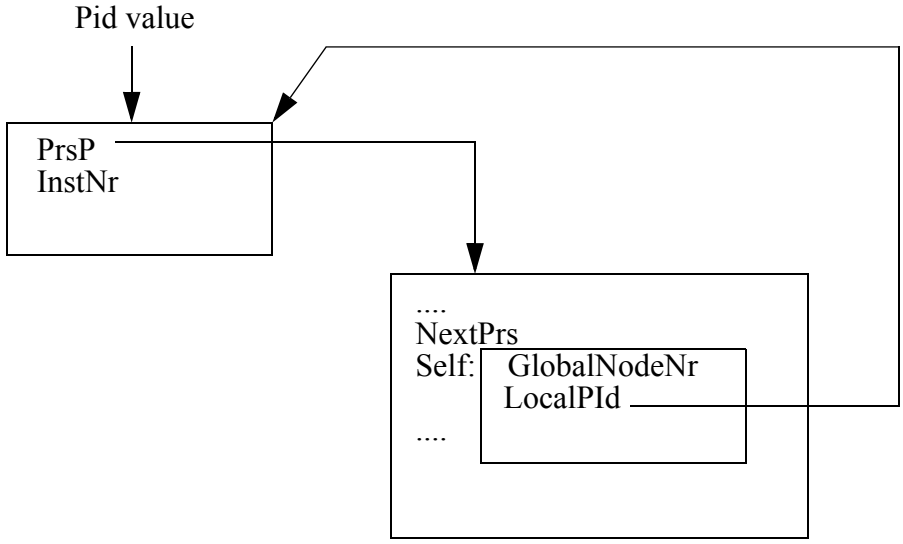



*Figure 240: A xLocalPIdRec and a yVDef_ProcessName representing an instance of an active class.*

When an instance of an active class performs a stop action, the memory used for the 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 attributes in other instances of active classes.

If a signal is sent to such an old `Pid` value, that is, to a stopped instance of an active class, it should be possible to find and perform appropriate actions. If the complete representation of an instance of an active class instance is re-used then this will not be possible. There must therefore remain some little piece of information and thus some memory for each instance of an active class that has ever existed. This is the purpose of the `xLocalPIdRec`. These structures will never be reused. Instead the following will happen when the instance of an active class performs a stop action.

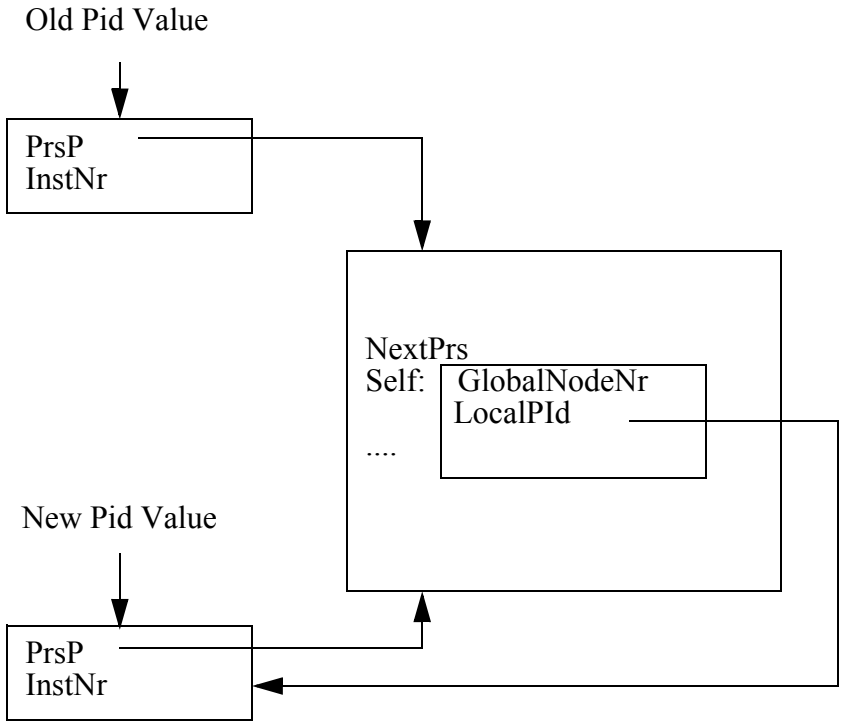



*Figure 241: The memory structure after the instance of an active class has performed a stop action*

- 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 active class.

To reuse the data area for an instance of an active class 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 instances of active classes 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 instance of an active class 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 an instance of an active class, the following steps are performed in the generated code:

1. Call `xGetSignal` to obtain the start-up signal.
2. Assign the parameters of the state machine to the start up signal parameters.
3. Call `SDL_Create` with the start-up signal as parameter, together with the `PrsIdNode` representing the active class to be created.

In `xGetProcess` the instance of an active class is removed from the avail list of the instance set (the component `AvailPrsList` in the `PrsIdNode` representing the set of all instances of an active class), or if the avail list is empty new memory is allocated.

The instance of an active class is linked into the list of active instances (the component `ActivePrsList` in the `PrsIdNode` representing the 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 active class. The start state can thus be seen as a state that receives the startup signal and saves all other signals. This implementation is completely transparent in the Model Verifier, where startup signals are never shown in any way.

**Note**

*The actual values for formal parameters 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 instance of an active class by calling `xRemoveTimer` with the first parameter equal to `0`. It then removes the instance of an active class executing the stop operation from the ready queue and from the active list of the active class and returns the memory to the avail list of the current instance set.

## Send and receive of signals

### General principles

Three actions are performed in the generated code to send a signal.

1. First `xGetSignal` is called to obtain a data area that represents the signal instance.
2. Then the signal parameters are assigned their values.
3. Finally the function `SDL_Output` is called to actually send the signal.

### Detailed operation of SDL_Output

In the `SDL_Output` function a number of dynamic tests are first done to check if the receiver in a direct addressing situation is not NULL and not stopped and check if there is a path to the receiver.

If the signal sending does not contain any direct addressing and the C Code Generator has not been able to calculate the receiver, the `xFindReceiver` function is called to calculate the receiver.

Next, in `SDL_Output`, signals to the environment are handled. Three cases can be identified here:

1. First, the environment function `xOutEnv` is called.
2. Then the signal is inserted into the input port of the instance of an active class representing the environment (`xEnv`).
3. Finally, internal signals in the system are treated. Here three cases can be identified (how this is evaluated is described last in this subsection):

- The signal can cause an immediate transition by the receiver.
- The signal should be saved.
- 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 instance of an active class 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 reuse the data area for the signal.

When a signal is identified to be the signal that should cause the next transition by the current instance of an active class (at a signal sending or Nextstate operation), the component `Signal` in the `yVDef_ProcessName` for the active class 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 immediately before the PAD function for the state machine 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 library where it is necessary to evaluate how to handle signals (receive, send, save, discard,...):

- At a signal sending operation to a currently idle instance of an active class.
- At a Nextstate operation, when the instance of an active class has 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 are:

- `SignalId`, which is a pointer to a signal.
- `VarP`, which is a pointer to an instance of an active class.

The result of the function is the following:

- The function should return the action that should be performed for this signal (receive, save,...), taking all information about this instance of an active class into account, like inheritance between active classes, 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 signal can be found.

After this last update the correct transition can be started by the scheduler. By calling the function referenced by `RestartPAD`, the first action performed will be to switch `RestartAddr` and to start executing the signal receipt symbol.

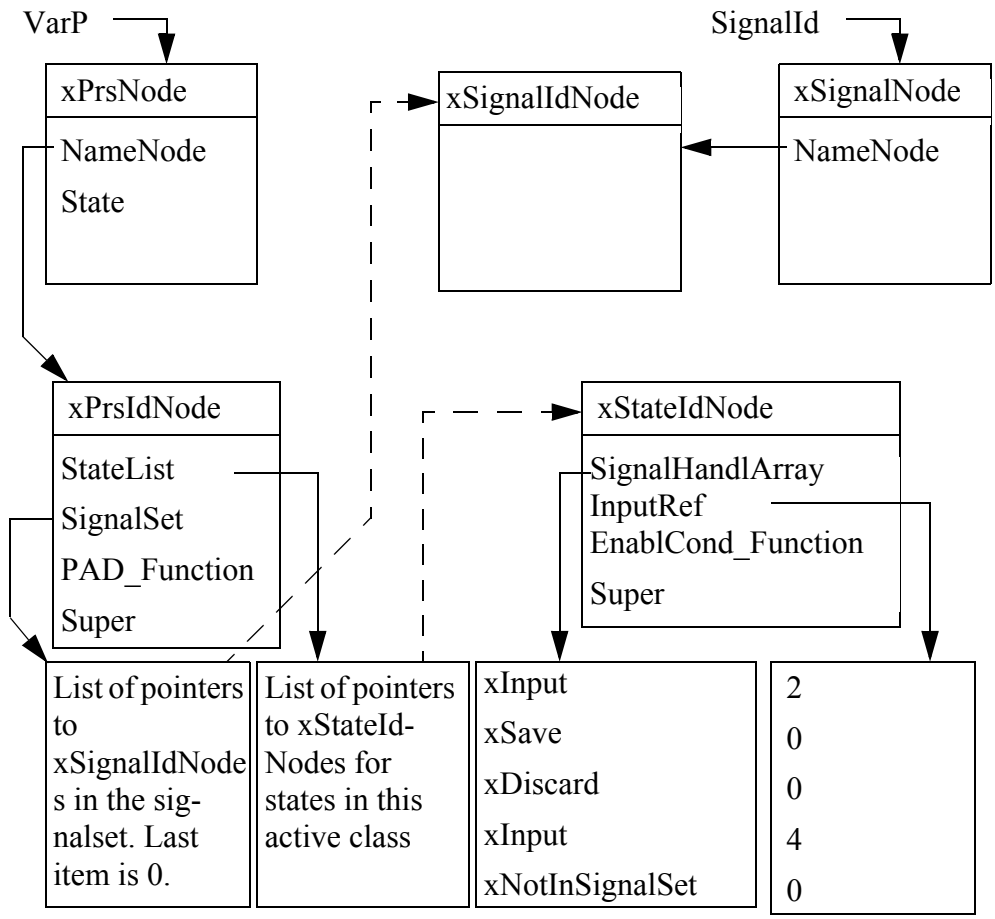



*Figure 242: Data structure used to evaluate the xFindInputAction*

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 active class, and the algorithm terminates returning the result `xDiscard`.
3. `StateId->SignalHandlArray[Index]` 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 the previous test 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 the previous test is `xSave`, the algorithm terminates returning this value.
   - If the result from the previous test is `xDiscard` and `ProcessId->Super` equal to `NULL`, then the algorithm terminates returning this value.
   - If the result from the previous test is `xDiscard` and `ProcessId->Super` not equal to `NULL`, then this is an active class that inherits from another active class. It is then necessary to perform these tests 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 active class is scanned for a signal that can cause a transition. During the scan, signals might be saved or discarded until a signal receipt is found.
3. If no signal that can cause a transition is found, a check is made if any continuous signal can cause a transition ("Guards and guards on triggered transitions" on page 1147). The instance of an active class is thereafter removed from the ready queue.
4. If any signal (or guard on a triggered transition) can cause a transition, the instance of an active class is re-inserted into the ready queue. If the new state contains any guard or guard on a triggered transition with an expression that might change its value during the time the instance of an active class is in the state, the instance is inserted into the check list

**See also**

"Guards and guards on triggered transitions" on page 1147 in Chapter 36, *C Code Generator Run-Time Model*

## Decision and action operations

Decision and Action operations are implemented in generated code, except for the Trace-functions implemented in the `sctutil.c` and `sctda.c` files and for informal decisions and any decisions that use any of the support functions in `sctda.c`. A decision is implemented as a C if-statement, while the assignments in an Action are implemented as assignments or function calls in C.

## Compound statements

A compound statement without attribute declarations is translated to the sequence of action it contains, while a compound statement with attribute declarations is translated in the same way as operations (without parameters).

The new statement types in compound statements are translated according to the following rules:

- Conditional is translated to `if` in C
- Decisions in compound statements are translated as ordinary decisions.
- Loops, continue, and break are all translated using `goto` in C.

## Guards and guards on triggered transitions

The expressions involved in guards or guards on triggered transitions are implemented in generated code in functions called `yCont_StateName` and `yEnab_StateName`. These functions are generated for each state containing guards and guards on triggered transitions. 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 guard 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 parameter, if it has a value >= 0 indicates that parameters three and four are used
  - The third is the IdNode for the active class or operation where the actual guard on triggered transition can be found
  - The fourth is the RestartAddress connected to this guard.

If an expression for a guard on triggered transition with value true is found, a signal instance representing the guard 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 xSignalIdNode (referenced by the variable xContSigId).

The check list contains the instances of active classes that wait in a state where guards and guards on triggered transitions need to be repeatedly recalculated.

An instance of an active class is inserted into the check list if:

- It enters a state containing guards or guards on triggered transitions.
- No signal or guard can cause an immediate transition.
- One or several of the expressions in the guards or guards on triggered transitions can change its value while the contained state machine 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 guards or guards on triggered transitions is modeled by letting the instance of an active class repeatedly send signals to itself, thereby repeatedly entering the current state. In the implementation chosen here, nextstate operations are performed "behind the scene" for all instances of active class 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.

## Global attributes

For a global attribute there are two components in the `yVDef_ProcessName` struct. One for the current value of the attribute and one for the currently "exported" value of the attribute. For each global attribute 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 an attribute when referring to it in an "import" action.

The "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 attribute (implicit or explicit)
- `P` is the Pid expression given in the import action
- `IsDef` is 0 or 1 depending on if any `Pid` expression is given in the import action or not
- `Importer` is the importing instance of an active class.

The `xGetExportAddr` will check the legality of the "import" action and will, if no `Pid` expression is given, calculate which active class it should be imported from.

If no errors are found, the function will return the address where the value of the attribute can be found. This address is then casted to the correct type (in generated code) and the value is obtained. If no active class possible to "import" from is found, the address of an attribute containing only zeros is returned by the `xGetExportAddr` function.

# Operations

## Data structure representing operations

An operation in an active class is represented by a struct type. The `xPrdRec` struct defined in `scttypes.h`, is, a struct containing general information about an operation, while the parameters and attributes of the operation are defined in generated code in the same way as for active classes. ("Data structure representing active classes" on page 1133).

In `scttypes.h` the following types concerning operations 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` structures 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 operation type.
- `StaticFather` of type `xPrdNode`. This is a pointer that represents the scope hierarchy of operations (and the active class at the top), which is used when an operation refers to non-local attributes. An example is shown in Figure 243 on page 1152. `StaticFather == 0` means that the static father is the active class.

- `DynamicFather` of type `xPrdNode`. This is a pointer that represents that this operation is called by the referenced operation. `DynamicFather == 0` means that this operation is called from the active class. This component is also used to link the `xPrdRec` in the avail list for the operation type.
- `RestartAddress` of type `int`. This component is used to find the appropriate symbol to continue execution from.
- `CallAddress` of type `int`. This component contains the symbol number of the operation call performed from this operation (if any).
- `RestartPRD` is a pointer to an operation function. This component refers to the PRD function where to execute the next sequence of symbols. `RestartPRD` is used to handle inheritance between operations.
- `pREPLY_Signal` of type `xSignalNode`. When an instance of an active class receives a pCALL signal, that is accepts an 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 operation.

Figure 243 on page 1152 presents an example of the structure of `yVDef_ProcedureName` after four nested operation calls. Operation Q is declared in the active class, operation R and S in Q and T in S.

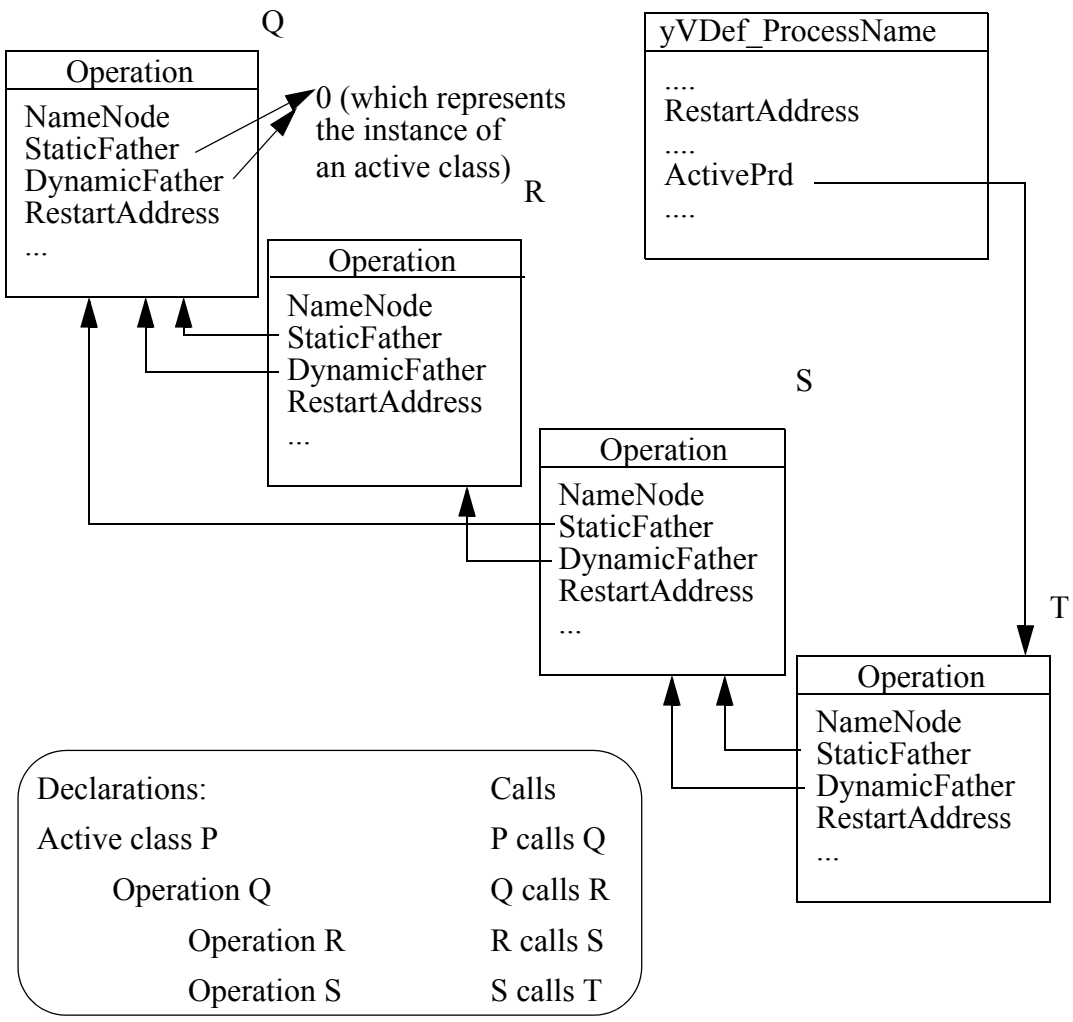



*Figure 243: Structure of* `yVDef_ProcedureName` *after four nested operation calls*

## PRD function

Operations are partly implemented using C functions and partly using the structure shown above. Each operation is represented by a **PRD function**, which is a C function that is called to execute actions defined in the operation. This function corresponds to the PAD function for state machines. The formal parameters and the attributes are however implemented using a struct defined in generated code. The operation stack for nested operation calls is implemented using the components `StaticFather` and `DynamicFather`, and does not use the C function stack.

## Calling and returning from operations

Operation calls and operation returns are handled by three functions.

- One function is handling the allocation of the data areas for operations:

```
xPrdNode  xGetPrd( xPrdIdNode  PrdId )
```

- Two functions are called from the generated code at an operation call and an operation return:

```
void xAddPrdCall(
  xPrdNode  R,
  xPrsNode  VarP,
  int       StaticFatherLevel,
  int       RestartAddress )

void xReleasePrd (xPrsNode  VarP)
```

An operation call is in C represented by the following steps:

1. Calling `xGetPrd` to obtain a data area for the operation.
2. Assigning operation parameters to the data area.
3. Calling `xAddPrdCall` to link the operation into the static and dynamic chains.
4. Calling the C function modeling the operation, that is the `yProcedureName` function.

The parameters to `xAddPrdCall` are as follows:

- `R`: This is a reference to the `xPrdNode` obtained from the call of `xGetPrd`.
- `VarP`: A reference to the `yVDef_ProcessName`, that is the data area for attributes and parameters of the active class (even if it is an operation that performed the operation call).
- `StaticFatherLevel`: This is the difference in declaration levels between the caller and the called operation. This information is used to set up the `StaticFather` component correctly.
- `RestartAddress`: This is the symbol number of the symbol directly after the operation 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 parameters and attributes of the operation. IN/OUT parameters are represented as addresses in this struct.

An operation return is in generated code represented by calling the `xReleasePrd` followed by return 0, whereby the function representing the behavior of the operation is left.

The function representing the behavior of the operation is returned in two main situations:

- When a `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 symbol after the operation call. If 1 is returned the execution of the instance of an active class should be terminated and the scheduler (main loop) should take control. This could mean that a number of nested operation calls should be terminated.

To continue to execute at the correct symbol when an operation should be resumed after a nextstate operation, the following code is introduced in the PAD function for state machines containing operation calls:

```
while ( yVarP->ActivePrd != (xPrdNode)0 )
  if ((*yVarP->ActivePrd->RestartPRD)(VarP))
    return;
```

This means that uncompleted operations are resumed one after one from the bottom of the operation stack, until all operations are completed or until one of them returns 1, that is executes a nextstate operation, at which the instance of an active class is left for the scheduler again.

# Connectors

## Finding the receiving instance of an active class

The `ChannelIdNodes` for connectors and ports are used in the functions `xFindReceiver` and `xIsPath` These two functions are called from `SDL_Output` to find the receiving instance of an active class when there is no direct addressing in the signal sending statement, respectively to check that there is a path to the receiver in the case of a direct addressing in the Signal Sending statement.

In both cases the paths built up using the `ToId` components in the `IdNodes` for active classes, connectors are followed. Figure 244 on page 1156 shows the structure of these paths.

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 as in "Example of a small system and the resulting symbol table" on page 1156. As you can see in this example the declarations in the system are directly reflected by `IdNodes`.

**Note**

*Each connector is represented by two* `IdNodes`*, one for each direction. This is also true for a unidirectional connector. In this case the signal set will be empty for the unused direction.*

# Example of a small system and the resulting symbol table

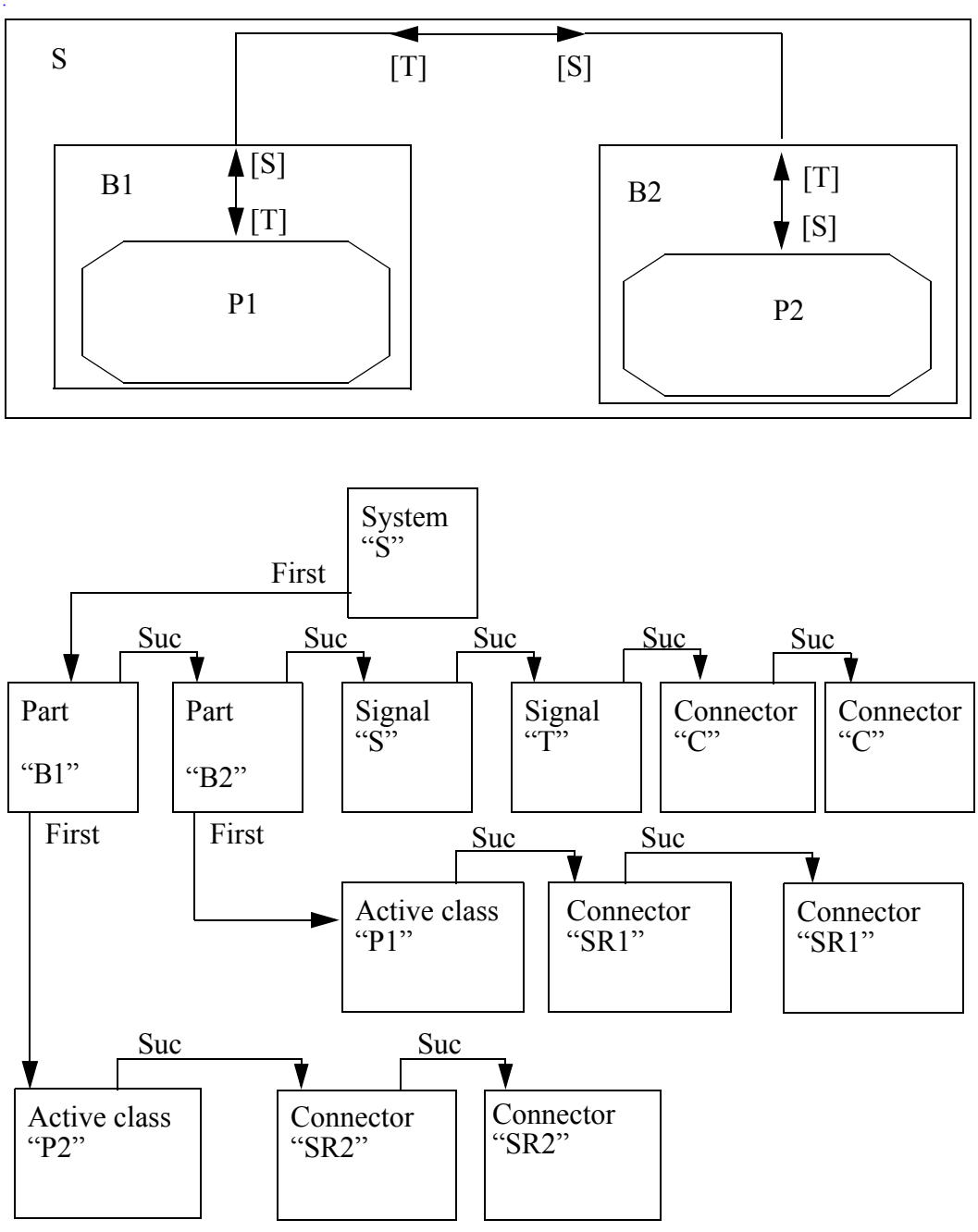



*Figure 244: The symbol table tree for the system.*

Each `IdNode` representing an active class, a connector 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. An active class 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 244 on page 1156 there is no branching, so all `ToId` arrays will be of the size necessary for two pointers.

Figure 245 on page 1157 shows how the `IdNodes` for the instances of active classes, connectors 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.

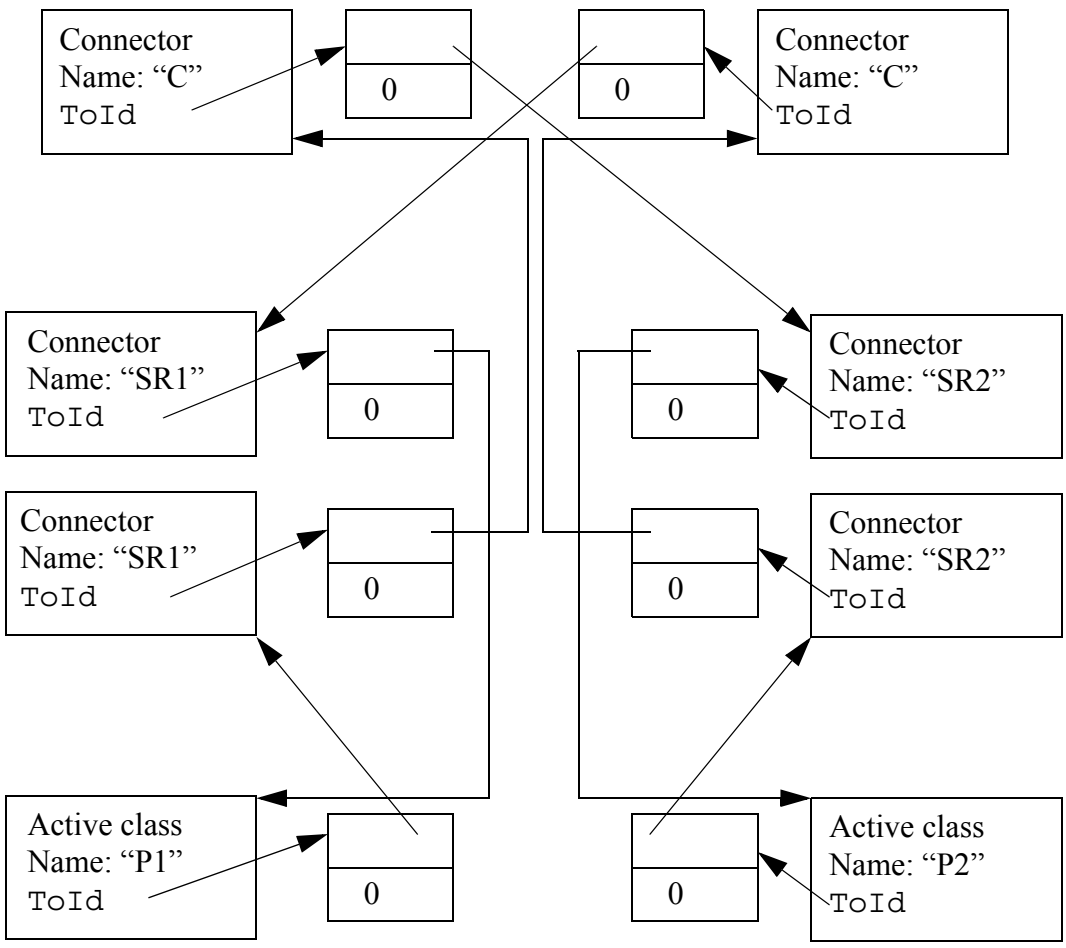



*Figure 245: The connection of* `ToId` *for the system.*

# 37

# C Code Generator Symbol Table

This chapter contains reference documentation about the symbol table created by the C Code Generator. The symbol table is used for storing information mainly about the static properties of the application, such as the structure in terms of parts, connectors and the valid input signal set for active classes. Some dynamic properties are also placed in the symbol table; for example the list of all active instances of active classes of an instance set.

The nodes in the symbol table are structures with components initialized in the declaration. During the initialization of the application, a tree is built up from these nodes.

# Symbol Table Creation and Structure

## Symbol table creation

The symbol table is a tree that is created in two steps:

1. Symbol table nodes are declared as structures with components initialized in the declaration.
2. The `yInit` function updates some components in the nodes and builds a tree from the nodes.

## Symbol table structure

The following names can be used to refer to the nodes that are always present in the table. 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 system. This is true also for packages containing the predefined data types. The nodes for the predefined data types, which are children to the node for the package Predefined, can be directly referenced by the names `xSrtN_SDL_<type>`, according to the list above.

## Symbol table nodes

Nodes in the symbol table are placed in the tree according to their place of declaration. A node that represents an item declared in a part is placed as a child node to that part node, and so on. The hierarchy in the symbol table tree will directly reflect the structure and declarations within the parts and active classes.

The following node types will be present in the tree:

| Node Type | Description |
|---|---|
| `xSystemEC` | Represents an instance of "root active class", that is the system or the system instance. |
| `xSystemTypeEC` | Represents a "root active class". |
| `xPackageEC` | Represents a package. |
| `xBlockEC` | Represents parts. |
| `xBlockTypeEC` | Represents active classes with parts. |
| `xBlockSubstEC` | Represents a decomposition of a part and can be found as a child of a part node. |
| `xProcessEC` | Represents active classes and instances of active classes. The "environment active class" node is placed after the system node and is used to represent the environment to the system. |
| `xProcessTypeEC` | Represents an active class. |
| `xProcedureEC` | Represents a procedure. |
| `xOperatorEC` | Represents an operation in a datatype or passive class. |
| `xCompoundStmtEC` | Represents a compound statement containing attribute declarations. |
| `xSignalEC`<br>`xTimerEC` | Represents a signal or timer type. |
| `xRPCSignalEC` | Represents the implicit signals (`pCALL`, `pREPLY`) used to implement calls of "remote" operations. |

| Node Type | Description |
|---|---|
| xSignalParEC | There will be one signal parameter node (a child to a signal and timer), for each signal or timer parameter. |
| xStartUpSignalEC | Represents a start-up signal, that is, the signal sent to a newly created instance of an active class containing the actual parameters of the state machine implementing the behavior. A xStartUpSignalEC node is always placed directly after the node for its active class. |
| xSortEC<br>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 the literal list. |
| xStateEC | Represents a state and can be found as a child to nodes for active classes and operations in active classes. |
| xVariableEC<br>xFormalParEC | Represents an attribute or a formal parameter to state machines and can be found as children to nodes for active classes and operations in active classes. |
| xChannelEC<br>xSignalRouteEC<br>xGate | Represents a connector or a port. |
| xRemoteVarEC | Represents a definition of attribute in an interface. |
| xRemotePrdEC | Represents a definition of an operation in an interface. |
| xSyntVariableEC | Represents implicit variables or components introduced by the C Code Generator. |
| xSynonymEC | Represent attributes in package. Not used. |

## Naming in symbol table

The nodes (the struct variables) will be given names in the generated code according to the following:

| Name in symbol table | Used for |
|---|---|
| `ySysR_SystemName` | System, system type, system instance |
| `yPacR_PackageName` | Package |
| `yBloR_BlockName` | Part, active class with part and instances of the active class |
| `yPrsR_ProcessName` | Inline active class, active class, instance of an active class |
| `yPrdR_ProcedureName` | Operation |
| `ySigR_SignalName` | Signal, timer, startup signal, RPC signal |
| `yChaR_ChannelName` | Connector, port |
| `yStaR_StateName` | State |
| `ySrtR_NewtypeName` | Newtype, syntype |
| `yLitR_LiteralName` | Literal |
| `yVarR_VariableName` | Attribute, formal parameter, signal parameter, struct component |
| `yReVR_RemoteVariable` | Attribute in interface |
| `yRePR_RemoteProcedure` | Operation in interface |

## Node references

In most cases it is of interest to refer to a symbol table node via a pointer. By taking the address of an attribute according to the table above, that is

```
& yPrsR_Process1
```

such a reference is obtained. To ensure future 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

## xIdNode type definitions in the symbol table

The following type definitions, defined in 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,
  xCompoundStmtEC,
  xSyntVariableEC
  xMonitorCommandEC
}   xEntityClassType;

typedef enum {
  xPredef, xUserdef, xEnum,
  xStruct, xArray, xGArray, xCArray,
  xOwn, xORef, xRef, xString,
  xPowerSet, xGPowerSet, xBag, xInherits, xSyntype,
  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 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;

                                      /*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;
```

## Components common to all table nodes

The type definitions 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 is needed to access any of the `xIdECNode` types. In the symbol table pointer types are defined for this purpose, accessing each of the `xECIdStruct` according to the following example:

```
typedef XCONST struct xIdStruct  *xIdNode;
```

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 "object" the node represents. `xEntityClassType` is an enum type containing elements for all entity classes.
- **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 "object" as a character string. This component is not needed in an application, only in a Model Verifier.

## Components specific to entity classes

Next in the table are present the components that depend on what entity class is to be represented. This section describes the non-common elements in the other `xECIdStruct`.

### Package components

- `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 (root active class) components

- **`Content`** of type `xIdNode *`. This component contains a list of all connectors at the system level (the "root" active class in the model).
- **`VirtPrdList`** of type `xPrdIdNode *`. This is a list of all virtual operations in active classes 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 graphical trace value defined for the system.
- **`MSCETrace`** of type `int`. This component contains the current Sequence diagram trace value defined for the system.

### Connector and port components

For connectors and ports there are always two consecutive `xChannelIdNodes` in the symbol table, representing the two possible directions for a connector or port. The components are:

- **`SignalSet`** of type `xIdNode *`. This component represents the signal set of the connector in the current direction (a unidirectional connector has an empty signal set in the opposite direction).

  `SignalSet` is an array with components referring to the `xSignalIdNodes` that represent the signals in 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 "object" that this connector/port is connected to (connected to in the sense: objects that signals are sent forward to).

  The objects that may be referenced in `ToId` are connectors, ports and active classes. The last component in the array is always a `NULL` pointer (the value `(xIdNode)0`).

- **Reverse** of type xChannelIdNode. This is a reference to the symbol table node that represents the other direction of the same connector or port.

**Parts and active classes with parts**

- **Super** of type xBlockIdNode. In a part, this component is NULL. In an active class with a part, this component is a reference to the part it inherits from (NULL if no inheritance). In an instance of the active class, this is a reference to the active class that is instantiated.
- **Contents** of type xIdNode *. In an instantiation of an active class with parts, these components contains lists of:
  - The instantiations of active classes in the part
  - The connectors in the part
  - The outgoing ports from the part
  - The inline active classes in the part
  - The ports defined in instantiations of active classes in the part.
- **VirtPrdList** of type xPrdIdNode *. This is a list of all virtual operations in this instance of an active class.
- **NumberOfInst** of type int. This is the number of instances in an instance set. The component is thus only relevant for an instance of an active class with parts.
- **Trace_Default** of type int. This component contains the current value of the execution trace defined for the part.
- **GRTrace** of type int. This component contains the current value of the graphical trace defined for the part.
- **MSCETrace** of type int. This component contains the current Sequence diagram trace value defined for the part.
- **GlobalInstanceId** of type int. This component is used to store a unique id needed when performing Sequence diagram trace.

**Active classes, inline active classes, instance components**

- **StateList** of type xStateIdNode *. This is a list of references to the xStateIdNodes for this (inline) active class. Using the state value of an executing instance of an active class, this list can be used to find the corresponding xStateIdNode.

- **SignalSet** of type `xIdNode *`. This represents the valid input signal set of the active class.

  `SignalSet` is an array with components that refer to `xSignalIdNodes` that represents 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 instance that this instance of an active class is connected to, that is an "object" that signals are sent forward to.

  The "objects" that may be referenced in `ToId` are connectors, ports, instance of active classes. The last component in the array is always a `NULL` pointer (the value `(xIdNode)0`).

- **MaxNoOfInst** of type `int`. This represents the maximum number of concurrent instances of an active class that may exist according to the specification for the current active class. An infinite number of concurrent instances of an active class 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 instance set.

- **NoOfStaticInst** of type `int`.This component contains the number of static instances that should be present at start up of the instance set of an active class.

- **ActivePrsList** of type `xPrsNode *`. This is the address of a pointer to the first position in the single-linked list of active instances of the current active class.

  The list is continued using the `NextPrs` component in the `xPrsRec` struct that is used to represent an instance of an active class. The order in the list is such that the first created of the active 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 active class (the struct: `yVDef_ProcessName`).

- **Prio** The priority of the signal.

- **AvailPrsList** of type `xPrsNode`. This is the address to the avail list pointer for instances of active classes that have stopped. The data area can later be reused in subsequent Create actions on this active class or instantiation of it.

- **Trace_Default** of type `int`. This component contains the current value of the trace defined for the active class.

- **GRTrace** of type `int`. This component contains the current value of the graphical trace defined for the active class.
- **GRrefFunc**, which is a pointer to a function that, given a symbol number (number assigned to a symbol in a state machine), will return a string containing the graphical reference to that symbol.
- **MaxSymbolNumber** of type `int`. This component is the number of symbols contained in the current active class.
- **SignalSetLength** of type `int`. This component is the number of signals contained in the signal set of the current active class.
- **MSCETrace** of type `int`. This component contains the current Sequence diagram trace value defined for the active class.
- **CoverageArray** of type `long int *`. This component is used as an array over all symbols in the active class. 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 active class 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 active class. 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 active class. This function is called when an instance of an active class of this type is to execute a transition. The `PAD_Functions` will be part of generated code, as they contain the actions defined in the state machine that implement the behavior.
- **Free_Vars**, which is a pointer to a function. This pointer refers to the `yFree_ProcessName` function for the current active class. This function is called when an instance of an active class performs a stop action to deallocate memory used by the local attributes in the active class.
- **Super** of type `xPrsIdNode`. In an inline active class this component is `NULL`. In an active class this component is a reference to the active class that it inherits from (`NULL` if no inheritance).
- **VirtPrdList** of type `xPrdIdNode *`. This is a list of all virtual operations in the instantiation of the active class.
- **InBlockInst** of type `xBlockIdNode`. This component is a reference to the part (if any) that this active class is part of.

- **RefToDefinition** of type `char *`. This is a graphical reference to this active class.

### Operations, compound statement components

Compound statements containing attribute declarations are treated as operations. However, such objects can, for example, not contain states.

- **StateList** of type `xStateIdNode *`. This is a list of references to the `xStateIdNodes` for this inline active class or active class. Using the state value of an executing instance of an active class, this list can be used to find the corresponding `xStateIdNode`.
- **SignalSet** of type `xIdNode *`. This represents the valid input signal set of the inline active class or active class.

  `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, that is 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 procedure is called and will execute the appropriate actions. The `yProcedureName` functions will be part of the 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 attributes in the procedure.
- **VarSize** of type `xptrint`. This is 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 graphical 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 operations in an operation in an active class.

**Remote operation components**

- **RemoteList** of type `xRemotePrdListNode`. This component is the start of a list of all active classes that "export" this operation. This list is a linked list of `xRemotePrdListStructs`, where each node contains a reference to the "exporting" active class.

**Signal, timer, startup signal, and RPC signals components**

- **VarSize** of type `xptrint`. This is the size, in bytes, of the data area used to represent the signal (the struct: `ySignalPar_SignalName`).
- **AvailSignalList** of type `xSignalNode *`. This is the address to the avail list pointer for signal instances of this signal type.
- **Equal_Timer** This 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** 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 *`. This is a reference to the definition of the signal or timer.
- **Prio** The priority of the signal.

**State components**

- **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 state machine 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 state machine 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 guard expression to know if the signal should result in a signal receipt or if it 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 signal receipt symbol in the graph.
- **EnablCond_Function** is a function that returns xInputAction. If the state contains any guards, 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 an instance of an active class and calculates the guard for receiving of the current signal in the current state of the given instance.

  The function returns either of the values xInput or xSave. The EnablCond_Functions will be part of the generated code, as they contain guard 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** is a function returning int. If the state contains any guards on triggered transition, 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 guard or guard on triggered transition expression in the state contains a reference to an object that might change its value even though the instance of an active class does not execute any actions.
- **CoverageArray** of type `long int`. This component is used as an array over the signal set (+1) of the active class. 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 signal set, is used to record the number of guards on triggered transition that are 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 *`. This holds a reference to the definition of the state (one of the symbols where this state is defined).

### Sort and syntype components

- **Free_Function** This function pointer is non-0 for types represented using dynamic memory (`Charstring`, `OctetString`, `Strings`, `Bag`, for example). The `Free_Functions` are used to return dynamic memory to the pool of dynamic memory.
- **Test_Function** is a function returning `xbool`. It is non-0 for all types containing range conditions. The function pointers are used by the Model Verifier to check the validity of a value when assigning it to an attribute.
- **SortSize** of type `xptrint`. This component represents the size, in bytes, of an attribute 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, xUnionC, and xChoice. Depending on the value of SortType you also have the following components:

  If SortType is xArray, xGArray or 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 an 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, that is 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 (that is the value representing the default value). In xArray and xCArray it is 0.

  If SortType is xString, xGPowerSet or 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.

  If SortType is xInherits

  – **CompOrFatherSort**, of type xSortIdNode. This is a pointer to the SortIdNode that represents the inherited sort.

If `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 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 Model Verifier when it handles arrays with this type as index type.

**Attribute, formal parameter, signal parameter, and struct components**

- **`SortNode`** of type `xSortIdNode`. This component is a pointer to the `SortIdNode` that represents the sort of this attribute or parameter.
- **`Offset`** of type `xptrint`. This component represents the offset, in bytes, within the struct that represents the attributes of the active class or operation, the signal parameter, or the struct. This is the relative place of this component within the struct.
- **`Offset2`** of type `xptrint`. For a formal parameter of a state machine, this component represents the offset, in bytes, of a formal parameter in the `StartUpSignal`. For a global attribute in an active class this component represents the offset, in bytes, of the "exported" value for this attribute.
- **`IsAddress`** of type `int`. This component is only used for formal parameters of operations and is then used to indicate if the parameter is an IN parameter, an IN/OUT parameter, or a result attribute.

**Attribute in interfaces**

- **`SortSize`** of type `xptrint`. This component is the size of the type of the global attributes.

- **`RemoteList`** of type `xRemoteVarListNode`. This component is the start of a list of all active classes that "export" this global attribute. This list is a linked list of `xRemoteVarListStructs`, where each node contains a reference to the "exporting" active class and the `Offset` where to find the "exported" value.

# Type Info Nodes

## General

Type info nodes are data structures that are mainly used at run-time by the Generic Functions providing generic implementations of operations in data types.

The C type `tSDLTypeInfo used for the type info nodes` contains essentially the same information as the type `xSortIdNode`, the latter being a special case of the generic `xIdNode` used for SDL sorts only.

The type info nodes are used in most places in the generated code where `xSortIdNode` could have been used instead.

**Note**

*The type* `xSortIdNode` *is still used by the generated C code in some situations, and is therefore still maintained for backward compatibility. The type is candidate to become discontinued, and instead the* `tSDLTypeInfo` *type should be used.*

**See also**

"xIdNode type definitions in the symbol table" on page 1164

## Type definitions of type info nodes

The type definitions that describe the type info nodes are available in the `sctpred.h` file.

Each type info node is a struct that consists of:

- General components in type info nodes that are available for all type info nodes.
- Type specific type info node components that are specific for each type.

## Type info node optimization

A type info node is a data structure that during run-time describes the properties of a data type. This information is needed for the implementation of the generic operators, like for example a function that can perform assignment for any data type.

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 `#ifdef` constructs 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 header (`.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:

```
auto_cfg.h
```

The last section in this file will contain the defines for the usage of type info nodes. Below is an example.

**Example 372: Defines for the 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 `auto_cfg.h` is automatically included and used by AgileC Code Generator. In C Code Generator the following code can be found in sct-types.h:

```
#ifdef USER_CONFIG
#include "uml_cfg.h"
#elif defined(AUTOMATIC_CONFIG)
#include "auto_cfg.h"
#endif
```

So by defining USER_CONFIG and including `auto_cfg.h` in `uml_cfg.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. The code generator will look in the directory where the intermediate (`.pr`) file is produced. 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 according to the following rules:

- Each type that should be registered as used should be mentioned on a line of its own, starting with the type name.
- When 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 373: 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 be case sensitive. 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**

## General components in type info nodes

**Note**

*The components that are described in this section are available for **all** type info nodes, independent of the type they represent, and is not repeated for the type specific components.*

```
/* --- General type information for 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
{
  /* 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,

  /* 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,

  /* user defined types */
  type_SDL_Syntype=170,
  type_SDL_Inherits=171,
  type_SDL_Enum=172,
  type_SDL_Struct=173,
  type_SDL_Union=174, /* Not used */
  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,

  /* 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. If the first bit is set, it is necessary to look for memory to be freed inside 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, a special compare function needs to be supplied.
  - The third bit indicates if `memcpy` can be used to perform assign of this type. If the bit is set, a special assignment function needs to be supplied.
  - 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.

## Type specific type info node 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.

### Type info node components for enumeration types

```
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**: This is the number of literals in the enum type.
- **LiteralList**: This is a pointer to an array of tSDLEnumLiteralInfo elements. This list implements a translation table between enum values and literal names as strings

### Syntype, type with inheritance, and Own, Oref instantiations

```
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`: This is a reference to the type info node of the father sort (syntype, inherits) or component sort (Own, ORef).

### Type info node components for PowerSet (implemented as unsigned in [ ])

```
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.

### Type info node components for 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 UML struct. This component is ~0 for bit fields in UML (offsets cannot be calculated for bit fields).

- **ExtraInfo** in tSDLFieldInfo: The interpretation of this component depends on the properties in the UML field.
  - If Offset is ~0, the field is a bit field and ExtraInfo is a pointer to a tSDLFieldBitFInfo struct containing two functions to set and get the value of the bit field.
  - If Offset is not ~0 and ExtraInfo != 0, the 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).

### Type info node components for choice

```
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.
- **NumOfComponents**: The number of fields in the choice.
- **OffsetToUnion**: The offset to where the union, within the representation of the choice, starts.
- **TagSortSize**: The size of the tag type.

- **`TagSort`**: A reference to the type info node of the tag sort.

### Type info node components for `array` and `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).

### Type info node components for general arrays

A general array is an array that is represented as a linked list in C.

```
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.

### Type info node components for general PowerSet, Bag, String and Objectidentifier

```
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

### Type info node components for 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.

### Type info node components for signal

A signal is treated in the same way as a struct.

```
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.

### Utility macros for type info nodes

The following utility macros can be used to configure the type info nodes, to adapt them to compilers or for instance adding or removing components. This should not be done for the normal application areas of the C Code Generator.

```
#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;
```

# 38

# *C Code Generator Macros*

This chapter contains a reference to all C preprocessor macros, which are used to decide the properties of the run-time library and the code generated by the C Code Generator.

# General

This chapter is divided in a number of sections, each treating one major aspect of the generated code. Within each section the macros are enumerated in alphabetic order.

In the run-time library source and include files, and also in the generated C code, `#define/#ifdef/#ifndef` patterns are used to include or exclude parts of the code at compilation.

The macros that are used can be divided into three groups.

- Macros defining the properties of the selected library,
- Macros defining the implementation details of the properties,
- Macros defining properties of the compiler. This group of macros is discussed in "Adaptation to Compilers" on page 1074 in Chapter 33, *C and AgileC Runtime Libraries*.

To fully understand the descriptions of some of the macros in this chapter, it is strongly recommended to know the basic data structures used, especially for the static structures, i.e. the `xIdNodes`. This information can be found in the section "C Code Generator Symbol Table" on page 1159.

The information about the data types used for the dynamic structure and behavior of the application, that is instances, signals, timers, and so on, are also of interest. This can be found in the chapter "C Code Generator Run-Time Model" on page 1125.

# C Code Generator Macros

The following section contains a listing of the preprocessor macros used by the C Code Generator.

## Library version macros

### SCTAPPLCLENV

This macro defines a library for applications.

### SCTAPPLENV

This macro defines a library for applications without clock.

### SCTDEB

This macro defines a stand-alone Model Verifier application for any environment.

### SCTDEBCL

This macro defines a stand-alone Model Verifier application in real time, for any environment.

### SCTDEBCLCOM

This macro defines a Model Verifier application in real time for host debugging.

### SCTDEBCLENV

This macro defines a stand-alone Model Verifier application, with real time properties and environment functions, for any environment.

### SCTDEBCLENVCOM

This macro defines a Model Verifier application in real time and with environment functions, for any environment.

### SCTDEBCOM

This macro defines a Model Verifier application for host debugging.

### SCTOPT1APPLCLENV

This macro defines an application with minimal memory requirements, in which real numbers cannot be used. No information about connectors is generated.

### SCTOPT2APPLCLENV

This macro defines an application with minimal memory requirements, in which real numbers cannot be used. Connector information is declared as `const`.

## Compiler definition section macros

### SCT_POSIX

This macro defines UNIX/POSIX like compilers/systems.

### SCT_WINDOWS

This macro defines compilers on Windows.

## Configuration macros

### COMMENT(P)

This macro should be defined as:

```
#define COMMENT(P)
```

The macro is used to insert comments in included C code.

### GETINTRAND

This macro defines a random generation function, usually `rand()` or `random()`

### GETINTRAND_MAX

This macro defines the maximum integer value generated by function mentioned in GETINTRAND, usually RAND_MAX or 2147483647 (32-bit integers).

### SCT_VERSION_4_4

This macro is defined in the generated code if the C Code Generator version 4.4 was used.

### XCAT(P1,P2)

This macro defines how to concatenate tokens P1 and P2. The options are:

```
#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
```

### X_LONG_INT

The sort Integer is translated to `int` in C. To translate the Integer sort to `long int` instead, just define the macro `X_LONG_INT`.

### XMULTIBYTE_SUPPORT

This macro should be defined if the compiler supports multi-byte characters.

### XNOSELECT

This macro 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 pressing 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

This macro is defined in the generated code if the Cbasic C Code Generator is used. This macro should never be defined when building an application.

### XSCT_CADVANCED

This macro is defined in the generated code if the C Code Generator is used. This macro should be defined.

### X_SCTTYPES_H

This macro is defined in `scttypes.h` and used to allow including the `scttypes.h` file multiple 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 macros

The property macros described in this section can be used to tailor libraries. If not stated otherwise for a certain property, all C code, variables, struct components, and so on, are either included or excluded using conditional compiling, depending on whether the property is used or not.

This means, for example, that all code for the command-line interpreter used by the Model Verifier will be removed in an application, which makes the application both smaller and faster.

The property macros are in principle independent, except for the relations given in the descriptions below, and it should be possible to use them in any combination unless such a combination introduces a conflict.

The number of combinations is, however, so huge that it is hardly possible to even compile all combinations and test that they function to satisfaction. If you happen to form a combination that does not work, please send a complete model to Tau Support. so that the case can be addressed promptly.

### XASSERT

This macro is used to detect and report user-defined assertions that are not valid.

### XCALENDARCLOCK

This specifies that the clock function in `sctos.c` (not simulated time) should be used. Time is whatever the clock function returns.

### XCLOCK

This specifies that the clock function in `sctos.c` (not simulated time) should be used. Time is zero at start up.

### XCOVERAGE

This macro specifies to compile the application with code that computes and stores information about the current coverage of code. It should be used together with `XMONITOR`.

### XCTRACE

This macro should be defined if you want to compile preserving the possibility to report the current C line number during simulations. Defining this macro 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 Model Verifier command Show-C-Line-Number.

### XEALL

This macro defines all the following: XASSERT, XECREATE, XECHOICE, XECSOP, XEDECISION, XEERROR, XEEXPORT, XEFIXOF, XEINDEX, XEINTDIV, XEOPTIONAL, XEOUTPUT, XEOWN, XERANGE, XEREALDIV, XEREF.

### XECHOICE

Defining this macro detects and reports attempts to access non-active components in `choice` variables.

### XECREATE

Defining this macro detects and reports if more static instances are created at start up, than the maximum number of concurrent instances.

### XECSOP

Defining this macro detects and reports errors in ADT operations.

### XEDECISION

Defining this macro detects and reports when there is no possible path out from a `DecisionAction`.

### XEERROR

Defining this macro detects and reports the usage of the error term in an expression.

### XEEXPORT

Defining this macro detects and reports errors in referring global data.

### XEFIXOF

This macro will report overflow when a Real value is converted to an Integer value using the operation Fix.

### XEINDEX

Defining this macro detects and reports index out of bound in arrays.

### XEINTDIV

Defining this macro detects and reports integer division with zero.

### XENV

If this compilation macro is defined the environment functions `xInitEnv`, `xCloseEnv`, `xInEnv`, and `xOutEnv` will be called at appropriate places.

### XEOPTIONAL

Defining this macro detects and reports attempts to access non-present optional attributes in passive classes.

### XEOUTPUT

Defining this macro detects and reports warnings in signal sending (mainly signal sending where a signal is immediately discarded).

### XEOWN

Defining this macro detects and reports illegal usage of Own and ORef pointers.

### XERANGE

This macro will report range errors when a value is assigned to an attribute of a sort containing range conditions.

### XEREALDIV

Defining this macro detects and reports real division by zero.

### XEREF

Defining this macro detects and reports attempts to de-reference null pointer.

### XGRTRACE

This macro enables graphical trace back to source UML diagrams. This feature is used to implement graphical trace of simulations and debugging sessions and Model Verifier commands like Show-Next-Symbol and Show-Previous-Symbol. It is possible to use graphical trace without the command-line interpreter in the same way as the ordinary trace (substitute `Trace_Default` with `GRTrace`). The graphical trace is however synchronized which means that the speed of the application is dramatically reduced.

### XMAIN_NAME

If this macro is defined, then the `main ( )` function in `sctsdl.c` will be renamed to the name given by the macro. Sometimes when integrating generated applications 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 can be found in the file `sctsdl.c`.

### XMONITOR

Defining this macro will compile and link the Model Verifier command line interpreter into the application. This macro will implicitly set up a number of other macros as well.

### XMSCE

Defining this macro, the code will compile with the graphical Sequence diagram trace enabled.

### XNOMAIN

If this macro is defined the main function in `sctsdl.c` will be removed. The functions `main` and `xMainLoop` are removed using conditional compiling. This feature is intended to be used when code generated from a system should be part of an already existing application, that is when the system implements a new function in an existing environment. The following functions are available for you to implement scheduling of 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 system before any other function in the run-time library is called.

**SDL_Execute**: This function will execute one transition by the instance of an active class first in the ready queue. Before calling this function it must be checked that there really is at least one instance of an active class in the ready queue.

**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.

**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 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 run-time library works is given below:

**Example 374**

```
  while (1) {
#ifdef XMONITOR
    xCheckMonitors();
#endif
    if ( SDL_Timer_Prio() >= 0 )
      SDL_OutputTimer();
    else if ( SDL_Transition_Prio() >= 0 )
      SDL_Execute();
  }
```

## XSIGLOG

This facility makes it possible for you to implement your own log of the major events in the system. This macro is normally not defined. By defining this macro, each sending of a signal, that is 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 signal sending according to the following table:

| Numbers Of Receivers | Output Statement Contents |
|---|---|
| -1: | A direct addressing clause, but no path of channels and signal routes were found between the sender and the receiver. |
| 0: | No direct addressing clause, and no possible receivers were found in the search for receivers. |
| 1: | If the statement contains a direct addressing, a path of channels and signal routes is found between the sender and the receiver.<br>If the statement contains no direct addressing, exactly one possible receiver is found in the search for receivers.<br>The signal sending is successful. The only error situation that still might be present is if a signal sending with a direct addressing is directed to an instance of an active class that is stopped. |

The third parameter, `Path`, is an array of pointer to `IdNodes`, where `Path[0]` refers to the `IdNode` for the sending instance of an active class, `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 instance.

The parameter `P` in the `xProcessLog` function will refer to the instance of an active class 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. 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 "active class".

In the implementation of the `xSignalLog` and `xProcessLog` functions which should be user-provided, you have 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 you to implement a simple log facility in environments where standard input/output 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`

### XTENV

This is the same as `XENV` (it actually defines `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 system becomes idle (waiting for signals).

### XTRACE

If this macro is defined, traces of the execution can be printed on standard output. This facility is normally used together with the Model Verifier, but could also be used without it. The file `stdout` must be available for printing.

When the Model Verifier is not present, setting trace values must be performed in included C code, as the user interface is not available. The trace components are called `Trace_Default` and can be found in `IdNodes` representing the various active classes defined in the system, and in the struct `xPrsRec` used to represent an instance of an active class. The values stored in these components are the values given by the Set-Trace command in the Model Verifier. When the value is unspecified it is represented by -1.

When the Model Verifier is not present, 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 375**

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`.

## Code optimization macros

### XAVL_TIMER_QUEUE

Defining this macro will change the data structure used for the timer queue from a sorted linked list to a AVL tree. Models using a large number of timers will get a performance gain since insertions and deletions scale logarithmically with the number of timers in the avl tree instead of linearly as with the linked list.

### XCONST

Defining this macro will allow the majority of the `xIdNode` structures to be made constant by defining `XCONST` as `const`. This is only possible in applications, not in Model Verifiers.

### XCONST_COMP

This macro should normally be defined as const if XCONST is const. It is used to introduce const in the component declarations within the `xIdNode` structures.

Using the macros XCONST and XCONST_COMP most of the memory used for the `IdStructs` can be moved from RAM to ROM. This of course depends on the compiler and its properties.

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 left to the C 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, a lot of RAM memory will be saved, as the major part of the data area for `IdStructs` can be made `const`.

### XNOCONTSIGFUNC

This macro is to be defined in order to exclude the functions that calculate the expressions in guards on triggered transitions. This also saves one function pointer in the `xIdNode` for the states. If this macro is defined, then guards on triggered transitions cannot be used.

### XNOENABCONDFUNC

This macro is defined in order to exclude the functions that calculate the expressions in guards. This also saves one function pointer in the `xIdNode` for the states. If this macro is defined, then guards cannot be used.

### XNOEQTIMERFUNC

This macro is defined in order to exclude the functions that compare the parameters of two timers. This also saves one function pointer in the `xIdNode` for the signals. If this macro is defined, then timers with parameters cannot be used.

### XNOREMOTEVARIDNODE

This macro is defined in order not to include `xIdNodes` for definitions of attributes in interfaces.

### XNOSIGNALIDNODE

This macro is defined in order not to include `xSignalIdNodes` for signals and timers.

### XNOSTARTUPIDNODE

This macro is defined in order not to include `xSignalIdNodes` for startup signals.

### XNOUSEOFEXPORT

By defining this macro you state that you are not going to use global attributes.

**Note**

*An attempt to refer to global data when XNOUSEOFEXPORT is defined will result in a compilation error, as the function* `xGetExportAddr` *is not defined.*

### XNOUSEOFOBJECTIDENTIFIER

By defining this macro, the type `ObjectIdentifier` and all operations on that type are removed.

### XNOUSEOFOCTETBITSTRING

By defining this macro, the types `Bitstring`, `Octet`, `OctetString` and all operations on these types are removed.

### XNOUSEOFREAL

By defining this macro, the type real and all operations on real are removed. Defining this macro will remove all occurrences of C `float` and `double` types, and means for example that the type Real is no longer available.

This macro is intended to be used in situations when it is important to save space, to ensure that the library functions for floating type operations are not loaded. It cannot handle situations when you include floating type operations in C code. Another consideration is if `BasicCTypes.pr`, or other ADTs, are included in the system. If so, it is required that types dependent on Real are removed from these packages.

### XOPT

This macro will turn on full optimization (except `XOPTCHAN`), that is it will define the following macros:

| | |
|---|---|
| XOPTSIGPARA | XOPTDCL |
| XOPTFPAR | XOPTSTRUCT |
| XOPTLIT | XOPTSORT |

The `XOPT` macros should not be used together with the monitor.

### XOPTCHAN

This macro can be used to remove all information about the paths of connectors in the system. The following memory optimization will take place:

- The two `ChannelIdNodes` for each connector and port are removed.
- The `ToId` component in the `xPrsIdNodes` representing instances of active classes is removed.
- A number of functions in the library (`sctsdl.c`) are no longer needed and are removed.

When the information about connectors and ports is not present the following types of calculations can no longer be performed.

1. To check if there is a path of connectors between the sender and the receiver in a signal sending statement with direct addressing. This is no problem as this is just an error test that probably will not be performed in an application.
2. To calculate the receiver in a signal sending without direct addressing, if the C Code Generator has not performed this calculation at code generation time. This is more serious, as it means that signal sending without direct addressing will be used.

In an ordinary system, signal sending without direct addressing must be used to start up the communication between different parts of the system, as there is no other way to distribute the Pid values needed for signal sending with direct addressing.

This situation is solved if the C Code Generator can calculate the receiver. Otherwise the data type `PIdList` in the library of abstract data types is intended to solve the situation. When this data type is used, global Pid literals may be introduced, implemented as constant attributes. These literals can then be used to utilize signal sending statements with direct addressing clauses from the very beginning.

**Note**

*If this compilation macro is defined all signal sending must either be done with direct addressing an instance of an active class, or the receiver of the signal must be possible to calculate during code generation. If the* `XOPTCHAN` *macro is defined and signal sending without direct addressing clause is still used (which the C Code Generator cannot optimize), there will be a C compilation error saying that the name* `xNotDefPId` *is not defined.*

### XOPTDCL

This macro is to be defined in order not to include `xIdNodes` for attributes. There will be a `VarIdNode` in the symbol table tree for each attribute declared in active classes and in operations. These nodes are not used in an application and may be removed by defining the macro `XOPTDCL`.

### XOPTFPAR

This macro is to be defined in order not to include `xIdNodes` for parameters of state machines. There will be a `VarIdNode` in the symbol table tree for each formal parameter in active classes and in operations. These nodes are not used in an application and may be removed by defining the macro `XOPTFPAR`.

### XOPTLIT

This macro is to be defined in order not to include `xIdNodes` for literals. For each literal in a passive class that will be translated to an `enum` type, there will be a `LitIdNode` representing the literal. These nodes will not be used in an application and may be removed by defining the macro `XOPTLIT`.

### XOPTSIGPARA

This macro is to be defined in order not to include `xIdNodes` for signal parameters. In the symbol table tree, there will be one node for each parameter to a signal. These nodes are not necessary in an application and may be removed by defining the macro `XOPTSIGPARA`.

### XOPTSORT

This macro is to be defined in order not to include `xIdNodes` for passive classes and any syntype. Each passive class and syntype will be represented by a `SortIdNode`. These nodes are not used in an application if all the other `XOPT (XOPTSIGPARA, XOPTDCL, XOPTFPAR, XOPTSTRUCT, XOPTLIT)` described above are defined.

### XOPTSTRUCT

This macro is to be defined in order not to include `xIdNodes` for struct components. For each component in an SDL struct there will be one `VarIdNode` defining the properties of this component. A `VarIdNode` is not used in an application and can be removed by defining the macro `XOPTSTRUCT`.

### XPATH_INFO_IN_ENV_FUNC

This compilation switch should be set if the signal path for a signal sent from the application to the environment should become available to the user.

### XPRSCOUNT, XPRSCOUNTHASH

Defining one of these macros optimizes the counting of the number of instances of an active class. This is normally done before a new instance can be created, so this optimization gives a performance gain when dynamically creating a large number of instances. `XPRSCOUNT` keeps a linked list with precalculated counts of instances, while `XPRSCOUNTHASH` further optimizes `XPRSCOUNT` by keeping the counts in a hashtable for even faster lookups.

### XPRSHASH

Defining this macro will change the data structure tracking instances of an active class from a linked list to a hash table. The optimization is suitable for systems that use a very large number of active class instances.

**Note**

*When having active class instances in a part with non-single multiplicity, this optimization will change the ordering of these instances in the part container. When appending a new instance to it, or deleting an instance from it, the ordering of the container will probably change. Do not rely on the container index to give you a specific instance. If you need this functionality, use a parallell container with class references.*

### XPRSOPT

This macro, if defined, will optimize memory use for instances of active classes. All memory can be reused, but signal sending to a stopped instance, which memory has been reused by a new instance of an active class, cannot be detected. The new instance will in this case receive the signal.

The section "Create and stop operations" on page 1139 in Chapter 36, *C Code Generator Run-Time Model* describes how `xLocalPIdRec` structures are allocated for each created instance of an active class, and how these structures are used to represent 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 an instance of an active class that has performed a stop operation.

In an application that is going to run for a long period of time and that uses dynamically created instances, this way of handling `xLocalPIdRecs` will eventually lead to no memory being available.

By defining the macro `XPRSOPT`, the memory for the `xLocalPIdRecs` will be reused together the `yVDef_ProcessName` structures. This has two consequences:

- The need for memory will not increase due to the use of dynamically created and terminated instances (the memory need depends on the maximum number of concurrent instances of the active class).
- It will no longer be possible to always find the situation when a signal is sent to an instance that has performed a stop action.

More precisely, consider a situation where a Pid attribute, that refers to an instance of an active class, performs a stop operation. After that a create operation is performed, on the same active class, where the same data area is reused. Then the Pid attribute will now refer to the new instance.

This means, for example, that signals intended for the old instance will be sent to the new instance. It is still possible to detect signal sending to instances in the avail list even if `XPRSOPT` is defined.

### XUSE_SIGNAL_NUMBERS

This compilation switch should be set when compiling an application where the environment function that implements signals to the environment is designed to look up signals by their numbers (assigned by the C Code Generator) rather by their name. See "Improving performance of xOutEnv when many signals" on page 1050 for how to use this feature.

## Macros for definition of minor features

### XBREAKBEFORE

This macro should be defined mainly if the MONITOR or GRTRACE macros are defined. It will make the functions and struct components for 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
- XAT_LAST_SYMBOL

to suitable function calls. These functions are used to interrupt a transition between symbols during debugging.

### XCASEAFTERPRDLABELS

The symbols just after an operation call have to be treated specially, as the symbol number (case label in C) for these symbols is used as the restart address for the calling flow graph. Normally this macro should be defined. If calls of operations are transformed to proper C function calls, and Return is translated to a C return, and Nextstate in an operation is NOT translated to a C return (that is the active class will be "hanging" in the C function representing the operation) then it is not necessary to define XCASEAFTERPRDLABELS. This macro is related to the function of XCASELABELS.

### XCASELABELS

The function implementing the behavior of a state machine or operation contains one large C switch statement with a case label for each symbol. This switch is used to be able to restart the execution of a state machine or operation at any symbol. In an application, most of these labels can be removed (all except for those symbols that start a transition, that is start, triggered transition, guard on triggered transition). The macro XCASELABELS should be defined to introduce the case labels for all symbol. This means that XCASELABELS should be defined in a simulation, but not in an application.

### XCOUNTRESETS

This macro is used to count the number of timers that are removed at a reset operation. This information is used by the textual trace (XTRACE) to present this information, which is of interest at a stop action when more then one timer might be (implicitly) reset. XCOUNTRESETS should not be defined in an application.

### XENVSIGNALLIMIT

If this macro is defined, only a limited number of signals will be stored in the input port of the environment function. This macro is defined to determine the number of signals sent to the environment that, during simulation, should be saved in the input port of the "environment active class". Such signals can be inspected with the Model Verifier commands for listing of signals. This macro is only of interest in a simulation. The limit is equal to the value defined for `XENVSIGNALLIMIT` and is normally set to 20.

### XERRORSTATE

This macro is used to 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

This macro is used to 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.

### XFREEVARS

This macro is used to insert free function calls for all attributes of a type with free function, just before the stop or return actions. This means that free actions are performed on the allocated data that is referred to from attributes, before the object ceases to exist. This macro should be defined.

### XIDNAMES

This macro is used to determine if the name of an object should be stored in the `xIdNode` for the object. This character string is used for communication with the user, in for example the Model Verifier. Normally this macro should not be used in an application.

It might also be useful for target debugging to define XIDNAMES, as it is then fairly easy to identify objects by just printing the name from the Model Verifier. This feature requires a few percent additional memory.

### XNRINST

This macro should be defined if active class instance numbers (the number associated to an individual instance of an active class instance set) are to be maintained. XNRINST is normally only used in simulation applications such as the Model Verifier.

### XOPERRORF

This macro is defined to include the function `xSDLOpError` in `sctsdl.c`. This function is used to print run-time errors in ADT operations.

### XPRSSENDER

This macro is used to 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 Model Verifier after the transition is completed and the signal has been returned to the pool of available memory.

### XREADANDWRITEF

This macro is used to include the functions for basic Read and Write. This is needed mainly in simulations.

### XREMOVETIMERSIG

This macro is used to allow the removal of timers for not-executing Pid instances. This is needed only in simulations to implement the Model Verifier commands to set and reset timers.

### XSIGPATH

If this macro is defined then the functions `xIsPath` and `xFindReceiver` will return the path of connectors and ports from the sender to the receiver, as out parameters. This information can then be used in the Model Verifier, 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`. 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 Model Verifier, but not in an application, that is XSYMBTLINK should be defined in a Model Verifier but not in an application.

### XTESTF

This macro is used to include or remove test functions for syntype (or passive classes) with range conditions. The `yTest` function is used by the Model Verifier by to test index out of bounds in arrays and to test ranges. This means that XTESTF should be defined if the monitor is used or if XERANGE or XEINDEX is defined.

### XTRACHANNELSTOENV

This macro is to define the redirection connectors to the environment.

When using partitioning of an application, the number of connectors going to the environment is not known at code generation time. This means that the size of the data area used for the connections is not known. This situation is solved in two ways.

Either the function handling redirections allocates more memory, which is the default, or you specify how many connectors that will be redirected (which could be difficult to compute, but will lead to less need of memory).

**Example 376**

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 you want to specify the number of connectors, then the macro should be defined in the following way

```
#define XTRACHANNELSTOENV  10
#define XTRACHANNELLIST    ,0,0,0,0,0,0,0,0,0,0
```

that is `XTRACHANNELSTOENV` should be the number of connectors (that is 10 as in the example above), while `XTRACHANNELLIST` should be a list of that many zeros (a sequence of 10 zeros in this case).

### XTRACHANNELLIST

This macro is related to the function of XTRACHANNELSTOENV.

## Macros for static data, mainly xIdNode

### XBLO_EXTRAS

All generated struct values for (possibly inline) active classes that contain parts contain this macro last in the struct. By defining this macro new components can be inserted. The type `xBlockIdStruct` must be updated as well. Normally this macro should be empty.

**Example 377**

```
#define XBLO_EXTRAS  ,0
```

## XBLS_EXTRAS

All generated struct values for active classes that contain parts structures contain this macro last in the struct. By defining this macro new components can be inserted. The type `xBlockSubstIdStruct` must be updated as well. Normally this macro should be empty.

**Example 378**

```
#define XBLS_EXTRAS  ,0
```

## XCOMMON_EXTRAS

All generated struct values for `xIdNode` structures 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 379**

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 structures contain this macro last in the struct. By defining this macro new components can be inserted. The type `xLiteralIdStruct` must be updated as well. Normally this macro should be empty.

**Example 380**

```
#define XLIT_EXTRAS  ,0
```

## XPAC_EXTRAS

All generated struct values for package structures contain this macro last in the struct. By defining this macro new components can be inserted. The type `xPackageIdStruct` must be updated as well. Normally this macro should be empty.

**Example 381**

```
#define XSYS_EXTRAS  ,0
```

## XPRD_EXTRAS

All generated struct values for operation structures contain this macro last in the struct. By defining this macro new components can be inserted. The type `xPrdIdStruct` must be updated as well. Normally this macro should be empty.

**Example 382**

```
#define XSYS_EXTRAS  ,0
```

## XPRS_EXTRAS

`(PREFIX_PROC_NAME)`

All generated struct values for active classes, instances of active classes and inline active classes structures contain this macro last in the struct. By defining this macro new components can be inserted. The type `xPrsIdStruct` must be updated as well.

**Example 383**

```
#define XPRS_EXTRAS(PREFIX_PROC_NAME) \
   ,XCAT(PREFIX_PROC_NAME,_STACKSIZE)
```

## XSIG_EXTRAS

All generated struct values for signal, timer and startup signal structures contain this macro last in the struct. By defining this macro, new components can be inserted. The type `xSignalIdStruct` must be updated as well.

Normally this macro should be empty.

**Example 384**

```
#define XSIG_EXTRAS  ,0
```

### XSPA_EXTRAS

All generated struct values for signal parameter structures contain this macro last in the struct. By defining this macro, new components can be inserted. The type `xVarIdStruct` must be updated as well.

**Note**

*Attributes, parameters of state machines, signal parameters, and struct components are all handled in* `xVarIdStruct`.

Normally this macro should be empty.

**Example 385**

```
#define XSPA_EXTRAS  ,0
```

### XSRT_EXTRAS

All generated struct values for passive classes and syntype structures contain this macro last in the struct. By defining this macro new components can be inserted. The type `xSortIdStruct` must be updated as well. Normally this macro should be empty.

**Example 386**

```
#define XSRT_EXTRAS  ,0
```

### XSTA_EXTRAS

All generated struct values for state structures contain this macro last in the struct. By defining this macro new components can be inserted. The type `xStateIdStruct` must be updated as well. Usually this macro should be empty.

**Example 387**

```
#define XSTA_EXTRAS  ,0
```

### XSYS_EXTRAS

All generated struct values for the system (the “root” active class and instances of it), contain this macro. By defining this macro new components can be inserted, last in the structures. The type `xSystemIdStruct` must be updated as well. Normally this macro should be empty.

**Example 388**

```
#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 “root” active class or main thread.

### 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 main thread (if separate generation is used).

### XVAR_EXTRAS

All generated struct values for attributes, parameters to state machines, and struct components contain this macro last in the struct. By defining this macro new components can be inserted. The type `xVarIdStruct` must be updated as well (signal parameters also use the type `xVarIdStruct`). Normally this macro should be empty.

**Example 389**

```
#define XVAR_EXTRAS  ,0
```

## Data in state machines and operations in active classes

### PROCEDURE_VARS

This macro defines the struct components that are needed for each operation in active classes, such as state.

### PROCESS_VARS

This macro defines the struct components that are needed for each instance of a state machine. Examples:

```
state, parent, offspring, self, sender, inputport
```

### YGLOBALPRD_YVARP

This macro is used to declare the `yVarP` pointer, which points to the `yVDef` struct for the active class, in an operation defined outside of the active class. As a global operation never can access data that is local to an active class, 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 state machine. Example: temporary variables needed for signal sending, 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 state machine. It should be expanded to a declaration of `yVarP`, which is the pointer that is used to access attributes in the state machine. `yVarP` should be of type `VDEF_TYPE *`, where `VDEF_TYPE` is the type of the `yVDef` struct for the state machine. 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

This macro defines local variables in the function implementing the behavior of an operation in an active class.

### YPRD_YVARP

Parameters: `(VDEF_TYPE)`

This macro is used within an operation in an active class. It should be expanded to a declaration of `yVarP`, which is the pointer that is used to access attributes in the active class. `yVarP` should be of type `VDEF_TYPE *`, where `VDEF_TYPE` is the type of the `yVDef` struct for the state machine. If the pointer to the `yVDef` struct is passed as parameter to the operation function, `yVarP` can be assigned its correct value already in the declaration.

## Macros used within PAD functions

### BEGIN_PAD

Parameters: `(VDEF_TYPE)`

BEGIN_PAD is a macro that can be used to insert code that is executed in the beginning of the PAD function. `VDEF_TYPE` is the `yVDef type` for the state machine.

### BEGIN_START_TRANSITION

Parameters: `(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 struct (prefixed with `ySignalPar`) for the startup signal for this state machine.

### CALL_SUPER_PAD_START

Parameters: `(PAD)`

During the start transition of a state machine, 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 attributes defined in the state machine. 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

Parameters: `(PRD, THISPRD)`

This macro is used in the same way as CALL_SUPER_PAD_START but for the start transition in an operation in an active class. `THISPRD` is the executing function, while `PRD` is the inherited function.

### LOOP_LABEL

The LOOP_LABEL macro should be used to form the loop from a nextstate operation to the next signal receipt operation necessary in the OS where OS tasks do not perform return at end of transition (which is the case for many OS). This macro is also suitable to handle free on received signals and the treatment of the save queue. In an OS where nextstate is implemented using a C `return` the LOOP_LABEL macro is usually empty.

### LOOP_LABEL_PRD

This macro is similar to LOOP_LABEL, but is used in operations in active classes that contain states.

### LOOP_LABEL_PRD_NOSTATE

This macro is similar to LOOP_LABEL but used in operations in active classes that do not contain states. This macro is usually expanded to nothing.

### 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.

### XEND_PRD

This is a macro generated at the end of a function that represents the behavior of an operation in an active class. It does not have 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

This macro should usually be expanded to the type `xPrsNode`.

### XNAMENODE

This macro reaches the `xPrsIdNode` from a PAD function. Normally this is `yVarP->NameNode`.

### XNAMENODE_PRD

This macro reaches the `xPrdIdNode` from a PRD function. Normally this is `yPrdVarP->NameNode`.

### YPAD_FUNCTION

Parameters: `(PAD)`

This macro gives the function heading of the PAD function that is given as parameter.

### YPAD_PROTOTYPE

Parameters: `(PAD)`

This macro gives the function prototype of the PAD function that is given as parameter.

### YPRD_FUNCTION

Parameters: `(PRD)`

This macro gives the function heading of the PRD function that is given as parameter.

### YPRD_PROTOTYPE

Parameters: `(PRD)`

This macro gives the function prototype of the PRD function that is given as parameter.

## Macros for the yInit function

### BEGIN_YINIT

This macro is placed in the beginning of the `yInit` function. It can be expanded to variable declarations and initialization code.

### XPROCESSDEF_C

Parameters: `(PROC_NAME, PROC_NAME_STRING, PREFIX_PROC_NAME, PAD_FUNCTION, VDEF_TYPE)`

This macro can be used to introduce code for each instance set of active classes.

- `PROC_NAME`
  The name of the active class without prefix.
- `PROC_NAME_STRING`
  The name of the active class as a character string.
- `PREFIX_PROC_NAME`
  The name of the active class with prefix.
- `PAD_FUNCTION`
  The PAD function for this active class instance set.
- `VDEF_TYPE`
  The `yVDef` struct for this active class.

### XPROCESSDEF_H

Parameters: `(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 active class instance set.

- `PROC_NAME`
  The name of the active class without prefix.
- `PROC_NAME_STRING`
  The name of the active class as a character string.
- `PREFIX_PROC_NAME`
  The name of the active class with prefix.
- `PAD_FUNCTION`
  The PAD function for this active class instance set.
- `VDEF_TYPE`
  The `yVDef` struct for this active class.

### xInsertIdNode

In the `yInit` function the function `xInsertIdNode` is called for each `IdNode`. In an application this is not necessary, and the macro `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 signal sending

### ALLOC_SIGNAL

This macro is used together with ALLOC_SIGNAL_PAR.

### ALLOC_SIGNAL_PAR

Parameters: `(SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_TYPE)`

This macro, together with ALLOC_SIGNAL, is 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 referenced by the variable mentioned by the macro OUTSIGNAL_DATA_PTR.

- `SIG_NAME`
  The name of the signal without prefix.
- `SIG_IDNODE`
  The `xSignalIdNode` of the signal.
- `RECEIVER`
  This is the receiver given in a direct addressing clause, or the calculated receiver. In a NO_TO signal sending, `RECEIVER` is `xNotDefPId`.
- `SIG_PAR_TYPE`
  This is the type of the signal (prefixed with `ySignalPar`). If the signal has no parameters this macro parameter is XSIGNALHEADERTYPE.

### INSIGNAL_NAME

This macro should be expanded to the identification of the currently received signal. It is used to distinguish between signals when several signals are enumerated in the same signal receipt symbol.

### OUTSIGNAL_DATA_PTR

This should be the pointer referring to the signal data area while building the signal during a signal sending. 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.

### SDL_2OUTPUT

This macro is related to SDL_ALT2OUTPUT_COMPUTED_TO.

### SDL_2OUTPUT_NO_TO

This macro is related to SDL_ALT2OUTPUT_COMPUTED_TO.

### SDL_2OUTPUT_COMPUTED_TO

This macro is related to SDL_ALT2OUTPUT_COMPUTED_TO.

### SDL_ALT2OUTPUT

This macro is related to SDL_ALT2OUTPUT_COMPUTED_TO.

### SDL_ALT2OUTPUT_NO_TO

This macro is related to SDL_ALT2OUTPUT_COMPUTED_TO.

### SDL_ALT2OUTPUT_COMPUTED_TO

Parameters: `(PRIO, VIA, SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_SIZE, SIG_NAME_STRING)`

The six macros named SDL_*OUTPUT* 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 signal sending. The version without suffix is used for a signal sending with direct addressing, while the suffix _COMPUTED_TO is used for a signal sending without a direct addressing but it is possible to compute the receiver during code generation time. The suffix _NO_TO indicates a signal sending without direct addressing, where the receiver cannot be calculated during code generation time.

- `PRIO`
  For future use.
- `VIA`
  The via list given in the signal sending.
- `SIG_NAME`
  The name of the signal without prefix
- `SIG_IDNODE`
  The `xSignalIdNode` for the signal.
- `RECEIVER`
  The receiver given (<SENDER>.<OUTPUT>), or calculated. In a NO_TO signal sending, `RECEIVER` is `xNotDefPId`.
- `SIG_PAR_SIZE`
  The size of the struct (prefixed with `ySignalPar`) 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 a signal sending when the receiver is THIS, the RECEIVER parameter in the ALLOC_SIGNAL and SDL_2OUTPUT macros will become SDL_THIS.

### SIGCODE

Parameters: `(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

Parameters: `(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`.

- `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 of such components are: sender, receiver, signal type.

### TO_PROCESS

Parameters: `(PROC_NAME, PROC_IDNODE)`

This macro is used as RECEIVER in the ALLOC_SIGNAL and SDL_2OUTPUT macros if the signal is sent to an active class instance set.

- `PROC_NAME`
  This is the name of the receiving active class without prefix.
- `PROC_IDNODE`
  This is the `xPrsIdNode` of the receiving active class.

### TRANSFER_SIGNAL

This macro is related to TRANSFER_SIGNAL_PAR, below.

### TRANSFER_SIGNAL_PAR

Parameters: `(SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_TYPE)`

These macros are used as alternatives for the ALLOC_SIGNAL macros if the directive `#TRANSFER` is given in the signal sending.

- `SIG_NAME`
  This is the name of the signal without prefix.
- `SIG_IDNODE`
  This is the `xSignalIdNode` of the signal.
- `RECEIVER`
  This is the receiver given in a direct addressing clause, or calculated. In a NO_TO signal sending, `RECEIVER` is `xNotDefPId`.
- `SIG_PAR_TYPE`
  This is the type (prefixed with `ySignalPar`) of the signal. If the signal has no parameters this macro parameter is XSIGNALHEADERTYPE.

### XNONE_SIGNAL

This is the representation of a none signal.

### XSIGNALHEADERTYPE

This macro is used to indicate a struct (prefixed with `ySignalPar`) for a signal without parameters. Such a signal has no generated 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 call of remote operations

### ALLOC_REPLY_SIGNAL

This macro is related to ALLOC_REPLY_SIGNAL_PRD_PAR.

### ALLOC_REPLY_SIGNAL_PAR

This macro is related to ALLOC_REPLY_SIGNAL_PRD_PAR.

### ALLOC_REPLY_SIGNAL_PRD

This macro is related to ALLOC_REPLY_SIGNAL_PRD_PAR.

### ALLOC_REPLY_SIGNAL_PRD_PAR

Parameters: `(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 an operation in an active class.

- `SIG_NAME`
  The reply signal name without prefix.
- `SIG_IDNODE`
  The `xSignalIdNode` for the reply signal.
- `RECEIVER`
  The receiver of the reply signal. The macros XRPC_SENDER_IN_ALLOC and XRPC_SENDER_IN_ALLOC_PRD are used as actual parameters. The suffix _PRD is used if the implicit RPC transition is part of an operation in an active class.

- `SIG_PAR_TYPE`
  The type for the reply signal (prefixed with `ySignalPar`). If the reply signal does not contain any parameters the macro name XSIGNALHEADERTYPE is generated as actual parameter.

### REPLYSIGNAL_DATA_PTR

This macro is related to REPLYSIGNAL_DATA_PTR_PRD.

### 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 an operation in an active class.

### SDL_RPCWAIT_NEXTSTATE

This macro is related to SDL_RPCWAIT_NEXTSTATE_PRD.

### SDL_RPCWAIT_NEXTSTATE_PRD

Parameters: `(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 an operation in an active class.

- `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

Parameters: `(PRIO, VIA, SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_SIZE, SIG_NAME_STRING)`

This macro sends the call signal of an RPC.

- `PRIO`
  For future use.

- `VIA`
  The via list, which in this case always is `(xIdNode *)0`, that is 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 direct addressing expression, or, in case of no explicit receiver specified in the call using the macro, it becomes XGETEXPORTINGPRS.
- `SIG_PAR_SIZE`
  The size of the struct (prefixed with `ySignalPar`) 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

This macro is related to SDL_2OUTPUT_RPC_REPLY_PRD.

### SDL_2OUTPUT_RPC_REPLY_PRD

Parameters: `(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 an operation in an active class.

- `PRIO`
  For future use.
- `VIA`
  The via list, which in this case is always `(xIdNode *)0`, that is 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 macros XRPC_SENDER_IN_OUTPUT or XRPC_SENDER_IN_OUTPUT_PRD.

- `SIG_PAR_SIZE`
  The size of the struct (prefixed with `ySignalPar`) 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

Parameters: `(REMOTENODE)`

This macro should be expanded to an expression that, given the remote operation in an active class as actual macro parameter (more precisely, the `IdNode` for the remote operation), returns one possible "provider" of this remote operation. Usually this macro is expanded to a call of the library function `xGetExportingPrs`.

### XRPC_REPLY_INPUT

This macro is related to XRPC_REPLY_INPUT_PRD.

### 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

This macro is related to XRPC_SAVE_SENDER_PRD.

### XRPC_SAVE_SENDER_PRD

This macro 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 an operation in an active class.

### XRPC_SENDER_IN_ALLOC

This macro is related to XRPC_SENDER_IN_ALLOC_PRD.

### XRPC_SENDER_IN_ALLOC_PRD

This macro is 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 an operation in an active class.

### XRPC_SENDER_IN_OUTPUT

This macro is related to XRPC_SENDER_IN_OUTPUT_PRD.

### XRPC_SENDER_IN_OUTPUT_PRD

This macro is 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 an operation in an active class.

### XRPC_WAIT_STATE

The state number used for a RPC wait state. XRPC_WAIT_STATE is usually defined as -3.

## Implementation of static and dynamic create and stop

### ALLOC_STARTUP

This macro is related to ALLOC_STARTUP_PAR.

### ALLOC_STARTUP_PAR

Parameters: `(PROC_NAME, STARTUP_IDNODE, STARTUP_PAR_TYPE)`

This macro allocates the data area for a startup signal and lets 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.

- `PROC_NAME`
  The name without prefix for the created active class.
- `STARTUP_IDNODE`
  The `xSignalIdNode` for the startup signal of the created active class.

- `STARTUP_PAR_TYPE`
  The type (prefixed with `ySignalPar`) for the startup signal of the created active class.

### ALLOC_STARTUP_THIS

This macro allocates the data area for a startup signal and lets 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

Parameters: `(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 called once for each active class instance set in the `yInit` function. It should be used to initiate common features for all instances of an active class instance set.

- `PROC_NAME`
  The name without prefix for the active class instance set.
- `PREFIX_PROC_NAME`
  The name with prefix for the active class instance set.
- `PROC_IDNODE`
  The `xPrsIdNode` for the active class instance set.
- `PROC_NAME_STRING`
  The name as character string for the active class instance set.
- `MAX_NO_OF_INST`
  The maximum number of instances of this active class instance set.
- `STATIC_INST`
  The number of static instances of this active class instance set.
- `VDEF_TYPE`
  The `yVDef` type for this active class instance set.
- `PRIO`
  For future use.
- `PAD_FUNCTION`
  the PAD for this active class instance set.

### SDL_CREATE

Parameters: `(PROC_NAME, PROC_IDNODE, PROC_NAME_STRING)`

This macro is used to create (a create action) an instance of an active class.

- `PROC_NAME`
  The name without prefix for the active class instance set.
- `PROC_IDNODE`
  The `xPrsIdNode` for the active class instance set.
- `PROC_NAME_STRING`
  The name as character string for the active class instance set.

### SDL_CREATE_THIS

This macro is used to implement the creation of THIS.

### SDL_STATIC_CREATE

Parameters: `(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" active class instance that should be created of an active class instance set.

- `PROC_NAME`
  The name without prefix for the active class instance set.
- `PREFIX_PROC_NAME`
  The name with prefix for the active class instance set.
- `PROC_IDNODE`
  The `xPrsIdNode` for the active class instance set.
- `PROC_NAME_STRING`
  The name as character string for the active class instance set.
- `STARTUP_IDNODE`
  The `xSignalIdNode` for the startup signal for the active class instance set.
- `STARTUP_PAR_TYPE`
  The type (prefixed with `ySignalPar`) for the startup signal for the active class instance set.
- `VDEF_TYPE`
  The `yVDef` type for the active class instance set.
- `PRIO`
  For future use.

- `PAD_FUNCTION`
  The PAD function for the active class instance set.
- `BLOCK_INST_NUMBER`
  If this active class instance set is part if an active class instance set that contains a composition, then this macro is the instance number for the active class instance set that it belongs to. Otherwise this macro parameter is 1.

### SDL_STOP

This macro is used to implement the operation Stop on active classes.

### 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 formal parameter 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 structures SIGNAL_VARS will be followed by STARTUP_VARS.

## Implementation of timers, timer operations and now

### ALLOC_TIMER_SIGNAL_PAR

Parameters: `(TIMER_NAME, TIMER_IDNODE, TIMER_PAR_TYPE)`

This macro allocates a data area for the timer signal with parameters.

- `TIMER_NAME`
  The name without prefix of the timer.
- `TIMER_IDNODE`
  The `xSignalIdNode` for the timer.
- `TIMER_PAR_TYPE`
  The type (prefixed with `ySignalPar`) for the timer.

### DEF_TIMER_VAR

This macro is related to DEF_TIMER_VAR_PARA.

### DEF_TIMER_VAR_PARA

Parameters: `(TIMER_VAR)`

There will be one application of this macro in the `yVDef` type for the active class for each timer declaration the active class 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

This macro is related to INIT_TIMER_VAR_PARA.

### INIT_TIMER_VAR_PARA

Parameters: `(TIMER_VAR)`

This macro will be inserted in start transitions, during initialization of active class attributes. 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

This macro is related to INPUT_TIMER_VAR_PARA.

### INPUT_TIMER_VAR_PARA

Parameters: `(TIMER_VAR)`

This macro will be inserted when receiving 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**

*If a timer signal is received in an input * statement, no INPUT_TIMER_VAR will be present.*

### RELEASE_TIMER_VAR

This macro is related to RELEASE_TIMER_VAR_PARA.

### RELEASE_TIMER_VAR_PARA

Parameters: `(TIMER_VAR)`

This macro 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

Parameters: `(TIMER_NAME, TIMER_IDNODE, TIMER_VAR)`

This macro is used to implement the operation Active on a timer. Active on timers with parameters is not implemented in the C Code Generator.

- `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.

### SDL_RESET

Parameters: `(TIMER_NAME, TIMER_IDNODE, TIMER_VAR, TIMER_NAME_STRING)`

This macro is used to implement the operation Reset on a timer without 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

Parameters: `(EQ_FUNC, TIMER_VAR, TIMER_NAME_STRING)`

This macro is used to implement the operation Reset on a timer with parameters.

- `EQ_FUNC`
  The name of the generated equal function that can test if two timer instances 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

Parameters: `(TIME_EXPR, TIMER_NAME, TIMER_IDNODE, TIMER_VAR, TIMER_NAME_STRING)`

This macro is related to SDL_SET_TICKS_WITH_PARA.

### SDL_SET_WITH_PARA

Parameters: `(TIME_EXPR, TIMER_NAME, TIMER_IDNODE, TIMER_PAR_TYPE, EQ_FUNC, TIMER_VAR, TIMER_NAME_STRING)`

This macro is related to SDL_SET_TICKS_WITH_PARA.

**SDL_SET_DUR**

Parameters: `(TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE, TIMER_VAR, TIMER_NAME_STRING)`

This macro is related to SDL_SET_TICKS_WITH_PARA.

**SDL_SET_DUR_WITH_PARA**

Parameters: `(TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE, TIMER_PAR_TYPE, EQ_FUNC, TIMER_VAR, TIMER_NAME_STRING)`

This macro is related to SDL_SET_TICKS_WITH_PARA.

**SDL_SET_TICKS**

Parameters: `(TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE, TIMER_VAR, TIMER_NAME_STRING)`

This macro is related to SDL_SET_TICKS_WITH_PARA.

**SDL_SET_TICKS_WITH_PARA**

Parameters: `(TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE, TIMER_PAR_TYPE, EQ_FUNC, TIMER_VAR, TIMER_NAME_STRING)`

The macros with prefix SDL_SET are used to implement the 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 a call of the macro ALLOC_TIMER_SIGNAL_PAR, 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 operation returning a duration value. In this case, both the time and the duration values are available as macro 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 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 the 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 of such components are: sender, receiver, timer type.

Since timers are regarded as signals once they have been sent, TIMER_VARS has to be identical to SIGNAL_VARS, with the addition that new components may be added last in TIMER_VARS, after the components they have in common.

### XTIMERHEADERTYPE

This macro is used to indicate a struct for a timer without parameters. Such a timer has no generated struct with parameters. 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

Parameters: `(PROC_NAME, PROC_IDNODE, VAR_SIZE)`

This macro allocates a data area (`yVDef`) for the called operation in an active class.

- `PROC_NAME`
  The name of operation with prefix.
- `PROC_IDNODE`
  The `xPrdIdNode` of the called operation.
- `VAR_SIZE`
  The size of the `yVDef` struct for the operation.

### ALLOC_THIS_PROCEDURE

This macro allocates a data area (`yVDef`) for an operation when call THIS is used.

### ALLOC_VIRT_PROCEDURE

`(PROC_IDNODE)`

This macro allocates a data area (`yVDef`) for the called operation in an active class when calling a virtual operation. The `PROC_IDNODE` parameter is `xPrdIdNode` for the called operation.

### CALL_PROCEDURE

This macro is related to CALL_PROCEDURE_IN_PRD.

### CALL_PROCEDURE_IN_PRD

Parameters: `(PROC_NAME, PROC_IDNODE, LEVELS, RESTARTADDR)`

This macro is 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 call of the operation is made in an operation in an active class.

- `PROC_NAME`
  The name of operation with prefix, which is the same as the name of the C function representing the behavior of the operation.
- `PROC_IDNODE`
  The `xPrdIdNode` of the called operation.
- `LEVELS`
  The scope level between the caller and the called operation.
- `RESTARTADDR`
  This is the restart address for the symbol after the operation call.

### CALL_PROCEDURE_STARTUP

This macro is related to CALL_PROCEDURE_STARTUP_SRV.

### CALL_PROCEDURE_STARTUP_SRV

This macro is only of interest if the PAD functions are left via a return at the end of transitions. In that case any outstanding operation in an active class must be restarted when the state machine becomes “active” again.

### CALL_THIS_PROCEDURE

Parameters: `(RESTARTADDR)`

This macro is used to implement a call on THIS operation. `RESTARTADDR` is the restart address for the symbol after the operation call.

### CALL_VIRT_PROCEDURE

This macro is related to CALL_VIRT_PROCEDURE_IN_PRD.

### CALL_VIRT_PROCEDURE_IN_PRD

Parameters: `(PROC_IDNODE, LEVELS, RESTARTADDR)`

This macro is used to implement a call operation on a virtual operation. 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 operation call is made in an operation in an active class.

- `PROC_IDNODE`
  The `xPrdIdNode` of the called operation.
- `LEVELS`
  The scope level between the caller and the called operation.
- `RESTARTADDR`
  This is the restart address for the symbol after the operation call.

### PROCEDURE_ALLOC_ERROR

This macro is inserted after the ALLOC_PROCEDURE macro and the assignment of parameter values to the operation 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 operation data area. It should be assigned in the macro ALLOC_PROCEDURE and will be used to assign the actual operation parameters.

### SDL_RETURN

The implementation of Return.

### XNOPROCATSTARTUP

If this macro is defined then all the code discussed for the macro CALL_PROCEDURE_STARTUP (just above) is removed.

## Implementation of join

Join statements are normally implemented as `goto` 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

Parameters: `(RESTARTADDR,RESTARTPAD)`

This macro is related to XJOIN_SUPER_SRV, below.

### XJOIN_SUPER_PRD

Parameters: `(RESTARTADDR,RESTARTPRD)`

This macro is related to XJOIN_SUPER_SRV, below.

### XJOIN_SUPER_SRV

Parameters: `(RESTARTADDR,RESTARTSRV)`

The macros with prefix XJOIN_SUPER_SRV represent join to super types of active classes and operations, in that order.

- `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 an operation in an active class. START_STATE_PRD should be defined as 0.

### SDL_NEXTSTATE

Parameters: `(STATE_NAME, PREFIX_STATE_NAME, STATE_NAME_STRING)`

Nextstate operation in a state machine of the given state.

- `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

Dashed nextstate operation in a state machine.

### SDL_NEXTSTATE_PRD

Parameters: `(STATE_NAME, PREFIX_STATE_NAME, STATE_NAME_STRING)`

Nextstate operation (in operation in active class) of the given state.

- `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

Dashed nextstate operation in an operation in active class.

## Implementation of any decisions

An any decision (non-deterministic 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

Parameters: `(NO_OF_PATHS)`

Start of the any decision. `NO_OF_PATHS` is the number of paths in the decision.

### BEGIN_ANY_PATH

Parameters: `(PATH_NO)`

A path (not the first) in the implementation part of the any decision. `PATH_NO` is the path number.

### BEGIN_FIRST_ANY_PATH

Parameters: `(PATH_NO)`

The first possible path in the implementation part of the any decision. `PATH_NO` is the path number.

### DEF_ANY_PATH

Parameters: `(PATH_NO, SYMBOLNUMBER)`

Definition of a path in the decision.

- `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**
Parameters: (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 is similar to any decisions.

**BEGIN_FIRST_INFORMAL_PATH**
Parameters: (PATH_NO)

The first possible path in implementation part of the informal decision. PATH_NO is the path number.

**BEGIN_INFORMAL_DECISION**
Parameters: (NO_OF_PATHS, QUESTION)

This is the start of the informal decision.

- NO_OF_PATHS
  The number of paths in the decision.
- QUESTION
  The string constant that is printed.

**BEGIN_INFORMAL_ELSE_PATH**
Parameters: (PATH_NO)

The else path in implementation part of the informal decision. PATH_NO is the path number.

**BEGIN_INFORMAL_PATH**

Parameters: `(PATH_NO)`

A path in implementation part of the informal decision. `PATH_NO` is the path number.

**DEF_INFORMAL_PATH**

Parameters: `(PATH_NO, ANSWER, SYMBOLNUMBER)`

Definition of a path in the informal decision.

- `PATH_NO`
  The path number.
- `ANSWER`
  The answer string.
- `SYMBOLNUMBER`
  The symbol number for the first symbol in this path.

**DEF_INFORMAL_ELSE_PATH**

Parameters: `(PATH_NO, SYMBOLNUMBER)`

Definition of the else path in the informal decision.

- `PATH_NO`
  The path number.
- `SYMBOLNUMBER`
  The symbol number for the first symbol in this path.

**END_DEFS_INFORMAL_PATH**

Parameters: `(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 test the validity of for example a component selection of a `choice` variable. Also tests for optional components in structures and for de-referencing of pointers is treated here.

### XCHECK_CHOICE_USAGE

Parameters: `(TAG, VALUE, NEQTAG, COMPNAME, CURR_VALUE, TYPEINFO)`

This macro is related to XSET_CHOICE_TAG_FREE.

### XSET_CHOICE_TAG

Parameters: `(TAG, VALUE, ASSTAG, NEQTAG, COMPNAME, CURR_VALUE, TYPEINFO)`

This macro is related to XSET_CHOICE_TAG_FREE.

### XSET_CHOICE_TAG_FREE

Parameters: `(TAG, VALUE, ASSTAG, NEQTAG, FREEFUNC, COMPNAME, CURR_VALUE, TYPEINFO)`

The macros with prefix XSET_CHOICE 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.

- 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

Parameters: `(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 component's name as a char string.

### XCHECK_REF

This macro is related to XCHECK_OREF.

### XCHECK_OWN

This macro is related to XCHECK_OREF.

### XCHECK_OREF

Parameters: `(VALUE, REF_TYPEINFO, REF_SORT)`

These macros are used to implement a test that null pointers (using the Own or ORef template) are not de-referenced. These macros are inserted before each statement containing a Own/ORef pointer de-referencing. In case of an ORef pointer it is also checked that the ORef is valid, that is it refers to an object owned by the current active class.

- VALUE
  This is the value of the pointer.
- REF_TYPEINFO
  The type info node for the referenced sort.
- REF_SORT
  The C type that corresponds to the referenced instantiation passive class.

### XCHECK_OREF2

Parameters: `(VALUE)`

Checks that an ORef pointer is a valid pointer, that is NULL, or that it refers to an object owned by the current active class.

## Debug and simulation macros

### XAFTER_VALUE_RET_PRDCALL

Parameters: `(SYMB_NO)`

This is a macro generated between the implementation of a value returning operation call (implicit call symbol), and the symbol containing the call to the value returning operation.

`SYMB_NO` is the symbol number of the symbol containing the value returning operation call.

### XAT_FIRST_SYMBOL

Parameters: `(SYMB_NO)`

This is a macro generated between a signal receipt 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

Related to XBETWEEN_STMTS_PRD.

### XBETWEEN_STMTS_PRD

Parameters: `(SYMB_NO, C_LINE_NO)`

A macro generated between statements in an action. The suffix _PRD indicates that these statements are part of an operation in an active class.

- `SYMB_NO`
  The symbol number of the next statement.

- C_LINE_NO
  Line number in C of this statement.

### XBETWEEN_SYMBOLS

Related to XBETWEEN_SYMBOLS_PRD.

### XBETWEEN_SYMBOLS_PRD

Parameters: (SYMB_NO, C_LINE_NO)

A macro generated between symbols in a transition. The suffix _PRD indicates that these symbols are part of an operation in an active class.

- SYMB_NO
  The symbol number of the next symbol.
- C_LINE_NO
  The line number in the C code for this statement.

### XDEBUG_LABEL

Parameters: (LABEL_NAME)

This macro gives the possibility to insert labels at the beginning of transitions. Such labels can be useful during debugging. The LABEL_NAME parameter is a concatenation of the state name and the signal name.

**Example 390**

The '*' that appears in the statements

"STATE *;"

"INPUT *;"

will have the name ASTERISK appear instead.

```
state State1; input Sig1;
state State2; input *;
state *; input Sig2;
```

In the generated code for these 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: ;
```

---

**XOS_TRACE_INPUT**

Parameters: `(SIG_NAME_STRING)`

This macro is generated as input statements and can, for example, be used to generate trace information about received signals. The `SIG_NAME_STRING` parameter is the name of the signal.

**YPRSNAME_VAR**

Parameters: `(PRS_NAME_STRING)`

This macro is generated among the declarations of variables in the PAD function for an active class. It can, for example, be used to declare a C `char *` variable containing the name of the active class. Such a variable can be useful during debugging. The `PRS_NAME_STRING` parameter is the name of the active class as a character string.

**YPRDNAME_VAR**

Parameters: `(PRD_NAME_STRING)`

This macro is generated among the declarations of variables in the PRD function for an operation in an active class. It can, for example, be used to declare a `char*` variable containing the name of the operation. Such a variable can be useful during debugging. The `PRD_NAME_STRING` parameter is the name of the operation 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.

#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 XCTRACE
#define xCTrace(p)  p,
#define xCTraceS(p)  p;
#else
#define xCTrace(p)
#define xCTraceS(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)

#define xbool int
```

### MAX_READ_LENGTH

This macro controls the length of the `char *` buffers used to read values of sorts. 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.

```
#ifndef MAX_READ_LENGTH
#define MAX_READ_LENGTH 5000
                        /* max length of input line */
#endif
```

### SDL_NULL

A null value for the type Pid.

### xNotDefPId

This is used as RECEIVER parameter in the SDL_2OUTPUT macros. The section "Implementation of signals and signal sending" on page 1233 contain examples of how this macro is used.

## Macros for threaded integrations

The following macros are used for threaded integrations only.

### THREADED

Main macro for the threaded integration model defining the kernel specifics.

### 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**

Waits for `xInitSem` to be released.

**THREADED_LOCK_INPUTPORT**

Protects the input queue by taking the semaphore.

**THREADED_UNLOCK_INPUTPORT**

Releases the semaphore for the input queue.

**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 access and actions on global data by taking a semaphore

**THREADED_EXPORT_END**

Release the semaphore after access and actions on global data.

**THREADED_START_THREAD**

Starts a new thread.

### THREADED_STOP_THREAD

Terminates a thread.

### THREADED_AFTER_THREAD_START

Synchronizes the start-up of newly created threads.

# 39

# *AgileC Code Generator Reference*

AgileC Code Generator is intended to be used to develop applications on embedded systems. It is of course possible to use AgileC Code Generator for other types of applications as well, but its features are designed to use when developing embedded systems.

An application generated by AgileC Code Generator can be executed in two modes, Bare and Threaded.

# File Structure

An application generated by the AgileC Code Generator consists and depends on a number of files.

The generated code reflects the contents and behavior of the system described by the UML 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 C files generated from the UML model are placed.

- `.m`

  This is the makefile for compiling the set of files generated from the UML 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 AgileC 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 UML 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 AgileC Code Generator. It contains information about sizes and about what UML 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.

- `uml_cfg.h`

  This is a configuration file generated by the Application Builder. 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/agilec/kernel`

- `uml_kern.c` and `uml_kern.h`

  These files are the basic implementation of the run-time kernel. The file `uml_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

There are two additional files that are important for building an executable. These files are `comp.opt` and `makeoptions` or `make.opt`.

- `comp.opt`

  When a **Target Kind** is selected for a Build Artifact, it is actually a directory that is selected. The AgileC Code Generator reads the `comp.opt` file in this directory and uses the information in this file to determine how to generate the "makefile".

  `comp.opt` also contains templates for invoking the compiler, the linker and so on.

- `makeoptions` or `make.opt`

  This file contains information about compilation switches and how to compile the kernel. This file is included in the generated "makefile" for the system. The name of the file and the syntax for how to include, is defined in the beginning of `comp.opt`

## Include structure for C files

### uml_kern.h

The top level for definitions is the `uml_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.
- `uml_cfg.h`

- `auto_cfg.h`
- `comphdef.h` (compiler/hardware integration)
- `rtapidef.h` (run-time API integration)
- `sctpred.h`

### uml_kern.c

Include is also used for `.c` files. In this way the complete kernel will be compiled when the top element, `uml_kern.c`, is compiled.

- `uml_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 UML 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 UML system should for example cause a hardware register to be set or a TCP/IP packet to be sent over Internet to 'somewhere in the Internet world'. 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 Application Builder 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 `uml_cfg.h` generated by the application builder.

## 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 `uml_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 `uml_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 `uml_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 not using 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 interrupt 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 `uml_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 AgileC 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 application, 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 environment functions. In the generated code for the system, except the `.ifc` file, all UML name are prefixed or suffixed to make the names unique in C. In the `.ifc` file the UML 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 "uml_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 `uml_cfg.h` file generated by the Application Builder. 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 `uml_cfg.h` file by the Application Builder.

### 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 UML. 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 `uml_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

## Essential files

The essential files used for compilation and linking of an AgileC Code Generator application are

- **`comp.opt`**: controlling syntax of different commands
- **`make.opt`** or **`makeoptions`**: Contains compiler flags and compilation of the kernel. This file is included by the generated makefile.
- **`systemname.m`**: This is the generated makefile for the application.
- **`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 is used to control the complete build process. When you select a "Target Kind" in the Application Builder, this information is used to refer to a directory containing such a `comp.opt` file. The code generator will read this file during code generation time and will use the information according to the following.

The `comp.opt` file consists of five important lines (plus lines counted as comments). The syntax for the file is described in "Library files" on page 1065, where a more detailed description of the complete make process can also be found. Below an overview for AgileC Code Generator can be found.

- **Line 1**: The syntax for including `make.opt` in the generated makefile for the application. The code generator copies this line at the start of the generated makefile.
- **Line 2**: Template for a compilation command. This is used by the code generator to generate proper compilation commands in the generated makefile.
- **Line 3**: Template for link command. This is used by the code generator to generate a link command in the generated makefile.
- **Line 4**: The command to be executed to start the make process. This should be a shell command that the code generator 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 something 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 commands for the kernel files. In the generated makefile it will then find the compilation 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. You can in the Application Builder specify what template makefile that should be used. 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 and modify them. In the Application Builder the new directory is then selected as kernel.

# 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" on page 1283.

### 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 `uml_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 (`uml_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, according to the list below, in the `uml_cfg.h` file generated by the Application Builder, 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.

The define should be added in the Build Artifact in the **Extra code** / **Head** field. This will put the statement into the `uml_cfg.h` file.

```
#define USER_CFG_COMPHDEF
```

Another option, which is 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 `uml_cfg.h` (or as a compiler option):

```
#define USER_CFG_RTAPIDEF
```

In that case the file `rtapidef.h` will be included in `uml_kern.h` and `rtapidef.c` will be included in `uml_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 UML 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 UML 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 be specified in the Application Builder. 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 `uml_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 a 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 `rtapidef.h` define two macros 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 391: 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

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 392: 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 393: 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 394: 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 395: 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 AgileC 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 UML 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" on page 1314. 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: `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 396:** ———————————————————————————————

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, 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. Together with the definition of the Using Thread Artifacts, 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. 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 397**

```
#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 398**

```
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 399**

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
      /* 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 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 AgileC 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 400: 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);
```

# MISRA coding rules

The code in the AgileC Code Generator run-time kernel and code generated by the AgileC Code Generator is to a large extent compliant with the MISRA coding rules.

AgileC Code Generator code complies with 121 of the total of 141 MISRA rule with some obvious UML restrictions (like if goto is not allowed in C, then goto should not be used in UML). With some more non-obvious restrictions the code from the AgileC Code Generator complies with an additional six rules. The AgileC Code Generator is not compliant with the remaining six required and eight advisory rules.

To enable generation of MISRA compliant code the AgileC Code Generator option “Generate MISRA compliant code” should be checked. The “Name mangling” option in “Code generation properties” should also be set to Prefix (to assure that names are unique within the 31 first characters, rule 5.1).

Enabling MISRA compliant code will change the some aspects of the generated code to make it MISRA compliant. The most significant is the way stop actions are translated. If there are several stop actions inside a diagram, the code with MISRA compliance is somewhat larger, as goto statements are not allowed.

Turning on MISRA compliance will also enable a number of MISRA checks, that will warn for problems in the UML model that could cause the generated C code to become non-compliant with some MISRA rule.

## Obvious restrictions in UML

It is assumed that the user follows obvious restrictions in UML, for example if there is a restriction for a certain concept in C then the same restriction should be applied to similar concepts in UML. Some more examples:

- As goto is not allowed in C, the user should not introduce goto statements or joining flows (which is a graphical goto) in UML. Rule 14.4.
- As functions should have a single point of exit, the user should write operations in UML in that way. Rule 14.7.
- The requirements on for loops in rule 13.4, 13.5, and 13.6 should be directly mapped to for loops in UML.

- An iteration should only contain at most one break. This rule should be followed in UML as well. Rule 14.6.
- Recursion is not allowed according to rule 16.2. This applies directly to UML operations.
- According to rule 18.4 unions should not be used. This implies that the counterpart in UML, "`choice`" should not be used.

As it is possible to include C code written by the user inside the generated code from the AgileC Code Generator, it is of course also assumed that such code does not violate any of the MISRA rules. The same is assumed for C code imported into the tool.

## Non-obvious restrictions in UML

The table below list rules that are violated in a general AgileC Code Generator application. By not using some advanced data types the generated code can be made compliant with these rules:

| Rule # | Rule |
|---|---|
| 11.1 | Do not perform cast from a "function type" to another "function type". |
| 12.10 | Do not use the comma operator |
| 16.1 | Do not use functions with variable number of arguments |
| 16.2 | Do not use recursion |
| 18.4 | Do not use unions |
| 20.4 | Do not use dynamic memory allocation |

**Rules that are violated in a general AgileC Code Generator application**

It is possible for a generated application to become compliant by following the rules below:

1. Do not use the data types and templates mentioned below.
   - `Bit_string`, `Octet_string`, `Object_identifier`
   - `String`, `Bag`, `Own`, `Ref`
   - `array` should only be used if the index type is a reasonable type for mapping to an `Array` in C (for example: enumeration types including character types and `Boolean` or a `syntype` with a limited range for the types just mentioned plus the `integer` types).
   - `Powerset` should only be used with a component type matching the description for index types in `Array` discussed above.
2. All non-simple attributes in active and passive classes should be “part”, as otherwise references to objects are introduced.
3. Do not use multiplicity for attributes, except for attributes that are active classes. Instead use `Array` or `CArray`.
4. Do not use operations returning a passive class, `Array`, `CArray`, or `Powerset`. This includes the make operator.
5. Do not use destructors for passive classes

Rule 20.4 (no dynamic memory allocation) puts two additional requirements on the configuration of AgileC:

6. 6. A maximum number of instances should always be specified for active classes.
7. 7. The user should set up the size of the signal data so all signal types fit. This is configuring `USER_CFG_MSG_BORDER_LEN`.

When it comes to rule 16.2 (No recursion) there are a few recursive functions in `sctpred.h`, but with the restrictions discussed above there will be no recursive calls.

## Violated rules

The following section contains rules that are violated and the reason for non-compliancy

**Rule 14.7: A function shall have a single point of exit at the end of the function.**

This coding standard has not been used during the development of the AgileC. There are pros and cons for such way of writing code. It simplifies code understanding to know that there is only one exit. However in many circumstances the control structure of the function will become more or even much more complex.

There is also one special case in the generated code where this rule is not really valid. Some functions in the generated code represent state machines. A state machine represents several different transitions, each with its own logical end. This logical end is implemented as a return in the function.

**Rule 15.2 An unconditional break shall terminate every non-empty switch clause.**

The purpose of this rule is (probably) to avoid fall through between switch clauses. Fall through is not used in the AgileC Code Generator. As return is allowed inside functions, the following rule is instead used:

**An unconditional break or an unconditional return shall terminate every non-empty switch clause.**

**Rule 17.1 Pointer arithmetic shall only be applied to pointers that address an array or array element.**
**Rule 17.4 Array indexing shall be the only allowed form of pointer arithmetic.**

To efficiently handle attributes of active class and at the same time not use dynamic memory, these rules are violated at a number of places in the AgileC Code Generator kernel.

**Rule 19.4 C macros shall only expand to a braced initializer, a constant, a parenthesized expression, a type qualifier, a storage class specifier, or a do-while-zero construct.**

This rule is violated. There are also a number of situations where code optimization is implemented using macros violating this rule.

### Rule 19.6 #undef shall not be used.

This is used in two situations in generated code. No simple solution to remove the `undef` statements has been found.

### Non-supported advisory rules

The advisory rules that AgileC code is not compliant with are: 5.6, 5.7, 6.3, 11.3, 11.4, 16.7, 19.1, 19.7.

# Optimization and Configuration

Optimization and configuration is mainly performed in two `.h` files. These are `auto_cfg.h` and `uml_cfg.h`. The file `auto_cfg.h` is generated by the AgileC Code Generator, and contains the optimization and configuration that can be automatically computed given the system that code is generated for. The `uml_cfg.h` file is generated by the Application Builder 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_PARTS 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_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 `uml_cfg.h`.

Example: 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
```

## uml_cfg.h

This file is generated by the Application Builder and contains information about the options that you have selected. In the Application Builder it is also possible to insert text written in a text field, in the `uml_cfg.h`. This feature can be used to insert macro definitions that can not be selected in the user interface.

`uml_cfg.h` is, just as `auto_cfg.h`, a file that should contain defines that are used to build the AgileC Code Generator application with specified options. The file `uml_cfg.h` will include `auto_cfg.h` in the beginning of the file.

The features discussed below can be selected in the Application Builder user interface and will affect the contents of the `uml_cfg.h` file.

### 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 UML level should be printed on `stdout`. Default is undefined.

**Note**

*The user interface for selecting features for AgileC Code Generator also includes a possibility to have free text that is just copied to the* `uml_cfg.h` *file. This feature should be used to include other defines that can not be directly selected in the user interface.*

### Selections in the Application Builder

There are some options in the Application Builder that is passed to the code generator and does not affect the `uml_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.

- Amount of **Comments** in generated code.

  In the code generated for state machines it is possible to select the amount of comments added to the code. Levels:

  – **1. Sparse**: Only some comments used to identify transitions are included.

  – **2. Structure**: As 1 plus a comment for each translated UML symbol.

  – **3. Explanation**: As 2 plus comments giving some explanations to the code.

- Include code for run-time tests in generated code.

  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.

- Include references to UML source in generated code as C comments.

  Such references can be used to navigate back to the UML source.

- Include code for textual execution trace in generated code.

  Selecting this feature will tell the code generator to include calls to trace functions in the generated code. This must be enabled if USER_CFG_UML_TRACE_STDOUT is enabled.

- **Name mangling** prefix or suffix to UML names.

  As the name scopes in UML and C are not the same it is not possible to use the UML names directly in the generated C code. By default the code generator will add a suffix to the UML name and use that as an identifier in C, to make certain that the C identifier is unique. If you have used long UML names and at the same time the C compiler only looks at a limited number of characters to determine if two names are the same, name clashes might occur. By using prefixes instead of suffixes such situations can be avoided.

## Some information about performance for different concepts

For small systems where performance is important there are some concepts that can cause performance problems. You may decide not to use these language concepts to improve the performance of the generated application.

The first and most important is unlimited number of instances for a part. That is, no upper limit is given for the maximum concurrent instances of the part. If an upper limit is given the code generator uses this to declare a static array for the instance data for the instances of this part. If no upper limit is given dynamic memory must be allocated for a suitable number of instances and when this memory is not enough, a `realloc` must be performed. A `realloc` means allocation of a new larger data area and copying of all the data in the old area to the new area. This makes create operations on parts with no upper limit for number of instances unpredictable, sometimes they are fast, but in some cases they might be very slow.

The save concepts will introduce some inefficiency, as the saved signals will be accessed every now and then. The same goes for the guard concept, as that implicitly requires the save concept. However, if these concepts are really helpful to implement the behavior of the system, it is probably better to use them then to rewrite the system to avoid using the concepts, as that might be even more expensive.

The `uml_cfh.h` file setting USER_CFG_MSG_BORDER_LEN might mean a lot for the performance of signal sending with parameters. If the parameters do not fit into the signal data area, a new data area must be allocated and later freed again. This will reduce the execution speed. However setting USER_CFG_MSG_BORDER_LEN to a higher value will make all signals larger, meaning waste of memory.

# Overview of Important Data Structures

All the important types discussed in this section are defined in `uml_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. |
| `yIni_Function` | A reference to a function that initializes the attributes 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 `uml_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. |
| `CurrentSymbolNr` | 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. |
| `ExternSignalQueue` | 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 priorities on signals. |
| `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. |

### Translation of Passive Classes

Passive classes in UML are translated to data types and operations in C. The details for are the same for AgileC Code Generator as for the C Code Generator Reference. The section on Names in Generated C Code is also valid for AgileC Code Generator.

## See also

"Restrictions in UML Support when Building C Applications" on page 951 in Chapter 27, *Building Applications Reference*

# 40

# *C Compiler Driver*

The C Compiler Driver is a utility intended to simplify C debugging of generated C code. It can be invoked from the makefiles used by the Application Builder or “stand-alone” from the OS command line.

# Application areas for CCD

The C Compiler Driver (CCD) is a utility that is intended to simplify C debugging, by generating an intermediate C file placed in a user defined directory. This C file has all its C macros expanded and is optionally "beautified" (pretty-printed). For ease of use, CCD is used as a C compiler driver, which is called from the makefiles generated by the C Code Generator. CCD may also be used as a command line option when compiling the C code with a user command, should that be required.

The CCD utility is not a feature that is generally supported for all compilers. It should be noted that a premade CCD Configuration File is not part of the standard kernel files for most of the premade kernels.

### How to take advantage of CCD

The simplest way to introduce this facility for use in the UML tool set is to modify the `makeoptions / make.opt` file contained in the run-time library to be used. To enable this feature in a user-defined run-time library, the directory and file structure must be similar to the pre-defined one.

The only modification that is **required** is to change the line

```
sctCC = cc
/* or some other compiler executable name */
```

to

```
sctCC = sccd cc
/* or some other compiler executable name */
```

It is necessary that the path variable is set to include the directory containing the Tau executable binaries, for example (Windows):

```
C:\Program Files\Telelogic\TAU_4.2\bin
```

### Customizing CCD

To customize the behavior of CCD, modify the variables in the configuration file `sccd_<your_compiler_type>.cfg` How to do this is described in the section "CCD behavior" on page 1323.

# CCD User Interface

The syntax for the CCD command line interface is:

```
sccd [command] [option]
```

### Compile a C file

To use CCD to compile a C file, the following command is used:

```
sccd <C compiler command line>
```

The C compiler command should be the command line used to compile a C file in the usual way: it should include the name of your C compiler, possibly compiler options, and finally the name of the C file.

### Print configuration

To print the values of the variables in the configuration file `sccd.cfg`, the following command without any options should be used:

```
sccd
```

To print the variables with help information:

```
sccd -h
```

### Return Codes

0: Success.

1: Return code after `sccd` prints “configuration and help”.

2: No `.c input` file given.

3: Could not open `InFile.c` input file

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 CCD

CCD performs the following sequence of actions:

1. Execute an optional 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 preprocessor pass to expand all C 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 CCD (see "C Beautifier" on page 1322 for more information).
10. Optionally compile, that is run the original command.
11. Optional clean-up of the sub-directory, but leave the pre-processed `.c` file(s) for debugging purposes.

## C Beautifier

If you need a C beautifier to further format the C code generated by the C Code Generator, you may want to try the `indent` utility (courtesy of Joseph Arcaneaux).

**The `indent` executable must be placed in your path.**

In order to easily invoke `indent` from CCD, 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 much like the ones used in "The C Programming Language" by Kernighan & Ritchie. It will also try to force lines to be shorter than 70 characters and will use 2 positions indentation in `if/while/..` statements.

**Example 401**

A slightly more elaborate example for how to use indent:

```
sccdUSER_CMD4 = "indent -kr -l70 -br -nce -nlp -ci3 -i2
%p/%d/%f.c"
```

# CCD Configuration File

### CCD behavior

The behavior of CCD 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 within the UML tool set framework, CCD uses `$sctdir/sccd.cfg` as the configuration file; otherwise CCD searches for `sccd.cfg` in the current directory, `$HOME` (**UNIX**) and `$SCCD`.

**Note**

*If `sccd.cfg` is not found, hard-coded defaults suitable for the GNU C compiler (gcc) are used.*

## CCD variables

Below are the variables that control the behavior of CCD. All characters in variable values are significant, including spaces.

### sccdNAME

```
sccdNAME = "Default"
```

Compiler name as defined in `scttypes.h`

**sccdINFILESUFFIX**

```
sccdINFILESUFFIX = ".c"
```

The expected file name suffix of the In-file(s), Default `".c"`.

**sccdCPP**

```
sccdCPP = ""
```

The name of the C pre-processor. If this is left empty, CPP is used. Default is `""`

**sccdCPPFLAGS**

```
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**

```
sccdMACROPREFIX = "-D"
```

CPP command-line define MACRO prefix. Default `"-D"`.

**sccdINCLUDE1**

```
sccdINCLUDE1 = "-I"
```

CPP command-line include-path prefix. Default `"-I"`

**sccdINCLUDE2**

```
sccdINCLUDE2 = ""
```

Alternative CPP command-line include-path prefix. Default `""`

**sccdOUTFILEREDIR**

```
sccdOUTFILEREDIR = "-o "
```

Character sequence to control CPP output file name. If empty, use `sccdFMOVE` instead.

**sccdFMOVE**

```
sccdFMOVE = ""
```

OS forced file move or copy command. Used instead of `sccdOUTFILEREDIR`. Default: `""`

**sccdDELETE**

```
sccdDELETE = "rm -f"
```

OS forced delete file command. Default: `"rm -f"`.

**sccdCOPY**

```
sccdCOPY = "cp"
```

OS normal copy command. Default: `"cp"`.

**sccdCOMPILE**

```
sccdCOMPILE = "ON"
```

Controls whether the final compilation pass should be run or not. Values are: `"OFF"` and default is `"ON"`.

**sccdDEBUG**

```
sccdDEBUG = "OFF"
```

Enable execution. Values are: `"ON"` and default is `"OFF"`.

**sccdPURGE**

```
sccdPURGE = "ON"
```

Purge temporary files. Values are: `"OFF"` and default is `"ON"`.

**sccdUSE_HS**

```
sccdUSE_HS = "OFF"
```

When set `"ON"`, the `.hs` files are not included until the compilation pass. Values are: `"ON"` and default is `"OFF"`.

**sccdSILENT**

```
sccdSILENT = "OFF"
```

Enable trace printout. Values are: `"ON"` and default is `"OFF"`.

**sccdTMPDIR**

```
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

```
sccdUSER_CMD1 = ""
sccdUSER_CMD2 = ""
sccdUSER_CMD3 = ""
sccdUSER_CMD4 = ""
```

User-defined commands ("Actions Performed by CCD" on page 1322). 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 402**

```
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 `\\`

# *UML and Java*

The chapters listed under **UML and Java** describe how the UML tool set can be used for engineering Java code.

The Java support and Eclipse integration in Tau are only supported for Windows operating systems.

# 42

# *Java Support*

This chapter describes how to use the Java support. It extends Tau with import of existing Java applications and Java roundtrip support, meaning code generation and reverse engineering. It includes features for compiling and executing Java code, generation of javadoc and JAR files among other things.

It also adds the possibility to use Java syntax in text diagrams and text symbols (in addition to UML), giving you a free choice of design language. You can switch back and forth between the two syntaxes at any time.

Existing Java files can easily be reused by importing them into a UML model, and it is also possible to generate Java from an existing UML model.

# Creating a Java Project

To create a Java project use the following procedure:

1. Choose the command File->New...
2. Choose **UML for Java Code generation** as the project type when creating the project. If you want to use the Eclipse IDE for working with generated Java code you should instead choose **UML for Java Eclipse Project**.
3. Choose the Java language dialect you want to use. This will have consequences for what default libraries that are used, and a few similar things.

For details on how to create a project, see "Working with Projects" on page 35 in Chapter 4, *Introduction to Tau 4.2*.

To activate the Java support for an existing project:

1. From the **Tools** menu select Customize.
2. Click the Add-Ins tab and check the **JavaApplication** add-in.
3. Click **Close**.

**Note**

*When creating or loading a project with Java support activated, the entire Java Runtime Libraries are loaded as a library in Tau. Since this library is very big (in particular for later versions of the Java language)* ***this may take several minutes****. During loading of the libraries Tau will not respond to any user interaction.*

Some features of the Java support uses the Java Software Development Kit which can be downloaded from the Sun web site.

```
http://java.sun.com/javase/
```

# The Java View

The Java support includes a Java centric view of a model called Java View. It provides a different way of viewing elements compared to the Standard View, by only showing those model elements which exist natively in the Java language. UML specific constructs, such as signals and statemachines, are not shown when using the Java View.

If your model does not make use of UML specific constructs that are translated to Java code elements you might find the Java View convenient to use.

**See also**

"Model View" on page 19 in Chapter 4, *Introduction to Tau 4.2*

## Activating the Java View

To activate the Java View:

1. In the **View** menu, select **Reconfigure Model View...**
2. Select **Java View** in the dialog and click **OK**.

You can switch the Model View between the Standard View and the Java View at any time.

## Working in the Java View

While in the Java View, you can only see and create Java related elements. Other elements are filtered out and need to be accessed in the Standard View.

The elements available in the Java view are the elements naturally available in the Java language:

- Java packages, Classes, Interfaces, Stereotypes (representing Annotations), Attributes and Operations.

and the diagrams needed to completely define these elements:

- Class diagram and Text diagram.

In addition, artifacts representing Java files and the Java build artifact are also available in this view:

- Java File, JAR File and Java Build Artifact.

Some operations are most conveniently performed in the Java View, for example creating a Java package, while some operations can equally well be performed in any view, for example importing Java source code. Some operations can only be performed in the Standard View. One example is exporting a top level package. This is quite natural, since the package is not visible in the Java view if it is not a Java package, and a consequence of the export is that it becomes a Java package.

The Java View is split into two different top levels: Java Model and Java Libraries. The Java Model node corresponds to the Model node of the Standard View, showing elements of the user model. The Java Libraries node contains predefined Java libraries, such as the predefined data types and java.lang.

### Differences between the Java View and the Standard View

The Java View is designed to naturally represent Java elements in a way close to the language and the file system, while the Standard View provides a UML-like way of showing elements. Sometimes these two views are fundamentally different. This is best explained by an example, the JAR File artifact.

In the model a JAR file is represented by the following elements:

- An artifact representing the physical JAR file
- The entities contained in the JAR
- Dependencies from the artifact to the entities of the file

The artifact and the entities contained in the JAR are located *at the same level* in the model. The dependencies are owned by the artifact. This is the UML way of specifying the contents of a file.

In the Java View the fact that the elements are contained in the JAR is emphasized. Therefore, the elements are displayed as if they were owned by the artifact. The entities in the JAR are *not* displayed at the same level as the artifact, but below the artifact instead. The dependencies are not displayed at all in the Java View. The differences can be seen in the table below:

| **Standard View** | **Java View** |
|---|---|
| Artifact<br>Dependency to Element 1<br>Dependency to Element 2<br>Element 1<br>Element 2 | JAR File artifact<br>Element 1<br>Element 2 |

Keep these differences in mind when switching between the Java View and the Standard View.

# Java Build Artifact

An artifact with the stereotype <<Java>> applied is referred to as a Java build artifact. It defines the scope and meaning of many commands performed when working with Java in Tau:

- It supplies commands in the 'Java' context menu which allows you to generate or compile Java code and to update the model with changes made in generated Java source files.
- It manifests one or many UML elements (using <<manifest>> dependencies). This defines which elements of the UML model that are affected by the commands that are performed on the Java build artifact. Note that the build artifact commands apply also on elements that are indirectly manifested by the build artifact (i.e. elements that are directly or indirectly owned by a manifested element).
- It stores options and settings in the form of tagged values for the <<Java>> stereotype. Typically these options affect the result of invoking the commands in the context menu. For example, the 'Target Directory' is stored on the Java build artifact, specifying the connection between the manifested elements in the UML model and their source code location in the file system.

See Build Artifact and Using Build Artifacts for more information about build artifacts in general.

## Java Build Artifact Commands

The following commands are available in the 'Java' submenu of the context menu on a Java build artifact.

### Generate

Use this command to generate Java source code for all elements that are manifested by the Java build artifact. Only those UML elements that have been modified since the time of the last generation will be translated. If the model is unchanged no Java source files will be generated, and a "File is up-to-date" message is printed in the Build tab for each Java source file.

The Java menu contains a command **Update source code** which performs the **Generate** command on all Java build artifacts that are present in the model.

### Generate (force)

This command works as Generate, but it will force all Java source files for the manifested elements to be generated, no matter if the corresponding UML elements actually have been modified or not.

The Java menu contains a command **Force update source code** which performs the **Generate (force)** command on all Java build artifacts that are present in the model.

### Update

Use this command to update the manifested elements with changes that have been made in the generated Java source files for these elements. This procedure is sometimes called “roundtrip engineering”. Only Java files that have been modified since they were generated will be considered. If no files have been modified nothing will happen, and a “File is up-to-date” message is printed in the Build tab for each Java source file.

The Java menu contains a command **Update model** which performs the **Update** command on all Java build artifacts that are present in the model.

### Update (force)

This command works as Update, but it will force an update from all Java source files corresponding to manifested elements, no matter if these files have actually be modified or not.

The Java menu contains a command **Force update model** which performs the **Update (force)** command on all Java build artifacts that are present in the model.

### Build

Use this command to compile Java source files corresponding to the manifested elements. The command will first perform the Generate command to make sure the Java source files are up-to-date with the model, before compiling them. See Compiling and Executing Java for more information about this command.

# Java Build Artifact Settings

A Java build artifact stores a set of settings in the form of tagged values for the <<Java>> stereotype. To view and edit these settings follow these steps:

1. Select the Java build artifact in the Model View
2. In its context menu select **Build Settings**.

This will open the Properties Editor, which is used for viewing and editing the Java build artifact settings.

Some of the settings are common for all build artifacts. These are described in the chapter Using Build Artifacts. Settings specific to Java are described below.

### Perform transformations

By default the Java code generator performs a number of transformations when translating from UML to Java. All supported UML constructs for which native Java constructs do not exist must be transformed to obtain a correct Java program. Technically what happens is that relevant parts of the user model are copied during code generation to a new temporary model in which transformations can take place without affecting the original design model.

If you only use UML constructs in your model that have a native mapping to Java (for example classes, interfaces, packages etc.) you can make code generation somewhat faster by turning this setting off. Note, however, that doing so disables all kinds of transformations, not only those performed by the Java code generator itself. It also disables any custom transformations that might have been added by add-ins in order to customize generated Java code.

### Classpath

Provides a way to manually override the Classpath variable used during compilation and execution.

### Automatic source generation

If this setting is turned on the Java source files will be automatically updated (regenerated) when the model is changed. The update will happen shortly after the editing action is finished. The default is that automatic source code generation is turned off.

### Automatic model update

If this setting is turned on the model will be automatically updated when the Java source files change, e.g. when saving them in an external text editor. The default is that automatic model update is turned off.

### Support roundtrip

If you don't intend to make changes to generated Java source files that shall be propagated back to the model using the Update command, you can turn this option off. Then generated Java files will always be generated from scratch.

Benefits with using this option are that code generation becomes somewhat faster, and that the generated file may get a more consistent indentation layout.

### Apply <<informal>> stereotype to imported methods

This option is by default on, and means that bodies of imported Java methods won't be checked for correctness at the UML level.

# Java Files

This section describes how to work with existing Java files by importing them into a UML model. It also describes how to generate Java files from an existing UML model.

## Importing Existing Java Applications

To import an existing Java application into Tau do as follows:

1. Select the command **File->Import**
2. Select **Import Java source code** in the Import dialog and click OK
3. In the tree view that appear browse to the directory you want to import. You can select several directories to be imported at once.
4. In the same page you can also select a few options:
   - The option **Create diagrams** will cause the importer to create a diagram for each imported package. The diagram will show all classes contained in the package and their relationships. Text diagrams will also be created for operation bodies to show their contents.

5. Click **Finish** to complete the import wizard. One Java package will be created in the UML model corresponding to each directory you have specified.

The UML model corresponding to each specified directory will be stored in a separate file.

All Java files in the selected folder are parsed and inserted into the model. All folders in the selected folder are inserted as top level Java packages. Files in the selected folder itself are treated as being defined in the unnamed package and inserted there.

The import is recursive, so any sub folders are inserted as sub packages in the model. For details on the mapping between Java files and the model, see "Model to File Mapping" on page 1355.

A Java Build Artifact will be added for each imported folder. The path of the imported folder is stored as the 'Target Directory' of the Java build artifact.

Once the Java files have been imported the model and the source code are kept synchronized by using the **Update model** and **Update source code** commands in the Java menu. It is also possible to use the **Update** and **Generate** commands in the 'Java' context menu on the Java build artifact.

To edit the source code, double-click the Java File artifact representing the file, or use the **Go to Source** command on a model or presentation element. This will open the `.java` file in the built-in text editor.

## Importing JAR Files

The contents of a JAR file can be imported into the model so the definitions of the JAR file can be referred in the model. Only the signatures of the definitions are imported, not the implementations.

To import an existing JAR file into Tau do as follows:

1. Select the command **File->Import**
2. Select **Import JAR file** in the Import dialog and click OK
3. Browse to the the JAR file you want to import and select it.
4. Click **Open** to complete the import wizard. One Java package will be created in the UML model corresponding to each JAR file you have specified. Class diagrams will be generated in these packages to show imported classes and their relationships.

The JAR file is parsed and inserted into the model. The structure of the resulting model follows the contents of the JAR file, typically one top-level package is created, but the JAR file can contain several top level packages.

Each package corresponding to a JAR file will be stored in a separate UML file, named after the JAR file.

In addition, a JAR File artifact is created to represent the `.jar` file. Dependencies from the artifact to the imported packages are created.

The connection between a package and its JAR file is maintained by storing the path to the JAR file in a stereotype on the artifact.

Packages created by importing a JAR file are treated as a static library, and therefore they have a number of limitations compared to other root packages:

- They are not automatically synchronized with the JAR file. If the JAR file changes it has to be reimported manually.
- It is not possible to use the Update model and Update source code commands on a JAR package.

To reimport a JAR file do as follows:

1. Select the JAR file artifact in the Model View.
2. Select **Reimport JAR file** in the context menu.

## Generating Java from Existing Models

Java source files can be generated from a UML model, no matter how the model was originally created. Java files can be generated in the following ways:

- Exporting a package to Java source code
- Generating JAR files
- Generating javadoc

### Exporting a package to Java source code

To export a package in the model into Java source code:

1. Activate the *Standard View* by selecting **Reconfigure Model View...** in the **View** menu and then selecting the **Standard View**.
2. Select a package in the model.

3. In the **Java** menu, select the **Export** sub menu and click **Package...**
4. Select the destination folder in the file system and click **OK**.

The entire contents of the package is generated into `.java` files and written to the file system according to the rules described in Model to File Mapping and Java Code Generator Reference. In addition one Java Build Artifact is added to the model for each exported top level Java package. It manifests the package and stores the folder path as target directory in the <<Java>> stereotype.

Once the Java files have been exported the model and the source code are kept synchronized by using the **Update model** and **Update source code** commands in the Java menu. It is also possible to use the **Update** and **Generate** commands in the 'Java' context menu on the Java build artifact.

To edit the source code, double-click the Java File artifact representing the file, or use the **Go to source** command on a model or presentation element. This will open the `.java` file in the built-in text editor.

The export command is available for the following packages:

1. Top level UML packages
2. Java packages

Case 1 is used when generating Java from a package for the first time. Case 2 is used to move the generated Java files to a new location in the file system. As an alternative to using the export command for case 2 you can simply change the target directory value of the Java build artifact and select **Force Generate** in its context menu.

### Generating JAR files

To generate a JAR file from a package in the model:

1. Select a package in the model.
2. In the **Java** menu, select the **Generate** sub menu and click **JAR file...**
3. Select the destination folder in the file system and click **OK**.

This will generate a JAR file in the destination folder with the same name as the package. The JAR file is generated from *compiled classes and interfaces only*, that is `.class` files in the folder connected to the package. Before gen-

erating a JAR file make sure to execute the **Generate** command followed by the **Compile** command on the Java Build Artifact. Make sure the compilation does not report any errors.

### Generating javadoc

To generate `javadoc` from a package in the model:

1. Select a package in the model.
2. In the **Java** menu, select the **Generate** sub menu and click **Javadoc...**
3. Select the destination folder in the file system and click **OK**.

This will generate javadoc files for the selected package in the destination folder. The javadoc files are placed in a sub folder with the same name as the package appended with *_doc*. The index page is then displayed in the built-in web browser.

Javadoc is generated from *Java source files only*, that is `.java` files in the folder connected to the package. Before generating javadoc, make sure to execute the **Generate** command on the Java Build Artifact. Also note that the javadoc tool will not work on Java files that contain errors. You may use the **Compile** command on the Java build artifact to make sure the generated java code is correct before generating javadoc.

See Comments for more information about how to create Javadoc comments in the UML model.

# Java Syntax

The default syntax in the UML tool set is referred to as U2, which is a textual UML syntax. The Java support extends the tool with Java syntax. This means that models can now be edited either in U2 or Java, it is up to you which syntax you would like to use. You can create new models in either U2 or Java and you can look at existing models in either U2 or Java. The languages are very similar, Java is basically a subset of U2, although there are some differences at the semantic level. Note that some constructs of U2 are not translated to Java; for details see Java Code Generator Reference.

Java syntax is available in:

- Text diagrams
- Text symbols (in class and package diagrams)

Note that Java syntax is not used in state machine diagrams. The reason for that is that some U2 constructs, not present in the Java language, are commonly needed in state machine diagrams. Examples include constructs for sending signals and working with timers.

Java syntax can only be set at package level and is normally set on top level packages. When Java syntax is enabled for a package all text diagrams and text symbols contained in the package are using Java syntax. Java syntax is also hierarchically applied to all elements (including packages) owned by the package.

Java syntax has precedence over U2 syntax, and the higher level in the hierarchy has higher priority than lower levels. This means that if Java syntax is enabled on a top level package everything contained in that package will use Java syntax.

# Synchronizing Model and Source Code

The model and source are kept fully synchronized by updating the source code from the model or the model from the source code. Since the UML tool set is a true model-based tool it is chosen to treat the model and the source code as different things, the model being the most important one. The result is a slightly different behavior depending in which direction an update is made: model to code or code to model. It also depends on if automatic or manual update is used. The details are described in the following sections.

## Automatic vs Manual Synchronization

The update of the model and source code can be done in two different fashions, automatically when the source code / model is changed or manually using special update commands.

There are two settings that control the synchronization mode:

- Automatic roundtrip
- Automatic code generation

The details of these settings are described in the section Java Build Artifact Settings. The default is that the automatic roundtrip and synchronization are turned off.

## Manually Updating Java Source Code

Java source code can in manual synchronization mode be updated (i.e. generated) from a model in two ways:

- For all Java build artifacts in the active project
- For one particular Java build artifact only

To generate Java source code for all Java build artifacts in the active project:

1. In the **Java** menu, select **Update source code** or use the keyboard shortcut **CTRL + ALT + S**

To generate Java source code for one particular Java build artifact only:

1. Select the Java build artifact in the Model View or in a diagram.
2. Right-click and select **Generate** in the 'Java' submenu.

When using the **Update source code** or **Generate** commands to generate Java source code only the Java files that are affected by changes made in the model will be generated. So, in most cases only a subset of the Java source code files will be regenerated. To force an update of all Java source code files choose the command **Force update source code** in the Java menu, or **Force Generate** in the Java build artifact context menu.

Java source code can be generated for a number of different UML model elements, but not for all. See Java Code Generator Reference for more details on how UML elements are represented in Java. See also Model to File Mapping for information about how UML elements are mapped to generated files and folders in the file system.

The model and the file system is automatically kept synchronized, but ***the model has higher priority than the file system when generating source code***. If there are any model elements that lack a representation in the file system, the corresponding file system elements are created automatically. For example if a new class has been added to the model since the last time source code was generated, the corresponding `.java` file is automatically created.

In addition, if there are any file system elements that lack a representation in the model, for example a folder that has no corresponding package in the model, this is detected and the file system elements can be deleted. Automatic deletion of an element always prompts for user confirmation to avoid unintended loss of information.

### Renaming and moving model elements

The connection between the model and the files and folders in the file system is maintained automatically when importing source code or exporting packages from a model as described in "Model to File Mapping" on page 1355. ***When entities are renamed or moved in the model the connection is not automatically maintained, however.*** Therefore, great care has to be taken when moving model elements representing files or folders to preserve the connection between them. This applies to packages, Java file artifacts and JAR file artifacts.

There is one exception to this rule: Top level packages that are renamed in the model. When updating the source code after renaming a top level package, the file system directory is renamed, and all sub packages and Java file artifacts in the model are updated to reflect the new location.

The correct way to correctly rename or move an element while preserving the file connection is outlined below:

1. Rename the model element, and if applicable the Java file artifact manifesting the element.
2. Update the path of the element to reflect the new name and location. For Java file artifacts this is done by editing the Path tagged value, while for top level packages it is done by editing the 'Target Directory' tagged value in the Java build artifact that manifests the package.
3. Move and/or rename the corresponding folders and files in the file system.

If any of these steps is performed incorrectly problems can appear during synchronization. For example, if the files in the file system are not moved, new files will be generated in the new location. These new files will not include non-Javadoc comments from the old files since they are not stored in the model (see Comments for more information).

### Changing the location of generated Java files

To move all generated Java files of a Java package, re-export the package to the desired location, see Exporting a package to Java source code for details.

Existing `.java` files are copied to the new location, and the paths of all Java file artifacts, as well as the 'Target Directory' of the Java build artifact, are automatically updated.

Changing the location of a single Java file is not recommended since it can violate the package versus file system correspondence enforced by the Java language. It can however be done by following the procedure described in Renaming and moving model elements above.

### Changes to existing files

Existing files are only updated when needed, so files are only updated when changes have been made to the corresponding model element(s). The model changes are merged into the file preserving contents and formatting of the file to the greatest possible extent.

To force source files to be regenerated from scratch you can turn off the Support roundtrip setting. This makes code generation somewhat faster, and may result in a more consistent indentation layout in the file, but should only be used if you do not intend to modify generated Java source files manually.

## Manually Updating the Model from Java Source Code

The model can in manual synchronization mode be updated from source code in two ways:

- For all Java build artifacts in the active project
- For one particular Java build artifact only

To update all Java build artifacts in the active project from source code:

1. In the **Java** menu, select the **Update model** or use the keyboard shortcut **CTRL + ALT + M**

To update only one particular Java build artifact from source code:

1. Select the Java build artifact in the Model View or in a diagram.
2. Right-click and select **Update** in the 'Java' submenu.

Note that it is not the Java build artifact itself that will get changed when updating the model from the source code, but rather the UML elements that are manifested by it.

When updating the model from source code, the file system is scanned and the model is updated from the file system elements using the mapping rules described in Model to File Mapping below.

When using the **Update model** or **Update** commands to update the model based on Java source code only the Java files that have been changed since the last update of the model will be considered during the update. So, in most cases only a subset of the Java source code files will be used and only a subset of the model will be updated. To force an update based on all Java source code files whether they are modified or not choose the command **Force update model**, or **Force Update** in the Java build artifact context menu.

The file system and the model are automatically kept synchronized, but ***the file system has higher priority than the model when updating source code from the file system***. If there are any file system elements that lack a representation in the model, the corresponding model elements are created automatically. For example if a new `.java` file has been added to the file system since the last update was made, the corresponding UML elements are automatically created in the model.

In addition, if there are any model elements that lack a representation in the file system, for example a package that has no corresponding folder in the file system, this is detected and the package is deleted from the model. Automatic deletion of an element always requires user confirmation to avoid unintended loss of information.

### Synchronized Target Directory

The target directory of a Java build artifact is synchronized when the **Update** or **Update (force)** command is performed on the build artifact. Synchronization means that the contents of the folder and the model is kept identical.

When a new sub folder of a synchronized folder has been created in the file system, it is detected and it will be reversed and inserted into the model. Correspondingly, when a package in the model mapped to a subvocally of a synchronized folder is deleted you will be asked to delete the folder in the file system.

# Navigating to and from Generated Java Files

Once Java code has been generated Tau keeps track of the source code location for elements in the model. This makes it possible to navigate from the model to the code, and also from the code to the model.

To open a generated Java file you may double-click on the file artifact that represents it in the model. You may also use the **Go to Source** command that is available in the context menu of any generated definition, to navigate directly to the location (or locations - in some cases one UML definition can end up in multiple Java files) of that definition in the generated Java files.

When navigating to a generated Java file it will be opened in Tau's built-in text editor.

It is also possible to navigate in the opposite direction, from the Java code that was generated, to the corresponding UML model entity. This is done by the **Go to Source** command that is available in the context menu of Tau's built-in text editor.

# Compiling and Executing Java

The Java support of Tau contains an Eclipse Integration which facilitates compiling and executing generated Java code using the Eclipse IDE. However, it is also possible to compile and execute Java code from Tau directly.

### Compiling

To compile generated source code (`.java` files) into `.class` files:

1. Select a Java build artifact.
2. In the context menu select **Compile** in the 'Java' submenu.

This will perform the Generate command on the Java build artifact to make sure all Java files are up-to-date. Then the generated source files will be compiled into `.class` files.

You can also use the **Compile** command in the Java menu in order to compile all generated Java files for all Java build artifacts in the active project.

If the compilation fails, for example due to a syntax error, the compilation errors are written to the **Build** output tab. You can navigate to the error by double-clicking the line containing file name followed by the line number. The file is then opened and the cursor is positioned on the line causing the error.

Errors can be resolved in the model or in the source code, but it is important to remember to synchronize the model and the source code when fixing an error.

The classpath used during compilation is calculated according to the description in "Classpath variable" on page 1347. Compilation is always performed with the *source* compiler option explicitly set to the choosen language version (to enable asserts among other things).

### Executing a class

To execute a class as a Java application:

1. Select the class you would like to execute.
2. In the **Java** menu, select **Execute**.

This will execute the class as a Java application. The output of the execution will be displayed in the Script window. Executing a class requires that it has been successfully compiled and contains a main method. If the `.java` file exists but the `.class` file does not, the `.java` file is compiled automatically.

The classpath used during execution is calculated according to the description in the Classpath variable section. Execution is always performed with the **esa** option to enable asserts during execution. If you want to set other options to the Java VM, select the **Execute with options...** in the Java menu.

### Executing a class as an applet

This command is the same as Executing a class but the class is wrapped into an applet which is then opened in a web browser. This command only works for classes in unnamed packages. It is intended to be used as a quick check of simple classes.

1. Select the class you would like to execute as an applet.
2. In the **Java** menu, select **Execute applet**.

This will create a file on the form `<class name>.html` instantiating the class as an applet.The `html` file will then be displayed in a web browser.

### Classpath variable

The classpath variable used when compiling and executing Java code is calculated automatically, and no changes should normally be required.

The directories of all Java packages and jar-files in the project are added to the classpath. In addition, all Synchronized Target Directory are added. Also, if the model uses constructs which imply a dependency to the Java Run-time Framework, a class path entry for that library (`tor.jar`) will also be added.

It is possible to override the classpath if the scheme above is not sufficient, for example if you want to include external libraries not represented in the model. To manually override the classpath:

1. Select the **Java Model** node (**Model** node in Standard View) in the Model View.
2. Right-click and select **Properties...**
3. In the **Filter** drop-down, select **Java Settings.**
4. Add entries to the **Classpath** list as desired.

**Note**

*Manually overriding the classpath disables the automatic calculation of the classpath. Only the entries manually added to the list will be used. It is important to remember to update the classpath when new top level packages are created.*

# Execution Tracing

The execution of Java programs can be traced in Tau using sequence diagrams. This feature can be used for visualizing communication between Java objects, to detect incorrect program flows, and in general to obtain an understanding of the run-time behavior of a Java program. Tracing is often combined with debugging in an IDE, such as Eclipse. By setting breakpoints around interesting sections of code a trace can be obtained for visualizing what the program does in those parts of the code.

In order to produce an execution trace the Java program must be **instrumented**. This is done by loading a trace agent into the Java virtual machine. The agent collects information about what happens in the program by responding to events sent by the virtual machine. This information is then sent to Tau, where it is presented in UML diagrams.

The Tau module which produces the diagram from the trace events is called a **tracer**. It is possible to implement custom tracers in order to present the trace information in some other way, for example in another kind of diagram, or to filter the information in a different way.

Tau ships with a standard tracer, the Instance Tracer, for producing sequence diagrams where each Java class instance (object) is represented by its own lifeline, and where the interaction between instances in the form of method calls are visualized.

**Hint**

*Since Java trace instrumentation is done without modifying the Java source code it is possible to trace the execution of any Java program, not only programs generated from Tau.*

## Start a New Trace Session

Perform the following steps in Tau to start a new trace session:

1. Select the command **Java / New Trace Session...**
2. In the dialog select which tracer to use for the trace session.
3. The selected tracer will by default be enabled initially, meaning that as soon as the dialog is closed it is ready to receive trace events. If you want to wait a little with enabling the tracer (for example to give time for running the Java program up to a breakpoint of interest first) then uncheck the **Enabled** checkbox.
4. Press **OK** to close the dialog.

When the dialog is closed a new top-level "trace" package will be created in the model. It is by default called "JavaTrace". You may change this name to better describe the trace session.

The settings made in the dialog are stored on the trace package. You can change these settings at any time by opening the Properties Editor on the trace package with the filter `javaTrace` selected. The most common reason for doing this is to enable or disable the trace session in order to filter the trace to only cover interesting parts of the program execution.

## Instrumenting the Java Program

To make the Java program instrumented you need to add an option to the Java virtual machine:

```
-agentlib:JavaInstrument
```

This will tell the Java virtual machine to load the instrumentation agent `JavaInstrument` before running the Java program. Note that you also must have your environment variable `PATH` (`LD_LIBRARY_PATH` on Unix) set to include the Tau installation `bin` directory so that the Java virtual machine can find the `JavaInstrument` agent.

The JavaInstrument agent takes several options which may be specified after an equal sign. Options are separated by commas, and each option consists of a name-value pair separated by a colon. For example:

```
-agentlib:JavaInstrument=host:localhost,port:57000
```

Available options are described below.

### Option 'host'

This option specifies the name of the host computer where the Tau instance to trace to is running. The default value is "localhost" meaning that Tau should be running on the same machine as the Java program. You may use this option to send the trace to a Tau running on a different machine in the network.

### Option 'port'

This option specifies the web server port of the Tau instance to trace to. The default port number is 57000, but if you have (or have had) multiple instances of Tau running on the machine, you may need to use a different port number. To find out which port number a particular instance of Tau is using follow the instructions in How to Use the Tau Web Server.

### Option 'start_method'

By default instrumentation events start to be sent when the Java application reaches the main method. This option can be used for setting a different method to start the instrumentation in. Currently only the method name can be specified here; it is not possible to qualify the name or to specify a particular overload of a method.

The 'start_method' option is useful for applications which consists of a complex start-up phase, involving lots of calls to standard libraries. One example is applications with a user-interface. Typically it is uninteresting (and takes

too long time) to trace what happens during this initialization phase. Tracing can then begin at the first method that gets called after the initialization phase is completed.

**Example 403: Using the 'start_method' option to defer instrumentation start**

Consider the following Java program:

```
class MyClass {
  public static void main( String[] args) {
    MyDialog dlg = new MyDialog();
    start();
  }
}
```

Assuming that the `MyDialog` class is a Swing dialog, its instantiation will imply lots of calls to the Swing library. To avoid tracing these calls we use the 'start_method' option:

```
-agentlib:JavaInstrument=start_method:start
```

Tracing will now begin when the 'start' method is called, that is after the GUI has been initialized.

---

## Option 'skip'

This option specifies definitions that should not be instrumented. The value is a list of definitions using the same syntax as the Java import statement.

The default value of this option is "java.*;sun.*", meaning that definitions in the 'java' and 'sun' packages will be excluded from tracing.

The 'skip' option allows you to reduce the generated trace to only involve those parts of the program you are interested in. For example, you may want to skip instrumenting all 3rd party libraries you are using.

Note that a call from method A to method B is only skipped from instrumentation if both A and B are part of a definition listed in the 'skip' list. If only one of these methods is in the 'skip' list the call will be traced. This can for example be used to see which library calls your code performs, or which 3rd party components that call your code.

**Example 404: Using the 'skip' option to filter a trace**

The following JavaInstrument option:

```
-agentlib:JavaInstrument=skip:javax.*;MyPkg.MyClass
```

will exclude all definitions in the package `javax` and also the class `MyPkg.MyClass` from tracing.

---

**Setting instrumentation options when using Eclipse**

Here are the steps to perform in the Eclipse IDE in order to make the instrumentation settings described above. The instructions are for Eclipse 3.3; other versions may have a slightly different user interface.

1. Right-click the Eclipse Java project you want to instrument. Select **Debug As / Open Debug Dialog...**
2. Make sure the Run configuration for the project is selected, or create a new Run configuration for the project.
3. In the **Arguments** tab enter the instrumentation option in the **VM arguments** field. For example:
   `-agentlib:JavaInstrument=host:localhost,port:57000`
4. In the **Environment** tab create a new variable `PATH` (`LD_LIBRARY_PATH` on Unix) and set its value to the Tau installation `bin` directory. For example, `C:\Program Files\Telelogic\TAU_4.2\bin`
5. Click **Debug** to close the dialog and start the debug session.

Of course you can do the same for a Run configuration in case you do not want to debug the Java program.

# Instance Tracer

The Instance Tracer is a predefined tracer which visualizes application trace events as a UML sequence diagram. The diagram will contain one lifeline for each dynamic Java class instance that performs some actions during the trace session. The name of the lifeline is the address of the Java object.

For classes that contain static methods an additional lifeline will be created, called `static`. It represents all static parts of the class.

To improve trace performance certain low-level Java library objects are not traced. This includes objects for classes defined in the `java` and `sun` packages.

Called methods, including constructors, initializers etc. are visualized in the diagram as method calls. The creation of new Java objects is also visualized.

### Example

Consider the Java program below:

```
package JavaTraceTest;
class C {
    public static void main( String[] args) {
        D d = new D();
        d.foo();
    }
}
class D {
    void foo() {
        return;
    }
}
```

The sequence diagram trace produced by the Instance Tracer for this program looks like shown in Figure 246 on page 1354. The diagram shows the following events in the Java program:

1. The static `main` method in class `C` is called. The object making this call is a low-level library object which is not traced to the diagram.
2. A new instance of class `D` is created. The instance is located at address `0x1507fb2` in the Java virtual machine. The creation also implies the call of `D`'s constructor. The implementation of this constructor is auto-generated by the Java compiler.
3. Control is returned to `C.main` from `D`'s constructor.
4. A call to `D.foo` is made.
5. Control is returned to `C.main` from `D.foo`.
6. The `main` method is returned from.

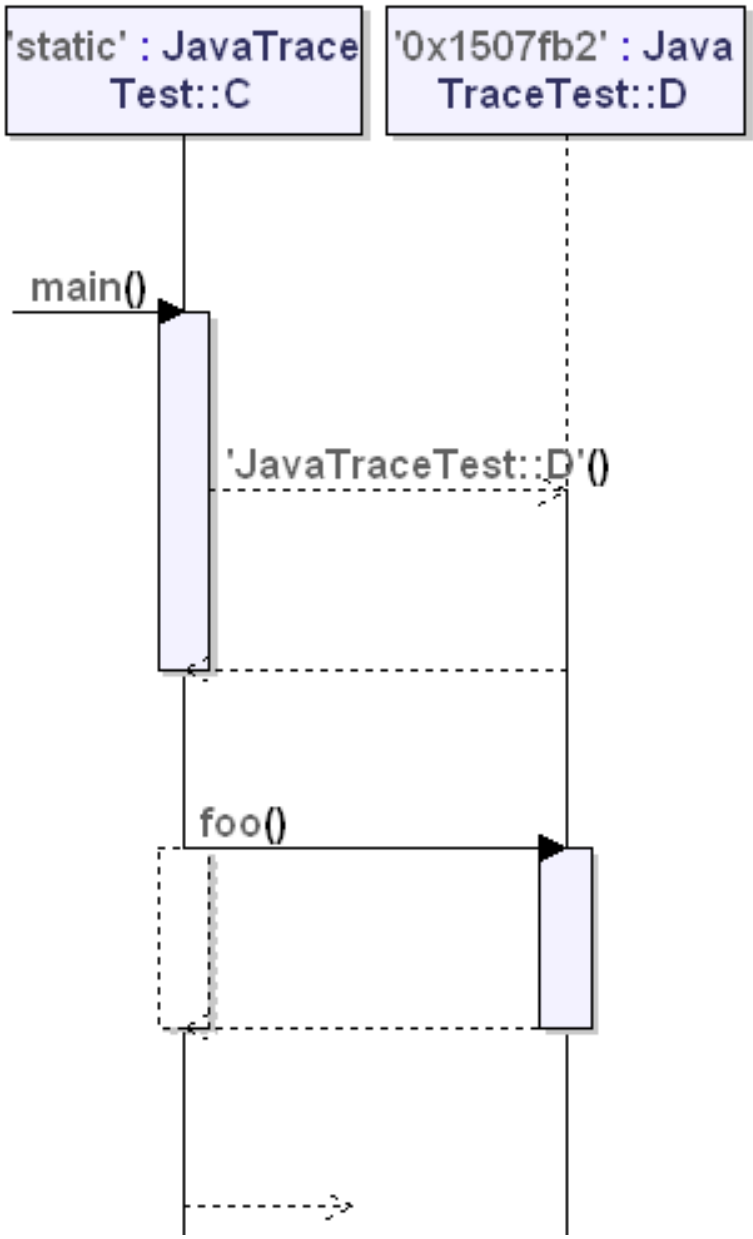



*Figure 246: Sequence diagram trace produced by the Instance Tracer*

## Adding Custom Tracers

It is possible to add custom tracers in order to present the trace events received from the Java virtual machine in a custom way. This is done by defining a class that inherits from a `JavaTracer` class, and defining agent operations in the class that overload appropriate virtual operations from `JavaTracer`. The agents get called when trace events are received from the

Java virtual machine. For more information about `JavaTracer` and its available operations see the package `TTDJavaModelCodeSync::Tracing` under the `Libraries` section of the model. There you can also find the definition of the `InstanceTracer`.

# Model to File Mapping

This section describes the mapping rules between model and file system elements. The table below is a brief summary.

| File system element | Model element |
|---|---|
| `.java` file | Java file artifact with manifest relations to the elements it represents, for example classes and interfaces. |
| `.jar` file | JAR file artifact with dependencies to the elements of the JAR. |
| folder | Java package |

### Java file artifacts

A Java file artifact is used to represent each `.java` file, or more specifically each Java compilation unit. A Java file artifact is an artifact with the `«javaFile»` stereotype applied. The artifact has the same name as the `.java` file it represents.

The path to the file is stored in the “path” property of the javaFile stereotype (shown as “File name” in the Properties Editor). This path can be relative to the project or absolute. It is automatically set during import or export (using a relative path if possible). The mapping can be changed manually if desired, by editing the value in the Properties Editor.

Manifest dependencies from the artifact to model elements are used to specify what elements that should be written to the `.java` file when updating the source code. The manifested elements are written into the file as top level declarations. Manifest relations can be created, moved or deleted to control the contents of the file.

Import declarations in Java are defined per file, or compilation unit. They are stored in the artifact as import dependencies. Import dependencies can be created, moved or deleted to control which import declarations that are

written to the file. Note, however, that the Java code generator can automatically compute which import declarations that are needed in a generated file based on which definitions that are referenced in the file. Therefore you only need to manually add import dependencies to a Java file artifact if you want to add additional import declarations (for example needed by inline target code that are informally represented in the model, and thus cannot be analyzed by the Java code generator).

In order for other elements in the UML package owning the artifact to see these import dependencies, the artifact is imported by the package that owns it. This makes all definitions imported by the artifact visible to the entire package.

Artifacts are automatically created and deleted when updating the source code or the model.

**Note**

*It is not recommended to create, move or delete these artifacts manually.*

### Java packages

A Java package in the model is mapped to a folder in the file system. Nested packages are mapped to nested folders in the file system.

A Java package is a package with the `«javaPackage»` stereotype applied. The mapping between a Java package and its folder in the file system is controlled by the target directory setting of the Java build artifact that manifests the package. This can be either an absolute or project relative path. If a Java package is not directly manifested by a Java build artifact, the folder path is computed by looking at the folder for the owning package.

When importing existing Java files, the mapping is set up automatically and it should normally not require any changes. The mapping can however be changed at any time.

### Unnamed package

An unnamed package is created automatically when needed. Typically when importing a `.java` file located directly in the folder selected during import.

The unnamed package is called ‘default’ and has the stereotype **unnamedPackage** applied.

No package declaration is generated when synchronizing elements in the unnamed package.

There can only be one unnamed package per project.

**Note**

*Different Java environments and compilers treat unnamed packages in different ways and this may cause problems when compiling and executing. Therefore usage of* ***unnamed packages should be avoided if possible****.*

### JAR file artifacts

JAR files are represented with a JAR file artifact that has the same name as the `.jar` file. The artifact has dependencies to the elements of the JAR file.

JAR file artifacts are created automatically when importing a JAR file. Although they can be created manually for design purposes, they can not be used to describe how to generate a JAR file.

Packages located in JAR files are Java packages with an additional stereotype, called `«jarPackage»` to indicate that they are located in a JAR.

JAR packages are considered to be static and are limited compared to other packages, see Importing JAR Files for details. For details on how to generate JAR files from the model see Generating JAR files.

# Java Runtime Libraries

The most common standard Java runtime libraries are included in the Java support. The contents of these libraries is found in the **Java Libraries** node **(Libraries** when using the **Standard View)** in the Model View. The runtime libraries include the entire contents of the rt.jar file of the Java distribution.

### Loading of runtime libraries

By default, the runtime libraries are automatically loaded when loading a Java project. Since not all Java programs makes use of these run-time libraries, and because loading of these large libraries can take some time, it is possible to disable the automatic loading of them.

To avoid loading the rt.jar library during startup, change the Java Settings for the project by deselecting the **Load rt.jar on startup** check box.

### Using the runtime libraries

Any of the elements in the runtime libraries can be used as ordinary UML elements with full binding and name completion, etc.

To use them in diagrams, just drag the element from the **Model View** to a diagram. To use them as types in symbols or in the textual syntax use the element names. The elements must be made visible in the scope in order to bind correctly, see Visibility of packages and elements below for details.

### Visibility of packages and elements

As required by any Java implementation, the contents of the package `java.lang` is always accessible. Any other package has to be explicitly imported or referred by a fully qualified name in order to bind. An example is given below in U2 syntax (entire packages can not be edited in Java syntax).

```
package P1 <<import>> dependency to java::io {
  class A {
    Integer i;
    FileWriter fw;
    java::util::Vector v;
  }
}
```

The type `Integer` of attribute `i` in the example above automatically binds to `java::lang::Integer`. The type `FileWrite` of `fw` attribute binds to `java::io::FileWrite` because package `P1` imports the package `java::io`. The fully qualified name has to be used for `Vector` of attribute `v` since that package is not visible in this scope.

**Note**

*Since Java file artifacts are used to represent `.java` files, only imports made by an artifact is written to the file it represents. Imports on UML packages or classes are not automatically transferred to all `.java` files generated from the package.*

### Tau Object Runtime library

In order to support translation of UML constructs that have no native representation in the Java language, such as statemachines, signals etc., a Java library called Tau Object Runtime (TOR) is used. To a large extent this library is only used at Java source code level by code generated by the Java code

generator, but there are some parts of the library which are useful to access also on the UML model level. Therefore TOR is loaded as a library (called ‘tor’).

For more information about the TOR library see Java Run-time Framework.

#### Additional libraries

If there is a need for other Java libraries, such as the encryption support in `jsse.jar` this JAR file can be imported manually, see Importing JAR Files.

# Java Modeling Utilities

Tau provides a few utilities which facilitate the modeling of Java programs in UML. Some of these utilities apply automatically when using the ordinary commands in Tau, while others manifest themselves as dedicated commands.

## Active Class Generalization

When a Java class is made active, Tau automatically adds a generalization for it, so that it will inherit the TOR class DispatchableClass. This is done in order to provide UML level access to important methods of that class, such as ‘init’ and ‘start’.

If you make a Java class passive, the generalization to DispatchableClass will be automatically removed.

## Generate Main Method

It is possible to automatically generate a main method for a Java class. To do this follow these steps:

1. Select the class in the Model View.
2. In the context menu perform the command **Utilities - Generate Main Method.**

The main method is generated from information found in the Java Build Artifact that is used for building the selected class. For each non-nested active class that is directly or indirectly manifested by that build artifact, an instance will be created. The instances will be added to a Dispatcher, followed by calls to ‘init’ and ‘start’ on the instances. Finally ‘run’ is called on the dispatcher.

The generated main method will thus execute the top-level active class instances as a single-threaded application. You can modify this default implementation in any way you like, for example using a ThreadedDispatcher to create a multi-threaded program.

# Java Settings

Each Java project (or model) has a set of Java specific settings. Some of these settings are specific to a particular Java build artifact, and can have different values for different Java build artifacts. These settings are described in Java Build Artifact Settings.

In this section we will describe the other kind of settings, those that are common for the entire model. These global settings are stored in the project file. To view or edit these settings do the following:

1. Select the **Java Model** node (**Model** node in Standard View) in the Model View.
2. Right-click and select **Properties...**
3. In the **Filter** drop-down, select **Java Settings.**

There are the following different settings:

- Language version
  - This option controls which Java language version to use. Normally this option is set when creating the Java project using the New Wizard and is not changed after that. You may however need to change this option when upgrading your project to use a newer Java version. In that case you should reload the Java project afterwards to make sure the correct version of the Java libraries will be loaded.
- Load rt.jar on startup
  - A flag specifying if the contents of the Java Runtime Libraries (rt.jar) should be loaded on startup or not. Disabling loading of rt.jar significantly improves the loading time, but references to elements in rt.jar will not be bound.

# Known Restrictions

This section lists all known limitations and restrictions relevant for the Java support.

### Active code generators

In order to generate code efficiently, it is recommended that each project limits the number of active code generators to one. For example if both the C++ Application Generator and the Java code generator are active in a project, the code generation may require longer time to complete and/or require larger memory resources.

### Model binding in inner classes

Sometimes when using Java syntax, statements defining inner classes will not bind properly. The following example illustrates this. The parameter passed to the call of `op1` in the example below will not bind.

```
abstract class classA {
  abstract void run();
}
class classB {
  public void op1(classA p1) {}
  public void op2() {
    op1( new classA() {
        void run() {
        }
      }
    );
  }
}
```

### No Java syntax for UML packages

It is not possible to edit an entire UML package in the Java textual syntax. Although the package concepts of UML and Java are superficially very similar they are different semantically. There is not a one-to-one mapping between packages in UML and in Java. UML packages can only be edited in U2 textual syntax.

### Switching between Java and U2 syntax fails

Sometimes it is not possible to switch back and forth between Java and U2 syntax. This happens when there is a syntax error in one of the syntaxes due to an unsupported or unmapped concept. If there is a syntax error it will not be possible to change the syntax to the other language.

If this happens, either fix the error or delete the text symbol or diagram and create a new one using the correct syntax to start with.

# Using the Java EE 5 Addin

The purpose of the Java EE 5 addin is to simplify the design of Java Enterprise Edition 5 applications using Telelogic Tau. The JEE5 addin provides the following features:

- A Java EE 5 UML profile that gives access to Java EE 5 concepts in the UML model.
- Utilities to facilitate Java EE 5 model design directly from context menus in UML diagrams.

The Java EE 5 addin documentation is structured into two parts:

- A simple introduction that step-by-step will take you through the design of a simple Java EE 5 enterprise component in Tau.
- A reference manual chapter that describes the commands and utilities available in the addin (see Java EE 5 Addin Reference Manual).

## The Hello Example

As a simple example let's consider creating a small Java EE model that will provide a greetings service to the world. We will start from scratch and end up with a complete Java EE component ready to be deployed on your application server of choice.

## Creating a Java EE 5 Project

The next step is to create a new java project. This can be done using the New wizard in Tau:

1. Select the **File->New** command and choose **UML for Java Code Generation** as the project type and 5 as the java version.
2. Give it a suitable name, in the rest of this document we will refer to it as "hello".
3. Select the **Location** to a suitable directory.

*The name of the directory must not contain spaces to make sure all utilities described below will work. In the rest of this introduction we will assume the location is C:\TauProjects\hello.*

4. Accept the default options in the rest of the New wizard

   You have now created a standard Java 5 EE project.

## Activating the JavaEE5 addin

To use the Enterprise Edition concepts it is necessary to switch on the JavaEE5 addin:

1. Choose the command **Tools->Customize**.
2. Open the **Add-ins** tab.
3. Activate the JEE5 addin and click on the **Close** button.

You are now ready to create Java EE 5 models.

## Creating an EJB Component Session Bean

There are several component kinds in the Java EE 5 framework; stateless and stateful session beans, message driven beans and persistent entities. In this example we will start by creating a stateless session bean. To do this we essentially only need to do three things:

- define the business interface of the bean
- implement the business interface in a bean class
- package the bean in a JAR file

We will start by designing the component interface with its business methods and mark this as a remotely accessible bean interface by stereotyping it with the <<Remote>> stereotype:

- Start by creating a package to hold the EJB components called “ejb” inside the “hello” package.
- Double-click on the “ejb” package in the Model View and create a class diagram.
- Create a new interface in the diagram and call it “Hi”.
- Add an operation “greetings():String” to the interface
- Select the interface symbol, and choose “Apply <<Remote>>” from the context menu.

We now have completed the component interface definition and the next step is to create a stateless session bean that implements the interface:

- To get a kick-start in the implementation use the command “Create implementation class” available in the context menu for the interface. This will create a class that contains the correct operation and that realizes the interface.

- To mark this as a stateless session bean, choose the command “Apply <<Stateless>>” from the context menu.

The next step is to provide the implementation of the greetings() operation. This can be done either in the UML model or by coding directly in the java file. In this example we’ll choose to edit the code directly. This is done as follows:

- To get java source code for the UML model select the “hello” package and choose the **Java->Export package** command.
- To edit the source code for the bean class, select the command **Go to Source** in the context menu of the “HiBean” class. This will open the java source code file in the built-in text editor.
- Fill in the implementation of the greetings method: public String greetings(){ return "Hello!"; }
- Save the file to update the UML model. Use for example the Ctrl-S keyboard shortcut in the source code editor.

The design and implementation of the EJB component is now finished and what remains is to build and package it. In general this is easiest to do using a build tool like Ant or using a Java IDE like Eclipse, but in this simple example we will use the standard java tools javac and jar, that are available in the Java EE 5 SDK. This SDK can be downloaded e.g. from http://java.sun.com/javaee/downloads/index.jsp. Tau contains some simple commands in the Java menu that wraps javac and jar and makes them available from inside the tool. To use them there is however a need to set up the classpath for the java compiler to include both the all jar files used in the model and the directories where the source code is generated. The classpath is stored as a property of the root of the UML model:

- Select the **Java Model** node in the Model view and choose the **Properties…** command from the context menu.
- In properties dialog choose **Java Settings** as the “Filter” to see the java related settings.
- Add a value to the Classpath property that includes the javax.jar file. This should be something like “;/Sun/SDK/lib/javaee.jar” Note that the citation markers should not be part of the value. Also note the initial semicolon.

If you use an external Java IDE to compile the program you will instead need to add the javaee.jar file to the compilation settings of the IDE. The exact steps to do this depend on the details of the IDE, but if you are using Eclipse this is done in the Libraries tab of the properties for the java project.

We're now ready to start compiling the two java files using the javac command:

- Select the "hello" package in the Model View.
- Choose the command **Java->Compile**. This will compile all files contained in this package.

The only remaining task is now to package the component into a jar file that can be deployed on an application server:

- Select the hello package and give the command **Java->Generate->Jar file…**
- When prompted: Select a suitable target directory.
- You can follow the progress of the jar command in the Script window. In our case this should show that the Hi.class and HiBean.class files are added to the jar file.

You now have a packaged component hello.jar that can be deployed on an application server. How to do this is specific for each application server but usually the application servers contain both a web based interface and a command line interface that can be used to deploy components. If you happen to be using the free Glassfish server the command line command to deploy our component would be "asadm deploy hello.jar".

## Creating Persistent Entities

The next step is to create a few persistent entities, i.e. classes that in the end will be stored on a database on the application server.

We will create two classes that will give some personal flavor to the hello service we're implementing by storing favorite greeting phrases for each person. The UML design consists of two classes, Person and Phrase as in the following figure:

After creating this small model (create them in the same ejb package as where you created the Hi interface and the HiBean class) we will mark them as persistent entities. This is done as follows for each class:

- Select the class.
- Choose the command **Apply <<Entity>>** in the context menu. This will in addition to applying the correct stereotype also create a constructor that is mandatory for Java EE entity classes.
- Choose the command **Create Get/Set operations**. This will generate Java Bean compatible getters and setters that furthermore will be stereotyped according to the multiplicities and navigability of the attributes in the class.

The next step is to give each class a primary key. This is an attribute of the class that can be used to uniquely identify instances of the class.

- Select the **name** attribute in the Person class and choose the command **Select as primary key**. This will apply the stereotype <<Id> to the corresponding Get operation.
- Do the same for the "text" attribute of the Phrase class.

We now have two persistent entities in the model and the next step is to use these entities from the "hello" session bean. The idea is to add business methods to the bean to accomplish the following:

- It should be possible to add a new person with a favorite greeting phrase
- A new greetings method should be available that uses the personalized phrase.

To accomplish this do as follows:

- Add two operations to the Hi interface: greetings(name:String):String and addPerson(name:String,phrase:String)
- Select the interface and choose the command **Push operations to class** in the context menu.
- Open the source code editor for the HiBean class (for example by selecting the class symbol and choosing the command **Go to Source** in the context menu).
- Modify the source code according to the following example. The italic text is the text that you have to insert. It contains three parts, a reference to an EntityManager and the implementations of the two new operations:

```
package hello.ejb;
@javax.ejb.Stateless public class HiBean implements Hi {
@javax.persistence.PersistenceContext
javax.persistence.EntityManager em;
public String greetings(){ return "Hello!"; }
```

```
public String greetings( String name) {
Person p = em.find(Person.class,name);
if (p.getFavorite() != null) {
return p.getFavorite().getText();
} else {
return "Hello!";
}
}
public void addPerson( String name, String phrase) {
Person p = new Person();
p.setName(name);
Phrase ph = new Phrase();
ph.setText(phrase);
p.setFavorite(ph);
em.persist(p);
em.persist(ph);
}
}
```

Now we have completed the ejb part of the application and what remains is to build and package it. Do this in the same way as was described in section "Creating an EJB Component – Session Bean" above.

When deploying the new component to the application server you need to make sure that the application server contains a data base server and that the mapping from the Person and Phrase entities to tables in a data base is defined. By default there is assumed to be one table per entity class, with the same name as the class, and one column per attribute, also with the same name as the attributes. Most application servers contain a utility to create the tables automatically based on the entity classes. In many cases this is done by giving a deployment descriptor in the jar file that you package the EJB in. The deployment descriptor relevant for entities is a file called persistence.xml that is located in a directory called META-INF in the root of the jar file. Please consult your application server manuals for details.

Below is an example of what the persistence.jar file could look like if you are using the Glassfish application server. Notice the non-standard property "toplink.ddl-generation" that tells the application server to delete and recreate all necessary tables in the database associated with the "__default" jdbc data source.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence
xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persisten
ce
http://java.sun.com/xml/ns/persistence/persistence_1_0.x
sd" version="1.0">
```

```
<persistence-unit name="hello" transaction-type="JTA">
<jta-data-source>jdbc/__default</jta-data-source>
<jar-file>hello.jar</jar-file>
<properties>
<property name="toplink.ddl-generation"
value="drop-and-create-tables"/>
</properties>
</persistence-unit>
</persistence>
```

# Java EE 5 Addin Reference Manual

## Commands Available for Interfaces

The following commands are available in the context menu for interfaces, both when selecting an interface symbol in a diagram and when selecting an interface in the Model View.

| Command | Description |
|---|---|
| Apply <<Remote>> | The interface will be marked as a remotely accessible bean interface by adding the stereotype javax.ejb.Remote. |
| Apply <<Local>> | The interface will be marked as a remotely accessible bean interface by adding the stereotype javax.ejb.Local. |
| Create Implementation Class | A class will be generated that implements the interface. The class will realize the interface and will contain all the operations that were defined in the interface. |

## Commands Available for Classes

The following commands are available in the context menu for classes, both when selecting a class symbol in a diagram and when selecting a class in the Model View.

| Command | Description |
|---|---|
| Apply <<Stateless>> | The class will be marked as a stateless session bean by adding the stereotype javax.ejb.Stateless. |
| Apply <<Stateful>> | The class will be marked as a stateful session bean by adding the stereotype javax.ejb.Stateful. |
| Apply <<Entity>> | The class will be marked as a persistent entity by adding the stereotype javax.persistence.Entity. The class will also automatically get a constructor generated to comply with the rules for persistent entities in Java EE 5. |
| Apply <<MessageDriven>> | The class will be marked as a message-driven bean by adding the stereotype javax.ejb.MessageDriven. |

| | |
|---|---|
| Create Interface Class | An interface will be generated that is realized by the class. The interface will contain all public operations defined in the class. |
| Create Get/Set Operations | Get/Set operations will be generated for all attributes in the class in a style compatible with the requirements for Java Beans.<br>If both the class that owns the attributes, and the type of the attributes, are persistent entities (i.e. marked by the <<Entity>> stereotype) there will also be generated annotations on the Get operations (one of OneToMany OneToOne ManyToOne ManyToMany depending on the multiplicity and navigability). |
| Copy Operations from Interface | Operations in realized interfaces will be inserted in the class (if not already existing). |

## Commands Available for Attributes

The following commands are available in the context menu for attributes, both when selecting an attribute in a class symbol in a diagram and when selecting an attribute in the Model View.

| **Command** | **Description** |
|---|---|
| Select as primary key | The attribute is marked as a primary key of the class. This is accomplished by stereotyping the corresponding get operation by javax::persistence::id. |
| Create Get/Set Operations | Get/Set operations will be generated for the attribute in the class in a style compatible with the requirements for Java Beans. |

## Persistent Entity Utilities

The Java EE support for persistent entities (in addition to what has been described above) consists mainly in generating the Java EE stereotypes for relationships based on the multiplicities given in the UML model. Essentially this gives us a possibility to design the relationships using UML class diagrams and automatically generate the correct EJB annotations in the Java source code.

The generation of the multiplicity annotations is handled by the “Create Get/Set operations” command described above.

## Stereotypes relevant for Java EE applications

There are many stereotypes loaded by the Java EE addin as part of the UML model of the javaee.jar java library. You can check the stereotypes by examining the javax package in the Library folder of the Model View. Most of the interesting stereotypes are found in the javax.ejb package. This includes the <<Stateless>> and <<Stateful>> stereotypes indicating different kinds of session beans and the <<Remote>> and <<Local>> stereotypes used to define different kinds of bean interfaces.

It might also be worthwhile to investigate the javax.persistence package that contains, among others, the stereotype <<Entity>>, used to mark a class as a persistent entity to be stored in a data base on the server. Some other useful stereotypes in this package include:

- <<Table>>: Used to specify the mapping between an entity and a database table.

- <<Column>>: Used to specify the name of a database table column for an attribute in an entity.

# 43

# *Java Code Generator Reference*

This chapter is a reference guide to the Java code generator. It describes how UML language constructs are translated to Java language constructs. It also describes how it is possible to customize the Java code that gets generated.

# General

The Java code generator implements a mapping of UML language constructs to Java language constructs. Two different categories of UML language constructs can be identified in this process:

1. UML constructs which has a native representation in Java, for example a class or an interface.
2. UML constructs for which no native Java construct exists, for example a statemachine or a signal.

UML elements of the first category are translated by Java code generator by simply printing the Java Syntax for the UML element to the generated `.java` file. This can thus be done directly without the need to first transform the UML element in any way.

UML elements of the second category can, however, not be translated in such a simple manner. Instead the Java code generator needs to transform the UML element into one or many different UML elements belonging to category 1 (and for which Java code thus can be directly generated). For example, a statemachine is transformed into a Java class that inherits from a certain library class, and has certain attributes and methods defined.

In order to perform such transformations transparently without modifying the original design model, the Java code generator copies relevant parts of the original model and performs required transformations on that copy. When code generation is completed the copy is thrown away.

Most of this document describes the translation rules from UML to Java which are implemented by these built-in transformations in the Java code generator. At the end of the chapter we also describe how you can implement your own custom transformations in addition to the built-in ones. This customization mechanism enables you both to customize the default translation of supported UML constructs, as well as to implement a Java translation for any UML element not already supported by the Java code generator. For example, it is possible to implement custom code generation for stereotyped UML elements, thus giving a code-level meaning to such stereotypes.

**See also**

Java Run-time Framework which describes the Java run-time library that is frequently used by generated Java code.

## Java to UML Translation

As described in Synchronizing Model and Source Code the Java support in Tau also supports translating Java source code to UML elements. This is what happens when you import existing Java code into Tau, or when updating the model with changes made in generated Java source files (also known as round-trip engineering).

The translation rules from Java to UML are in general the exact opposite of the translation rules from UML to Java. Note, however, that "code patterns" corresponding to transformed UML elements, for example a statemachine, will not be recognized in the Java to UML translation. This is obviously not an issue when importing legacy Java source code, since such code won't contain transformed UML constructs. However, when using round-trip engineering to update the model with changes made in generated files you have to keep this in mind. In general round-trip is only supported for Java constructs that can be directly mapped to a corresponding UML construct.

## Document Structure

The following chapters describe the subset of UML that can be translated to Java by the Java code generator. UML language constructs not mentioned here are not supported, and will be ignored during translation.

For each supported UML construct a translation rule is given. If there are exceptions to the rule, these are also mentioned.

For most translation rules examples are given using textual UML and Java syntax.

**Note**

*The purpose of each example is only to illustrate the translation rule at hand, not to give a precise description of how the generated code will look like. Also note that examples for brevity reasons typically are fragmental and may omit important things such as error handling etc. Thus, be careful if you decide to copy/paste such example code for use in your model.*

# General Translation Rules

This chapter describes general translation rules that apply to many kinds of translated entities.

# Name of Definitions

**The name of a Java definition is the same as the name of the UML definition from which it is translated.**

Note that no name mangling takes place in case the UML name is not a legal Java identifier (for example if it contains spaces, or is a Java reserved word). You must make sure that the names you use in UML are valid also in Java.

Also note that the <<ansiName>> stereotype which is supported by some Tau code generators is not supported by the Java code generator.

# Type of Typed Definitions

**The type reference of a typed definition (that is a definition that has a type, for example an attribute or a parameter) is translated to a corresponding type reference in Java.**

### Impact of aggregation kind

The Java code generator does not consider which aggregation kind that is specified for the typed definition. No matter if the definition has reference, shared or part aggregation, the generated Java definition will be a reference.

However, in the case of a UML attribute of part type (the attribute has “composition” as aggregation kind), and where the attribute multiplicity specifies an initial number of instances, code will be generated that makes sure that when the containing class is instantiated, the specified number of initial instance will also be created for the attribute.

**Example 405: Translation of type reference and aggregation kind ————————**

**UML**

```
class AC {}

class Class1 {
  AC 'ref';
  shared AC sh;
  part AC prt;
}
```

**Java**

```
class AC {}
```

```
class Class1 {
    AC ref;
    AC sh;
    AC prt = new AC();
}
```

A more complete description of the translation of part attributes can be found in Class Composite Structure Initialization.

### Predefined types

**A reference to a predefined type is translated to a corresponding reference in Java, using the same name for the referenced type. This is the case both for references to Java and UML predefined types.**

UML representations of Java built-in types, such as boolean and byte, are present in a library called TTDJavaPredefined. Usually these are the predefined types you use in a model intended for Java code generation.

If you decide to use the UML predefined types instead, for example Charstring, you must provide a Java definition of that type yourself to be used by generated code.

**Example 406: Translation of references to predefined types**

**UML**

```
class Class1 {
  boolean b;
  Charstring str;
}
```

**Java**

```
class Class1 {
    boolean b;
    Charstring str;
}
```

Note that unless you provide a Java implementation of the Charstring type you will get a compilation error on the definition of ‘str’.

## Collections and Multiplicity

Attributes (or associations) with Multiplicity > 1 (i.e. collections) can not be generated into Java directly since the concept is not supported by the Java language.

The mapping of this kind of attributes depends on if formal or informal multiplicity is used.

### Informal multiplicity

**If informal multiplicity is used (the property InformalMultiplicity is true for the attribute) then the multiplicity is ignored and the type of the attribute is translated literally to Java.**

So, if the type given is List<B> then this is what is mapped to Java independent of the multiplicity given for the attribute. This is useful when there is a need to specify a collection type for the attribute that is different from the default container type, but the multiplicity is needed from an analysis point of view.

### Formal multiplicity

**If formal multiplicity is used the type of attributes with multiplicity > 1 will be an implicit container type determined by the presence of an instance of the `<<containerType>>` stereotype.**

See The <<containerType>> stereotype for more information about how to control which default container type to use.

When a new Java project is created `<<containerType>>` is by default applied on the Model level. Which type it specifies depends on the Java dialect used:

- For Java 5 or later the default collection type is java.util.Vector<Any>, where Any refers to the type of the attribute.
- For Java 1.4 and earlier the default collection type is java.util.Vector.

**Example 407: Default mapping of formal non-single multiplicity**

**UML**

```
myAttribute : myType [*];
```

**Java 5 and later**

```
Vector<myType> myAttribute;
```

**Java 1.4 and earlier**

```
Vector myAttribute;
```

---

Whenever the multiplicity is set to a value that requires a Vector, an import dependency to `java.util.Vector` is automatically added to the model. This ensures that the correct import statements are written to the source code.

If the element owning the attribute is manifested by a Java File artifact, the import will be added to that artifact, otherwise the import is added to the element itself.

**Note**

*The import of* `java.util.Vector` *is only added when setting or changing the multiplicity to a value that requires it. If you generate Java code from an old model the imports have to be added manually. Note also that if you change the default container type you may have to add an import dependency to the specified container type yourself to get the correct import statements in generated code.*

## Visibility of Definitions

**The visibility of a UML definition is translated to the same visibility for the corresponding Java definition.**

**Note**

*The default visibility for definitions is different in Java and UML. It is therefore recommended to always be explicit about the visibility in UML, and not leave it unspecified. See Visibility for more information about UML visibility.*

An exception to the above rule applies for member definitions of active classes and state machines. Regardless of the UML visibility of such members, the corresponding Java members will have public visibility. This is necessary since they may be accessed by state and statemachine classes corresponding to composite states and inline state machines in the UML model. See Example 459 on page 1429 for an example.

## Qualified Names

In UML a definition can be referenced either using a fully qualified name, or using a name with a relative qualifier.

Java does not support relative scope qualifiers. A name in Java must either be fully qualified or have no qualifier at all. In the latter case an import statement for the used definition is required.

**A fully qualified UML name is translated to a corresponding fully qualified name in Java.**

By fully qualified in UML we include both qualifiers starting with the global scope qualifier (::), as well as qualifiers starting with a reference to a definition located in the global scope.

**A UML name with a relative qualifier is translated to a simple name in Java without qualifier. The required import statement (type-import-on-demand) is generated at the top of the file, if such an import statement doesn't already exist there.**

Note that the alternative approach to always generate fully qualified names in Java typically leads to poor code readability, and therefore is not used.

**Example 408: Translation of qualified names**

**UML**

```
// Relative qualifier to
com.telelogic.tau.tor.DispatchableClass
class C : tor::DispatchableClass {
  // Relative qualifier to java.util.concurrent.TimeUnit
  public TimeUnit tu;

  // Full qualifier to java.lang.Integer
  private ::java::lang::Integer i;
}
```

**Java**

```
import com.telelogic.tau.tor.*;
import java.util.concurrent.*;

class C extends DispatchableClass {
  public TimeUnit tu;
  private java.lang.Integer i;
}
```

## Comments

**Comments attached to UML definitions will be translated to Javadoc-style comments (/** ... */).**

Note that Javadoc only allows comments for certain specific kinds of definitions, for example classes and interfaces. If a comment is attached to another kind of definition in UML, it will not be translated to Java.

It is of course possible to add ordinary Java comments anywhere in the Java source files. But remember that only valid Javadoc comments will be synchronized to the UML model. Others will only be present in the source files.

**Example 409: Translation of comments**

**UML**

```
class C comment "A JavaDoc comment" {}
```

**Java**

```
/**A JavaDoc comment*/
class C {
}
```

## Non-Name Based References

**If a reference is not setup by name (but GUID or pointer) it is converted to a name-based reference. A minimal relative qualifier is added to the reference to make sure it binds to the same target as before.**

This translation rule is currently applied to a limited set of references (those references that are typically edited using graphical syntax in the editors).

# Package

**A non-empty UML package is translated to a Java package.**

If the UML package is not already stereotyped by the <<javaPackage>> stereotype that stereotype will be applied to make it visible in the JavaView.

Note that an empty UML package is not translated to Java. However, a folder for the package will be created in the file system, but it will be empty as long as the UML package is empty.

**Example 410: Translation of packages**

**UML**

```
package NJP {
  class Class1 {}
}

package Empty {}
```

**Java**

```
package NJP;

class Class1 {

}
```

# Dependency

Dependencies are used in many ways in a UML model and are often to be interpreted as informal. However, there are certain specific usages of dependencies that are visible in generated Java code. These usages of dependencies are described in this chapter. Dependencies that do not fall into the categories mentioned below are not translated to Java.

## Import and Access Dependencies

All import and access dependencies in the UML model are not automatically mapped to import declarations in Java.

Only import and access dependencies from an artifact representing a `.java` file are generated as import declarations in the corresponding file. Import dependencies between other elements, for example from a package to another package are not generated into any `.java` file.

**Access dependencies in UML are mapped to single-type-import declarations in Java. Import dependencies in UML are mapped to type-import-on-demand statements in Java.**

**Example 411: Translation of access and import dependencies**

**UML**

```
<<access>> dependency to java::util::Vector

<<import>> dependency to java::util
```

**Java**

```
import java.util.Vector;

import java.util.*;
```

**Note**

*Due to these mapping rules, to get syntactically correct Java, import dependencies must be used when importing packages and access dependencies must be used when importing individual model elements.*

Also note that it is the dependencies from the artifacts that are generated into java code. In most cases in UML import and access dependencies are used either between packages or classes. To simplify the handling of dependencies during Java code generation the dependencies are *copied* from the package owning the artifact and from the class(es) manifested by the artifact into the artifact when updating the source code based on the model. So when generating Java code these dependencies will be generated into the source code. Note however that if import statements are deleted in the Java code the corresponding dependencies between classes/packages in the UML model will *not* be deleted. They have to be deleted manually in the UML model or the import statements will be regenerated the next time the source code is updated from the model.

Note that in many cases the Java code generator is able to automatically generate necessary import statements. For example, import statements necessary in order to correctly handle relative UML qualifiers (see Qualified Names) will be automatically generated.

# Class

**A UML class is translated to a Java class.**

If the class is marked as abstract, the Java class will be abstract. If the class is marked as finalized, the Java class will be declared to be final.

**Example 412: Translation of classes**

**UML**

```
public abstract class A {}

public finalized class B {}

public class C {}
```

**Java**

```
public abstract class A {}

public final class B {}

public class C {}
```

## Nested Class

**A nested UML class (i.e. a class defined as a member of another class) is translated to a static nested Java class.**

The semantics of a nested class in Java is not the same as a nested class in UML. In Java there are two kinds of nested classes; inner classes and static nested classes. A static nested class in Java has similar semantics as a nested class in UML.

**A nested UML class stereotyped by <<innerClass>> is translated to an inner Java class.**

**Example 413: Translation of nested classes**

**UML**

```
class Outer
{
    class Nested1 {}
```

```
    class <<innerClass>> Nested2 {}
}
```

**Java**

```
class Outer
{
    static class Nested1 {}

    class Nested2 {}
}
```

---

## Active Class

An active class which contains a constructor statemachine (a.k.a. a classifier behavior statemachine) is translated to a Java class which extends the TOR class DispatchableClass.

The generated class contains the following members:

An attribute “m_sm” typed by the statemachine class (which is the classifier behavior).

- A redefinition of the method “init”, with an implementation that creates an instance of the statemachine class and stores it in the "m_sm" attribute. It also calls the inherited implementation. If the UML class already has an 'init' operation another one will not be generated.
- A redefinition of the method “start”, with an implementation calls the inherited implementation in order to start the state machine of the DispatchableClass class. The implementation also starts each active instance that is a part of the owning DispatchableClass instance. This means that when an instance of an active class is started, all contained instances will also be recursively started. If the UML class already has a “start” operation another one will not be generated.
- A redefinition of the method “receive”, with an implementation that just calls the inherited method.
- A redefinition of the method “getClassifierBehavior”, with an implementation that returns the “m_sm” attribute.

**Example 414: Translation of an active class with a classifier behavior statemachine**

**UML**

```
active class C
{
    statemachine initialize {}
}
```

**Java**

```
public class C extends DispatchableClass
{
    public void init()
    {
        m_sm = new C_initialize(this);
        super.init();
    }

    public void start()
    {
        super.start();
    }

    public boolean receive(Event e)
    {
        return super.receive(e);
    }

    public StateMachine getClassifierBehavior()
    {
        return m_sm;
    }

    C_initialize m_sm;
}
```

# Interface

**A UML interface is translated to a Java interface.**

**Example 415: Translation of interfaces**

**UML**

```
public interface Ifc {}
```

**Java**

```
public interface Ifc {};
```

## Interfaces with Signals

If the UML interface contains one or many signals the Java interface will inherit from the EventReceiver interface of TOR. This expresses the (rather obvious) requirement that a class that implements such an interface must be able to receive events.

**Example 416: Translation of interfaces containing signals**

**UML**

```
public interface Ifc {
  public signal sig;
}
```

**Java**

```
public interface Ifc extends EventReceiver {
  public static class sig2 extends Event {
    public sig2() {}
    public static boolean isTypeOf(Event e) {
      return e instanceof sig2;
    }
  }
};
```

Note that a class that realizes an interface with signals in UML does not necessarily have to contain a state machine. It is fully possible to use a class without a state machine, and just implement the `EventReceiver::receive` operation to define what should happen when the signal is received. Note, however, that the EventReceiver interface in Java also contains a few other methods which therefore must be provided an implementation. Usually the easiest way to implement these methods is to make the class active in UML, so that the corresponding Java class will inherit DispatchableClass which contains appropriate default implementations of these methods.

# Stereotype

**A UML stereotype which has the <<Metadata>> stereotype applied is translated to a Java annotation.**

Java 5 and later supports annotations to be attached to certain kinds of definitions. In UML annotations are represented by means of stereotypes stereotyped by a <<Metadata>> stereotype. Other UML stereotypes which are not <<Metadata>> stereotypes are not translated to Java.

**Example 417: Translation of stereotypes**

**UML**

```
<<Metadata>> stereotype Stereo extends Definition [0 ..
1] {}

<<Stereo>> class MyClass {}
```

**Java**

```
@interface Stereo {}

@Stereo class MyClass {}
```

Annotation stereotypes that are inserted into the model by importing existing Java source code or JAR files will extend TTDMetamodel::Definition. This makes it possible to apply these stereotypes on all kinds of definitions.

Annotation stereotypes can also be created manually from the Java class diagram palette, or using the New menu in Model View. In this case you must add the extension to TTDMetamodel::Definition manually if you need to apply the stereotype to a definition in the model. To do this you have to work in Standard View as Extension is a UML concept not visible in the Java View. You can also add the extension easily by generating Java code, and then updating the model again from the generated file that contains the Java annotation.

## Stereotype Attributes

**Attributes in a stereotype are translated to annotation elements in Java if they are stereotyped by the <<AnnotationElement>> stereotype.**

Default values of stereotype attributes are translated to corresponding default values for the annotation elements.

**Example 418: Translation of stereotype attributes**

**UML**

```
<<Metadata>> stereotype S extends Definition [0 .. 1] {

  <<AnnotationElement>> int id;
  <<AnnotationElement>> String desc = "[N/A]";
}

<<Stereo(.id = 14, desc = "foo".)>> class MyClass {}
```

**Java**

```
@interface S {
  int id();
  String desc() default "[N/A]";
}

@Stereo(id = 14, synopsis = "foo") class MyClass {

}
```

# Attribute

**A non-constant UML attribute defined in a class is translated to a Java attribute in the corresponding Java class.**

**A constant UML attribute defined in a class or an interface is translated to a final Java attribute in the corresponding Java class.**

The mapping of modifiers and default value is straight forward as shown in the example below:

**Example 419: Translation of attributes**

**UML**

```
public class C {
```

```
  public int x = 14;
  public const int y = 15;
}
```

**Java**

```
public class C {
  public int x = 14;
  public final int y = 15;
}
```

---

An attribute defined in a stereotype is translated according to the rules described in Stereotype Attributes.

## Static Attribute

**A static UML attribute is translated to a static Java attribute. If the UML attribute is a static constant, the Java attribute will be a "static final" attribute.**

**Example 420: Translation of static attributes**

**UML**

```
public class C {
  public static int x = 14;
  public static const int y = 15;
}
```

**Java**

```
public class C {
  public static int x = 14;
  public static final int y = 15;
}
```

---

# Operation

**A UML operation in a class or an interface is translated to a Java method in the corresponding Java class or interface.**

**A static UML operation in a class is translated to a static Java method in the corresponding Java class.**

See Example 421 on page 1391 for an example.

# Operation Body

**An operation body in UML is translated to a Java method body.**

Note that in UML an operation body can either be inline defined (owned by the operation), or it can be stand-alone (referencing the operation). In Java all method bodies are inline defined.

**Example 421: Translation of operation bodies**

**UML**

```
class C {
  void foo(){ // Inline operation body
    return;
  }
  void bar();
  static void z(); // No defined body
}
void C::bar(){ // Stand-alone operation body
  return;
}
```

**Java**

```
class C {
  void foo(){
    return;
  }

  void bar(){
    return;
  }

  static void z() {}
}
```

Note that in case the UML operation has no operation body defined, a default empty body will be generated for the corresponding Java method.

# Operation with Statemachine Implementation

**An operation implemented by means of a statemachine implementation is translated to a Java method.**

It is required that such a statemachine implementation only contains a start transition and possibly local attribute definitions. It must not contain any states. The actions of the start transition will be translated to corresponding statements in the Java method.

**Example 422: Translation of operations with statemachine implementations ——**

**UML**

```
class C {
  public int m_op( int p1) statemachine {
    int i = 0;
    start {
        {
            {
                i = p1;
            }
        }
        return p1;
    }
  }
}
```

**Java**

```
class C {
  public int m_op( int p1) {
    int i = 0;
      {
        i = p1;
      }
      return p1;
  }
}
```

## Operation Parameters

**A parameter to a UML operation is translated to a formal parameter of the Java method that is the translation of the operation.**

**The type of the first parameter with direction kind 'return' (there should at most be one) is translated to a return type for the method that is the translation of the operation. If there is no return parameter, a void method is generated.**

Other UML direction kinds (such as `inout` or `out`) are not translated to Java.

**Example 423: Translation of operation parameters**

**UML**

```
int foo(int p1, in int p2, inout int p3, out int p4);
void bar();
```

**Java**

```
int foo(int p1, int p2, int p3, int p4);
void bar();
```

The general rules for typed entities (compare with Type of Typed Definitions) apply also on the return parameter.

---

Note that parameter default values are not supported when generating Java, and will be ignored.

### Parameter multiplicity

The impact of a multiplicity specified for a parameter is in general the same as for any definition (see Collections and Multiplicity). However, parameter multiplicity is also used to specify that a parameter is optional (0 is then included in the specified multiplicity ranges).

## Constructor

**A UML constructor is translated to a Java constructor.**

Note that a UML class operation called ‘initialize’ is a constructor.

### Constructor initializer

**A UML constructor initializer is translated to Java attribute assignments or calls of `super` in the Java constructor implementation.**

Note that there are two alternative ways to initialize a base class in UML - by using the 'base' contextual keyword, or by referring to the name of the base class.

The order of the generated actions is:

1. Any call to `super()`. There may at most be one.
2. Assignments of attributes in the order in which these attributes are defined in the class.

**Example 424: Translation of constructor initializers**

**UML**

```
class D
{
    public D(long) {}
}

class C : D
{
    public C(boolean b) : m_c(true), m_b(b), base(5) {}

    private boolean m_b;
    private boolean m_c;
}
```

**Java**

```
class D
{
  public D(long) {}
}

class C extends D
{
  public C(boolean b)
  {
    super(5);
    m_b = b;
    m_c = true;
  }

  private boolean m_b;
  private boolean m_c;
```

```
}
```

## Destructor

**A UML destructor is translated to an overload of the `finalize` method.**

Note that a UML class operation called ‘finalize’ is a destructor.

**Example 425: Translation of destructors**

**UML**

```
class D
{
    public ~D(){}
}
```

**Java**

```
class D
{
  @Override
  public finalize() {}
}
```

## Abstract Operation

**An abstract UML operation is translated to an abstract Java method.**

The Java class that contains the abstract method will also be marked as abstract.

**Example 426: Translation of abstract operations**

**UML**

```
class S {
    abstract int f1();
}
class D : S {
    redefined int f1(); // Redefines S::f1
}
```

**Java**

```
abstract class S {
  abstract int f1();
```

```
};
class D extends S {
  @Override
  int f1() {}
};
```

---

# Virtual, Redefined or Finalized Operation

**A virtual UML operation is translated to an ordinary Java method.**

This is because in Java all non-static operations are implicitly virtual.

**A redefined UML operation is translated to a Java method annotated by @Override.**

**A finalized UML operation is translated to a final Java method.**

**Example 427: Translation of virtual, redefined and finalized operations ————**

**UML**

```
class S {
    int f1();
    virtual int f2();
    virtual int f3();
    virtual int f4();
}
class D : S {
    int f1(); // Hides S::f1
    virtual int f2(); // Hides S::f2
    redefined int f3(); // Redefines S::f3
    finalized int f4(); // Redefines S::f4
}
```

**Java**

```
class S {
  int f1();
  int f2();
  int f3();
  int f4();
};
class D extends S {
  int f1();
  int f2();
  @Override
  int f3();
  @Override
  final int f4();
```

```
};
```

Be careful to notice the difference in semantics between Java and UML. In the above example `D::f1()` is hiding the implementation of `S::f1()`. In Java `D.f1()` instead overloads `S.f1()`. To detect this subtle problem it is recommended, to compile the generated Java code with appropriate options, if available, so that missing use of the `@Override` annotation are reported.

## Exception Specification

**An exception specification (a 'throw' declaration) for a UML operation, is translated into an exception specification for the Java method that is the translation of the operation.**

Note that while exception specifications are optional in UML, they are mandatory in Java if the method calls other methods which may throw exceptions.

**Example 428: Translation of exception specifications**

**UML**

```
void foo() throw Exc1, Exc2;
```

**Java**

```
void foo() throws Exc1, Exc2;
```

## Synchronized Operations

**A UML operation stereotyped by <<synchronized>> is translated to a synchronized Java method.**

**Example 429: Translation of synchronized operations**

**UML**

```
class S {
  <<synchronized>> void foo(){}
}
```

**Java**

```
class S {
  synchronized void foo() {}
};
```

## Main Operation

To obtain a ‘main’ method in the generated Java code, which is required to make the generated application executable by itself, a corresponding UML operation may be created in one of the classes. The UML operation should have the following signature for the generated Java method to become correct:

```
public static void main( Array<String> args);
```

To facilitate the procedure of generating a ‘main’ operation, Tau supports a command for generating such an operation:

1. Select a class in the Model View.
2. In the context menu select the command **Utilities - Generate Main Method**.

The implementation of the default ‘main’ method is generated according to the following rule:

**One instance is created for each active class that is manifested by the Build Artifact. Each instance is added to one single Dispatcher, and is then initialized and started. Finally the ‘run’ method is called on the Dispatcher.**

This means that the default ‘main’ method makes a fully synchronous (single-threaded) application. You can of course modify this default implementation in any way to become appropriate for your program.

# Generalization

**A generalization between two UML classes or interfaces is translated to an inheritance (`extends`) between the corresponding Java classes or interfaces.**

Generalizations between operations are not translated.

**Example 430: Translation of generalization**

**UML**

```
class S {
}
class D : S {
}
```

**Java**

```
class S {
};
class D extends S {
};
```

# Association

Unnamed uni-directional associations are represented as attributes in the UML model.
The translation of such attributes thus follows the rules in Attribute.

Other associations are not supported when generating Java code.

# Datatype

**A UML datatype is mapped to a Java enumeration. Each literal is translated to an enum constant.**

Datatype members such as constructors, attributes and operations, are translated to corresponding members in the Java enumeration.

**Example 431: Translation of datatypes**

**UML**

```
public datatype Status {
  literals OK = new Status("All is OK!");
  literals PROBLEM = new Status("Problem occurred!");

  private const String description;
  private Status(String s) {
    this.description = s;
  }
}
```

**Java**

```
public enum Status {
  OK("All is OK!"), PROBLEM("Problem occurred!");

  private final String description;
  private Status(String s) {
    this.description = s;
  }
}
```

Note that it is possible to invoke a constructor when initializing the value of an enum literal.

# Expression

**UML expressions are translated to Java by translating each part of the expression individually. Constant expressions are not evaluated during translation (although it would be possible in most cases).**

The translation of most UML expression is straight-forward, as shown in the table below:

| UML Expression | Java Expression | UML Example | Java Example |
|---|---|---|---|
| Parenthesis expression | Parenthesis expression | `(a+b)` | `(a+b)` |
| Unary expression | A unary expression, where the Java operator to use is decided by the specified UML operator. | `not m_bOk`<br>`++var` | `!m_bOK`<br>`++var` |
| Binary expression | A binary expression, where the Java operator to use is decided by the specified UML operator. | `a + b` | `a + b` |
| This expression | 'this' expression | `this` | `this` |

| UML Expression | Java Expression | UML Example | Java Example |
|---|---|---|---|
| Call expression | Call expression | `foo(3)` | `foo(3)` |
| Field expression | Member access expression | `x.y` | `x.y` |
| Index expression | Subscripting operator ('[]') | `coll[4]` | `coll[4]` |
| Create expression | 'new' operator | `new C(1,2)` | `new C(1,2)` |
| Conditional expression | Conditional expression | `b ? x1 : y1` | `b ? x1 : y1` |
| Real value | float literal | `float a = 3.14;` | `float a = 3.14;` |
| Integer value | integer literal | `long a = 4;` | `long a = 4;` |
| Charstring value | string literal | `String s = "hola";` | `String s = "hola";` |
| Character value | character literal | `char c = 'q';` | `char c = 'q';` |

The translation of remaining expressions is described in the rest of this chapter.

## Identifier

**An identifier is translated in the same way as the name of the definition to which it is bound.**

This rule applies both when the identifier is part of an expression and when it represents a reference (compare Name of Definitions).

The translation of identifiers which are references to predefined UML or Java types is described in Predefined types. References to other predefined UML definitions is described in Reference to non-type UML predefined definition.

### Reference to base class

**A reference to the base class (a.k.a. super class) of a UML class can be made in action code using the name of the base class in a qualifier. Such references will be transformed to Java references using the 'super' keyword as the qualifier.**

The same translation rule applies if the base class is referenced through the use of a syntype.

**Example 432: Translation of base class references**

**UML**

```
class D
{
    public void foo() {}
}

class C : D
{
    syntype inherited = D;
    public void foo()
    {
      D::foo();
      inherited::foo();
    }
}
```

**Java**

```
class D {
    public void foo() {

    }
}

class C extends D {

    public void foo() {
        super.foo();
        super.foo();
    }
}
```

### Reference to non-type UML predefined definition

Usage of some UML predefined non-type definitions are translated in a special way by the Java code generator. Translation of references to predefined UML types are described in Predefined types.

The table below lists predefined UML definitions that are supported by the Java code generator:

| Referenced UML Definition | Java Translation |
|---|---|
| `is` | `instanceof` |
| `as` | Type cast operator |

**Example 433: Translation of UML predefined `is` and `as`**

**UML**

```
C var = new C();
if (is<D>(var))
{
    D d = as<D>(var);
}
```

**Java**

```
C var = new C();
if (var instanceof D)
{
    D d = (D)var;
}
```

## Informal Expression

**A UML informal expression is translated into a verbatim copy of the expression text in generated Java code.**

Informal expressions can thus be used as a mechanism for generating Java code which otherwise cannot be generated from UML constructs by the Java code generator. It is also useful in situations when interfacing with legacy Java code which is not represented in the UML model.

If the informal expression contains a reference to a UML definition, it is translated according to the ordinary rule for an Identifier before the expression is copied into the generated Java.

**Example 434: Translation of information expressions**

**UML**

```
RemoteClass x = new RemoteClass();
long ext = [[##(x).remoteMethod()]];
```

**Java**

```
RemoteClass x = new RemoteClass();
long ext = x.remoteMethod();
```

## TimerActive Expression

**A UML TimerActive expression is translated to a call of the `isActive` method on the Java timer attribute corresponding to the timer that is referenced by the expression.**

**Example 435: Translation of TimerActive expressions**

Checking the activeness of the timer defined in Example 454 on page 1420.

**UML**

```
boolean b = Clock.isActive();
```

**Java**

```
Boolean b = timer_Clock.isActive();
```

**Note**

*The only purpose of the actual arguments that may be used in the TimerActive Expression in UML is to identify which (possibly overloaded) version of a timer that shall be queried for its activeness. These actual arguments are thus not visible in the Java translation.*

## Now Expression

**A UML Now expression is translated to a call of the static `now` method on the TOR Time class.**

The returned **Time** object is converted to a double value by calling the method `to_double()` on it. Thereby it can participate in any Java expression of type double.

See Example 451 on page 1417 for an example.

# Template

**A UML class or interface template is translated to a Java class or interface generic type.**

**A UML operation template is translated to a Java method with generic type arguments and/or return type.**

**Example 436: Translation of template definitions**

**UML**

```
template <type T>
class C
{
  T t;
  public T get_t()
  {
    return t;
  }
}
```

**Java**

```
class C<T> {
  T t;
  public T get_t()
  {
    return t;
  }
}
```

## Atleast Constraints

**A UML Atleast constraint on a formal template parameter is translated to an upper bounded wildcard (`extends`) on the corresponding Java generic type parameter.**

Note that it is not possible to represent Java wildcards with a lower bound (`super`) in UML.

**Example 437: Translation of atleast constraints**

**UML**

```
class MyClass {}

template <type T atleast MyClass>
class C {}
```

**Java**

```
class MyClass {}

class C<T extends MyClass> {}
```

**Note**

*UML supports many more advanced usages of atleast constraints. These constructs have no correspondence in Java and are not translated.*

## Template Instantiation

**An instantiation of a UML template is translated to a usage of the Java generic type that is the translation of the template.**

If an actual template type parameter is an ordinary reference to a type, it is translated to a reference of the corresponding Java type.

**Note**

*Generic type parameter wildcards used outside the definition of the parameter are not supported when generating Java code.*

**Example 438: Translation of template instantiations**

This example instantiates the template `C` from Example 436 on page 1405.

**UML**

```
C<MyClass> v1 = new C<MyClass>();
MyClass mc = v1.get_t();
```

**Java**

```
C<MyClass> v1 = new C<MyClass>();
MyClass mc = v1.get_t();
```

# Action

**A UML action is translated to a Java statement.**

The translation of most UML actions is straight-forward, as shown in the table below:

| UML Action | Java Statement | UML Example | Java Example |
|---|---|---|---|
| Compound action | compound statement | `{ v = v + 1; }` | `{ v = v + 1; }` |
| Continue action | continue statement | `continue;` | `continue;` |
| Break action | break statement | `break;` | `break;` |
| If action | if statement | `if (b) { ...} else { }` | `if (b) { ...} else { }` |

The translation of other UML actions is described in the rest of this chapter.

## Definition Action

**A UML definition action where the associated definition is an Attribute is translated to a local variable definition in Java.**

**Example 439: Translation of definition actions**

**UML**

```
class C {
  public void foo(){
```

```
    integer v = 4;
  }
}
```

**Java**

```
class C {
  public void foo(){
    integer v = 4;
  }
}
```

---

If the associated definition is not an Attribute, the definition action will not be translated.

## Expression Action

The translation of a UML expression action depends on what kind of expression that is associated with the action.

**An expression action with an associated empty expression, is translated to an empty Java statement.**

**An expression action with an associated Informal Expression, is translated to a copy of the informal text.**

**An expression action with an associated Call expression, is translated by appending a semicolon (;) to the translation of the call expression.**

**An expression action with an associated Create expression, is translated by appending a semicolon (;) to the translation of the create expression.**

**Example 440: Translation of expression actions**

**UML**

```
void foo(){
    ;
    [[new C().doIt()]];
    open(true);
    new Integer();
}
```

**Java**

```
void foo(){
```

```
  ;
  new C().doIt();
  open(true);
  new Integer();
}
```

# Try Action

**A UML Try action with an associated catch clause, is translated to a Java try statement with a catch clause.**

**Example 441: Translation of try and throw actions**

**UML**

```
class C {
  void foo(boolean b) throws InternalError {
    if (b)
      throw InternalError();
  }

  void bar() {
    try {
      foo(false);
    }
    catch(InternalError e)
    {}
  }
}
```

**Java**

```
class C {
    void foo( boolean b) throws InternalError {
        if (b)
            throw InternalError();
    }
    void bar() {
        try
        {
            foo(false);
        }
        catch ( InternalError e)
        {

        }
    }
}
```

## Throw Action

**A UML Throw action is translated to a Java throw statement.**

See Example 441 on page 1409 for an example.

## Loop Action

**A UML Loop action is translated to a Java while statement, a do-while statement or a for statement.**

These three statements are all variants of the same construct, and which statement that will be used is determined by the syntax variant used in the UML model.

**Example 442: Translation of loop actions**

**UML**

```
int a = 0;
for (int i = 0; i < 10; i++) {
    a = a + 1;
}
while (a > 0)
    a = a - 1;
do {
    a = a + 1;
} while (a < 10);
```

**Java**

```
int a = 0;
for (int i = 0; i < 10; i++) {
    a = a + 1;
}
while (a > 0)
  a = a - 1;
do {
a= a+ 1;
} while (a < 10);
```

**Note**

*The "foreach" variant of a for-statement introduced in Java 5 cannot be represented in a useful way in UML.*

## Stop Action

**A UML Stop action is translated to a call of the “finish” method on the `statemachine` class that corresponds to the `statemachine` implementation in which the stop action is contained.**

**Example 443: Translation of stop actions**

**UML**

```
stop;
```

**Java**

```
finish();
```

## NextState Action

### Normal NextState action

**A “normal” NextState action (i.e. a NextState action that explicitly specifies a state to enter) is translated to a call of the “enter” method on the Java state attribute that corresponds to the specified state.**

**A NextState action with a 'via' clause, specifying an entry connection point, is translated to a call of the “enter” method on the Java state attribute that corresponds to the specified state.**

The attribute that is the translation of the connection point is passed as argument in the call to “enter”.

**Example 444: Translation of NextState actions**

**UML**

```
nextstate Idle;
nextstate Idle via Cin;
```

**Java**

```
m_s_Idle.enter();
m_s_Idle.enter(m_s_Idle.m_sm.Cin);
```

### History NextState action

**A history NextState action is translated to a call of the "enterHistory" method on the TopRegion attribute of the Java statemachine class that corresponds to the UML implementation which contains the NextState action.**

**Example 445: Translation of history NextState actions**

**UML**

```
nextstate -;
```

**Java**

```
theTopRegion.enterHistory();
```

### Deep history NextState action

**A deep history NextState action is translated to a call of the "enterHistory" method on the TopRegion attribute of the Java statemachine class that corresponds to the UML implementation which contains the NextState action. The "deepHistory" flag is set to true in the call.**

**Example 446: Translation of deep history NextState actions**

**UML**

```
nextstate ^-;
```

**Java**

```
theTopRegion.enterHistory(true /* deepHistory */);
```

## Signal Send Action

**A signal send action is translated to a call of the TOR method `Utilities.sendTo()`, with a dynamically created signal instance as argument.**

If the receiver of the signal is explicitly specified, the second argument in the call of `Utilities.sendTo` is a reference to the receiver. If the signal instead is sent via a port (so that the receiver is located at run-time using the port and connector structure) the second argument is a reference to the port.

**Example 447: Translation of signal send actions**

**UML**

```
output c.Ping("hello"); // Sending a signal directly to
a receiver
output Pong() via MyPort; // Sending a signal via a port
```

**Java**

```
Utilities.sendTo(new Ping("hello"), c);
Utilities.sendTo(new Pong(), m_owner.MyPort);
```

If a signal send action specifies the sending of more than one signal, there will be one "`sendTo`" call for each sent signal.

# Decision Action

**A UML decision action is translated to a either a Java switch-statement, with one case-branch for each decision answer, or an if-statement with one else-branch for each decision answer. In the latter case the if-statement will be placed in an empty switch-statement if there are break statements inside.**

Java switch statements are much more constrained than UML decision actions. Therefore a UML decision action can only be translated to a Java switch statement if the decision expression is a simple reference to an attribute typed by either of the following

- a primitive data type (byte, short, char, int)
- a class wrapping the above primitive types (Byte, Short, Character, Integer)
- an enumerated type

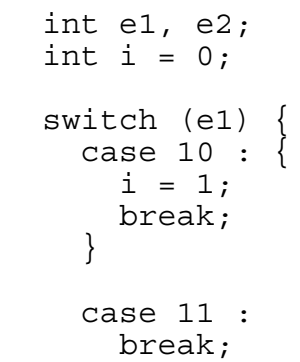
It is also required that each decision answer expression is a simple value. In all other cases the decision action will be translated to a Java if-statement. The if-statement will have one branch for each decision answer.

If a generated Java if-statement contains break statements, the entire if-statement is placed in the default branch of an empty switch statement (so that the break statements will break out from the switch).

**Example 448: Translation of decision actions**

The first decision action below will be translated to a Java switch-statement, the second to a Java if-statement, and the third will be translated to an if-statement inside a switch-statement.

**UML**



```
int e1, e2;
int i = 0;

switch (e1) {
  case 10 : {
    i = 1;
    break;
  }

  case 11 :
    break;
```

```
  default : {
    i = 3;
  }

}

const int x = 5;
int v;
switch (v) {
  case x : {return;}
  default : ;
}

switch (e2 > 5) {

  case true: {
    i = 1;
    break;
  }

  case false: {
    i = 0;
    break;
  }
}
```

**Java**

```
int e1;
int e2;
int i = 0;
switch (e1) {
  case 10:
    i = 1;
    break;
  case 11:
    break;
  default:
    i = 3;
    break;
}

final int x = 5;
int v;
if (((v) == (x)))
{
  return ;
}
else ;

switch (e2 > 5) {
default:
  if (((e2 > 5) == (true)))
  {
    i = 1;
```

```
    break;
  }
  else if (((e2 > 5) == (false)))
  {
    i = 0;
    break;
  }
}
```

## Return Action

**A Return action that is contained in a UML operation body is translated to a return statement in Java.**

**A Return action that is contained in a transition is translated to a call of the "finish" method on the TopRegion attribute which belongs to the Java statemachine class that is the translation of the UML state machine implementation that contains the transition.**

**Example 449: Return action within a transition**

**UML**

```
start {
    return;
}
```

**Java**

```
public void initialTransition( ) {
    theTopRegion.finish();
}
```

**If the Return action specifies a connection point to return via, the attribute that is the translation of the connection point is passed as argument in the call of "finish".**

**Example 450: Return via connection point**

Return action within a transition specifying an exit connection point.

**UML**

```
start {
    return Cout;
```

```
}
```

**Java**

```
void initialTransition( ) {
    theTopRegion.finish(Cout);
}
```

## Timer Set Action

**A Timer Set action is translated to a call of the “set” method on the Java timer attribute corresponding to the referenced timer.**

The first argument in the call is the specified time-out expression. If no time-out expression is specified there must be a default time-out expression specified in the timer definition, which then is used instead. The second argument in the call is a dynamically created instance of the Java timer class (a class which extends the TOR class TimerEvent). Actual timer parameters are translated to actual constructor parameters in the creation of this instance.

**Example 451: Translation of Timer Set actions**

Setting the timer defined in Example 454 on page 1420.

**UML**

```
set Clock() = now + 4;
set Clock; // Using default timeout value (15)
```

**Java**

```
timer_Clock.set(new Time(Time.now().to_double() + 4),
new C.Clock());
timer_Clock.set(new Time(Time.now().to_double() + 15),
new C.Clock());
```

## Timer Reset Action

**A Timer Reset action is translated to a call of the “reset” method on the Java timer attribute corresponding to the referenced timer.**

**Example 452: Translation of Timer Reset actions**

Resetting the timer defined in Example 454 on page 1420.

**UML**

```
reset Clock();
```

**Java**

```
timer_Clock.reset();
```

**Note**

*The only purpose of the actual arguments that may be used in a Timer Reset action in UML is to identify which (possibly overloaded) version of a timer that shall be reset. These actual arguments are thus not visible in the Java translation.*

# Signal

**A UML signal is translated to a Java class that inherits from the TOR class Event.**

The class has a static “isTypeOf” method which checks if an event is dynamically typed by the signal that corresponds to the class. The implementation of this method uses the Java `instanceof` operator.

The Java event class is placed in the Java scope that corresponds to the UML scope of the definition that owns the signal. If that scope is a Java class, the Java event class is a static nested class. If it instead is a Java package, it is an ordinary Java class.

## Signal Parameter

If a signal has parameters the Java event class gets a constructor with one parameter for each signal parameter. In addition there will be one public attribute for each signal parameter. The name, type, multiplicity etc. of a constructor parameter and class attribute are all identical to the signal parameter for which they are generated.

If a signal parameter has no name the corresponding constructor parameter and class attribute will get the name "parX", where X is the zero-based index of the signal parameter.

**Example 453: Translation of signals with parameters**

**UML**

```
class C
{
    signal Ping (String, long i = 4);
}
```

**Java**

```
public class C
{
    public static class Ping extends Event
    {
        public String par0;
        public long i = 4;

        public Ping(String par0, long i)
        {
            this.par0 = par0;
            this.i = i;
        }

        public Ping(String par0)
        {
            this.par0 = par0;
        }

        public static boolean isTypeOf(Event e)
        {
              return e instanceof Ping;
        }
    }
}
```

---

The name of a signal class is the translation of the name of the corresponding signal (compare Name of Definitions). However, since a signal is an event class there may be more than one signal in the same UML scope having the same name (overloading). If that is the case, all overloaded signals with the same name will have their names suffixed with the types of the signal parameters.

# Timer

**A UML timer is translated to a Java class that inherits from the TOR class TimerEvent.**

This is done in exactly the same way as when translating a Signal. In addition an attribute is generated in the Java class that is the translation of the UML definition that owns the timer. That attribute is called "timer_T", where T is

the name of the timer, and it is typed by the TOR class TimerObject. It is also initialized to an instance of that class, passing this as constructor actual argument.

**Example 454: Translation of timers**

**UML**

```
class C
{
    timer Clock() = 15;
}
```

**Java**

```
public class C
{
    public TimerObject timer_Clock = new
TimerObject(this);

    public static class Clock extends TimerEvent
    {
          public static boolean isTypeOf(Event e)
        {
            return e instanceof Clock;
        }
    }
}
```

**Note**

*The default timeout value (15 in Example 454 on page 1420) is not directly visible in the Java translation result. However, it is used in the translation of a Timer Set Action which do not specify a timeout value.*

# State Machine

There are three kinds of state machines in UML.

1. **Classifier behavior statemachine.** This is a constructor statemachine (usually called “initialize”) owned by an active class.
2. **Inline statemachine.** This is a statemachine owned by a composite state.
3. **Stand-alone statemachine.** This is a statemachine which can be referenced from one or many composite states.

The statemachine translation rules are partially dependent on the kind of statemachine. Such variations are marked in the text below by referring to the numbers in the above classification list.

**A UML statemachine is translated to a Java class which extends the TOR class StateMachine.**

The name of the Java state machine class becomes (depending on the kind of statemachine):

1. "C_SM", where C is the name of the owner class, and SM is the name of the state machine.
2. "C_S_SM", where S is the name of the composite state, and C is the name of the statemachine class that is the translation of the statemachine implementation that contains the state. SM is the name of the statemachine.
3. "SM", where SM is the name of the statemachine.

A Java statemachine class contains the following members:

- An attribute called "m_owner". The type of this attribute depends on the kind of statemachine:

1. the Java class that is the translation of the owning active class.
2. the Java class that is the translation of the owning statemachine.
3. the TOR class DispatchableClass.

- In case of a statemachine of kind 2) or 3) an attribute "m_ownerState" is also generated. The type of this attribute is the owner state class (2) or the TOR class State (3).
- A constructor with an implementation that initializes the "m_owner" attribute. If the class also has an "m_ownerState" attribute it is also initialized.
- A "getDispatchableClass" method. The implementation of this method returns an expression which evaluates to the dispatchable class to which the statemachine belongs. This expression depends on the kind of statemachine:

1. the "m_owner" attribute as a DispatchableClass.
2. "m_owner.getDispatchableClass()"
3. "m_owner"

- An attribute “theTopRegion” typed by the TOR class TopRegion. It is initialized to a new instance of TopRegion.
- A redefinition of the method “init”. The implementation of this method calls addRegion(theTopRegion).

**Note**

*It is possible to use a simple state machine as the implementation of an ordinary operation. If such a state machine implementation has no states, it will be translated as an ordinary operation body (the actions of the start transition will be put in the Operation Body.*

**Example 455: Translation of a classifier behavior statemachine (1)**

**UML**

```
active class C {
    statemachine initialize {
        ...
    }
}
```

**Java**

```
public class C_initialize extends StateMachine
{
    public TopRegion theTopRegion = new TopRegion(this);

    public C m_owner;

    public C_initialize(C owner)
    {
          m_owner = owner;
    }

    public void init()
    {
          addRegion(theTopRegion);
    }

    public DispatchableClass getDispatchableClass()
    {
          return m_owner;
    }
}
```

**Example 456: Translation of a composite state state machines (2 and 3)**

**UML**

```
active class C {
    statemachine initialize {
        state S1 : statemachine initialize {}; // 3)
        state S2 : SM {}; // 2)
    }
}

statemachine SM {}
```

### Java

```
public class C_initialize extends StateMachine
{
    // see Example 455 on page 1422 for the translation
}

public class C_S1_initialize extends StateMachine
{
    public TopRegion theTopRegion = new TopRegion(this);

    public C_initialize m_owner;
    public C_initialize.S1 m_ownerState;

    public C_S1_initialize(C_initialize owner,
C_initialize.S1 ownerState)
    {
          m_owner = owner;
        m_ownerState = ownerState;
    }

    public void init()
    {
        addRegion(theTopRegion);
    }

    public DispatchableClass getDispatchableClass()
    {
        return m_owner.getDispatchableClass();
    }
}

public class SM extends StateMachine
{
    public TopRegion theTopRegion = new TopRegion(this);

    public DispatchableClass m_owner;
    public State m_ownerState;

    public SM(DispatchableClass owner, State ownerState)
    {
          m_owner = owner;
        m_ownerState = ownerState;
    }

    public void init()
```

```
        {
            addRegion(theTopRegion);
        }

        public DispatchableClass getDispatchableClass()
        {
            return m_owner;
        }
    }
```

Note that the translation of the composite states are not shown in this example. See instead Example 458 on page 1426.

---

**Non-local definition access from a statemachine implementation**

Often in UML a definition can be referenced from a statemachine implementation without use of qualifier. This is because from a given statemachine implementation all definitions defined in enclosing scopes are directly accessible. This includes

1. Local definitions (within the same statemachine implementation)
2. Definitions in containing state machines (in case the reference is made from an inline statemachine of a composite state)
3. Definitions in the containing active class
4. Global definitions

In Java, references of definitions of category 2 and 3 require an access prefix using one or many “m_owner” attribute references. If the reference is located in a context that will be translated to a Java state class, one extra “m_owner” is added to the qualifier (since the state class has an “m_owner” attribute referring to the owning statemachine class). See Triggered Transition for an example.

# State

**A UML state is translated to a Java class which extends the TOR class State.**

The state class is placed within the Java statemachine class as a static nested class. This avoids the need to do name mangling of state names (two statemachine implementations may have states with the same name).

Each state class is instantiated in the “init” method of the state machine class. Each state object is stored in a dedicated attribute for the state. This attribute is called “m_s_S”, where S is the name of the state. It has protected visibility.

The Java state class has the following members:

- An attribute “m_owner” which is typed by the statemachine class that is the translation of the owning statemachine implementation.
- A constructor which initializes “m_owner”.
- An overload of the abstract method “execute”. It contains if-statements for all triggered transition originating from the state. This means that the state class contains the implementation of all Triggered Transitions originating from that state.

**Example 457: Translation of states and start transitions**

**UML**

```
active class C {
    statemachine initialize {
        start {
            nextstate Idle;
        }
        state Idle;
    }
}
```

**Java**

```
public class C extends DispatchableClass { … }

public class C_initialize extends StateMachine
{
    // … see State Machine translation above …

    public void initialTransition()
    {
          m_s_Idle.enter();
    }

    public void init()
    {
          // …
        m_s_Idle = new Idle(theTopRegion, this);
          m_s_Idle.init();
    }

    public static class Idle extends State
    {
        public C_initialize m_owner;
```

```
        public Idle(Region region, C_initialize owner)
        {
        super(region);
        m_owner = owner;
        }

        public Dispatchable.EventAction execute(Event e)
        {
        // … Transition if-statements …
            // (see Triggered Transition below)

        return Dispatchable.EventAction.NoMatch;
        }
    }

    protected Idle m_s_Idle;
}
```

For brevity, the example has left out most things not related to the translation of the state.

---

### Composite state

A state may contain or reference a state machine, in order to become a composite state. A composite state is translated as an ordinary state, with the following additions:

- The state class gets an attribute “m_sm” typed by the statemachine class that is the translation of the referenced statemachine. It has public visibility.
- Addition of an overloaded method “init” which initializes “m_sm” to a new instance of the statemachine class, and calls “init” on the created object.
- Addition of an overloaded method “getStateMachine” which returns “m_sm”.

**Example 458: Translation of composite states**

**UML**

See Example 456 on page 1422.

**Java**

```
public static class S1 extends State
```

```
{
    public C_initialize m_owner;

    public S1(Region region, C_initialize owner)
    {
        super(region);
        m_owner = owner;
    }

    public C_S1_initialize m_sm;

    public void init()
    {
        m_sm = new C_S1_initialize(m_owner, this);
        m_sm.init();
    }

    public StateMachine getStateMachine()
    {
        return m_sm;
    }
}

public static class S2 extends State
{
    public C_initialize m_owner;

    public S2(Region region, C_initialize owner)
    {
        super(region);
        m_owner = owner;
    }

    public SM m_sm;

    public void init()
    {
        m_sm = new SM(m_owner.getDispatchableClass(),
this);
        m_sm.init();
    }

    public StateMachine getStateMachine()
    {
        return m_sm;
    }
}
```

Note that a composite state which references a stand-alone statemachine (SM in the example) implements the “init” method slightly differently (it calls `getDispatchableClass()` on “m_owner” to get a reference on the dispatchable class).

## Start Transition

**A start transition is translated to a method "initialTransition" in the Java statemachine class that is the translation of the statemachine implementation in which the start transition is contained.**

The implementation of "initialTransition" is the translation of the actions of the start transition. This translation follows exactly the same rules as for any other Action.

See Example 457 on page 1425 for an example of the translation of a start transition.

## Triggered Transition

**A triggered transition is translated to a protected method in the Java statemachine class that is the translation of the statemachine implementation in which the triggered transition is contained.**

The name of the transition method is:

1. if the transition has no guard and no asterisk trigger: `trans_<StateNames>_<SignalClassNames>`
2. otherwise: `trans_<StateNames>_<SignalClassNames>_<GUID>`

<StateNames> are the names of all states in which the transition may be triggered (separated by underscores), and <SignalClassNames> are the names of all event classes corresponding to signals that may trigger the transition. <GUID> is the GUID of the triggered transition.

For each triggered transition an if-statement is also generated in the "execute" method of each state class, corresponding to a state in which the transition may be triggered. This if-statement uses the "isTypeOf" method to test that the dynamic type of the received signal event matches the static event type specified in the trigger of the transition. If so, the following happens:

1. Actual arguments of the event, if any, are assigned to the referenced attributes. In Java a cast from the generic Event type to the specific signal event type is needed in order to access the event parameters.
2. The current state is left by calling the "leave" method.
3. The transition method is called.

4. Finally the “execute” method returns “Dispatchable.EventAction.Consumed” to indicate that the event has been consumed.

**Example 459: Translation of triggered transitions**

**UML**

```
active class C {
    private long count;
    protected String strName;
    public C destination;

    statemachine initialize {
        String 'from';
        start {
            count = 0;
            ^ destination.sig(strName);
            nextstate Idle;
        }

        state Idle;
        for state Idle;
            input sig('from') {
                count = count + 1;
                ^ destination.sig('from');
                nextstate Idle;
            }
        }
    }
}
```

**Java**

```
public class C extends DispatchableClass
{
    // … see Active Class translation above …

    public long count;
    public String strName;
    public C destination;
}

public class C_initialize extends StateMachine
{
    // … see State Machine translation above …

    protected String from;

    public void initialTransition()
    {
        m_owner.count = 0;
        sendTo(new sig(m_owner.strName),
m_owner.destination);
          m_s_Idle.enter();
```

```
        }

        public void init()
        {
              // …
            m_s_Idle = new Idle(theTopRegion, this);
              m_s_Idle.init();
        }

        public static class Idle extends State
        {
            public C_initialize m_owner;

            public Idle(Region region, C_initialize owner)
            {
            super(region);
            m_owner = owner;
            }

            public Dispatchable.EventAction execute(Event e)
            {
            if (sig.isTypeOf(e))
                 {
                     m_owner.from = ((sig) e).sender;
                     leave();
                     m_owner.trans_Idle_sig(e);
                     return
    Dispatchable.EventAction.Consumed;
                 }
            return Dispatchable.EventAction.NoMatch;
            }
        }

        protected trans_Idle_sig(Event e)
        {
            m_owner.count = m_owner.count + 1;
            sendTo(new sig(from), m_owner.destination);
            m_s_Idle.enter();
        }

        protected Idle m_s_Idle;
    }
```

For brevity, the example has left out most things not related to the translation of the triggered transition.

Note in this example that the statemachine attribute “from” and the active class attributes “count”, “strName” and “destination” all get public visibility in Java (see Visibility of Definitions), although the UML visibilities are different.

The event that triggered a transition is available as a parameter of the generated “transition methods”. This parameter can be accessed from the actions of the UML transition by using target Java code.

**Note**

*The actual arguments of a received signal are assigned to the variables referenced in the UML signal sending action.*

### Multiple triggers

**If a triggered transition has more than one trigger, the corresponding Java if-statement will use an expression that becomes true if any of the referenced events are received.**

If the received signals have parameters the assignment of their actual values to attributes must be done within an if-statement that ensures that the actual event type matches the expected signal.

A special case of multiple triggers is “asterisk input”, which means that any signal can be received in a state. The Java translation of such a triggered transition is thus an if-statement which just checks that the event is not null.

**Example 460: Translation of triggered transitions with multiple triggers**

**UML**

```
state Idle;
for state Idle;
input sig1(a), sig2(b) {
    nextstate Idle;
}
input * {
   stop;
}
```

**Java**

```
public static class Idle extends State
{
    // … see State translation above

    public Dispatchable.EventAction execute(Event e)
    {
        if (sig1.isTypeOf(e) || sig2.isTypeOf(e))
        {
            if (sig1.isTypeOf(e))
               m_owner.a = ((sig1) e).par0; // Assuming
unnamed param.
```

```
            if (sig2.isTypeOf(e))
               m_owner.b = ((sig2) e).par0; // Assuming
unnamed param.
            leave();
            m_owner.trans_Idle_sig1_sig2(e);
            return Dispatchable.EventAction.Consumed;
        }
        if (e)
        {
            leave();

m_owner.trans_Idle__GEN_68y_42UVUZ_42PLLeZR70IzmiZ8I(e);
            return Dispatchable.EventAction.Consumed;
        }
        return Dispatchable.EventAction.NoMatch;
    }

    protected trans_Idle_sig1_sig2(Event e)
    {
        m_s_Idle.enter();
    }

    protected
trans_Idle__GEN_68y_42UVUZ_42PLLeZR70IzmiZ8I(Event e)
    {
        finish();
    }
}
```

The code that transfers actual signal arguments to state machine variables in this case will be placed inside an if-statement checking that the event types match.

---

Note the naming of transition methods which is made up of all names of events which may trigger the transition. In the case of an asterisk input, the GUID of the transition is part of its name.

# Guard

**A guard on a triggered transition is translated to an additional expression in the if-statement for the transition in a state "execute" method. The expression must evaluate to true for the transition to be triggered.**

If the transition only has a guard and no triggers, it should trigger without the reception of an event. Therefore, the if-statement for such a transition will check that the event is null and that the guard condition is fulfilled.

**Example 461: Translation of transition guard conditions**

**UML**

```
state Idle;
for state Idle;
input sig1()[x == y] {...}  // Trigger and guard
[b == true] {...}           // Only a guard
```

**Java**

```
public static class Idle extends State
{
    // … see State translation above

    public Dispatchable.EventAction execute(Event e)
    {
        if (sig1.isTypeOf(e) && (x == y))
        {
            ...
        }
        if (e == null && (b == true))
        {
            ...
        }
        return Dispatchable.EventAction.NoMatch;
    }
}
```

# Label Transition

**A UML label transition is translated to a protected method in the Java statemachine class that is the translation of the statemachine implementation in which the label transition is contained.**

The name of the transition method is "trans_L", where L is the name of the label of the label transition.

**Example 462: Translation of label transitions**

**UML**

```
statemachine initialize {
    Lbl:
    {
        count = 1;
    }
}
```

**Java**

```
public class C_initialize extends StateMachine
{
    // … see State Machine translation above …

    protected void trans_Lbl()
    {
        count = 1;
    }
}
```

---

# Connection Point

**A UML connection point is translated to an attribute in the Java class that is the translation of the statemachine in which the connection point is defined.**

The name of the attribute is the name of the connection point, and its type is

- the TOR class EntryPoint, if the connection point is an entry connection point.
- the TOR class ExitPoint, if the connection point is an exit connection point.

The attribute is initialized to a new instance of its type class.

An entry connection point is referenced from an entry point transition in the statemachine, and an exit connection point is referenced from an exit point transition in a composite state which refers to the statemachine. These transitions are translated to Java methods:

- `entryPointTransitions()`, in the class for the inner statemachine
- `exitPointTransitions()`, in the class for the outer statemachine

The implementations of these methods get a connection point as input parameter, and then uses one if-statement for each entry- or exit-point transition. Before calling the transition method (which is translated as usual, see Triggered Transition) in `exitPointTransitions()` the method “leave” is called on “theTopRegion” in order to leave the current state.

**Example 463: Translation of connection points and entry/exit point transitions —**

**UML**

```
active class C : DispatchableClass {
    statemachine initialize {
    ...
        state Idle : statemachine initialize
            in Cin      // Entry connection point
            out Cout    // Exit connection point
        {
            start Cin {
                ...
            }

            start {
                ...
            }

            state WaitForSig;
            for state WaitForSig;
            input sig() {
                return Cout;
            }
        };

        for state Idle;
        [Cout] {
            count = 14;
            stop;
        }
    }
}
```

**Java**

```
public class C_initialize extends StateMachine
{
    public static class Idle extends State
    {
        public C_initialize_Idle_initialize m_sm;

        ...
    }

    public Idle m_s_Idle;

    protected void trans_Idle_Cout()
    {
        m_owner.count = 14;
        finish();
    }

    public void exitPointTransitions(ExitPoint ep)
    {
        if (ep == m_s_Idle.m_sm.Cout)
        {
            theTopRegion.leave();
            trans_Idle_Cout();
```

```
        }
    }
}

class C_initialize_Idle_initialize extends StateMachine
{
    EntryPoint Cin = new EntryPoint(this);
    ExitPoint Cout = new ExitPoint(this);

    protected void trans_Cin()
    {
        ...
    }

    public void initialTransition() { … }

    public void entryPointTransitions(EntryPoint ep)
    {
        if (ep == Cin)
        {
            trans_Cin();
        }
    }

    Idle m_ownerState;

    public static class WaitForSig extends State { ... }
};
```

Details not related to the translation of the connection points have been omitted from the example.

---

Note that the names of the transitions that are triggered by connection points contain the name of the connection point instead of the name of a signal.

# Architecture

The architecture (or internal structure) of a class is composed of the attributes of the class (parts), the ports of the class and the connectors that link them together. This section describes how these UML constructs are translated to Java code.

## Port

**A UML port is translated to one or two attributes typed by the TOR class Port. The attributes will be placed in the Java class that is the translation of the UML class which contains the port.**

The names of the created attributes are <portName>_in (generated for ports which realize at least one definition) and <portName> (generated for ports which require at least one definition), where <portName> is the name of the port. If the port does not require or realize any definition it will not be translated to Java.

The attributes are initialized to an instance of the Port class with information passed to the constructor about the DispatchableClass that owns the port, and whether the port is an 'in' or 'out' port.

**Example 464: Translation of ports**

**UML**

```
active class C : DispatchableClass {
  port p in with s1 out with s2;
}
```

**Java**

```
class C extends DispatchableClass {
  Port p = new Port(this, Port.PortKind.OutPort);
  Port p_in = new Port(this, Port.PortKind.InPort);
}
```

# Connector

**A UML connector is translated to an attribute of the containing class typed by the TOR class Connector. The name of the created attribute is the name of the Connector.**

If the Connector has an empty name, a default name 'connector_N' is used. N is a number starting at 1 and incremented by one until the name of the Java attribute becomes unique in the containing class.

Code is also generated in the 'init' method of the containing class which calls the 'connect' method on the Connector attribute in order to connect the ports as specified by the Connector.

**Example 465: Translation of connectors**

**UML**

```
active class C : DispatchableClass {
  port p in with s1 out with s2;
}

active class Container : tor::DispatchableClass
{
    part C a;
    part C b;
    connector to a.p to b.p;
}
```

**Java**

```
class C extends DispatchableClass {
  Port p = new Port(this, Port.PortKind.OutPort);
  Port p_in = new Port(this, Port.PortKind.InPort);
}

class Container extends DispatchableClass {
    public C a = new C();
    public C b = new C();

    public void init() {
        addToCurrentDispatcher(a);
        a.init();
        addToCurrentDispatcher(b);
        b.init();

        c.connect(a.p, b.p_in);
    }
    public void start() {
        a.start();
        b.start();
```

```
        super.start();
    }
    Connector = new Connector();
}
```

## Dynamic Collection Attribute Typed by Active Class

Class attributes with non-single multiplicity, where the initial number of instances is lower than the maximum number of instances, are dynamic collection attributes. That is, the collection of instances stored in such an attribute at run-time is dynamic and may change during the execution of the Java application.

**For each dynamic collection attribute in UML that is typed by an active class, a public utility method is generated in the Java class that contains the corresponding Java attribute.**

The name of this utility method is `<attributeName>_insert`, where `<attributeName>` is the name of the attribute.

The purpose of this utility method is to

1. insert a new instance of the active class into the attribute's collection
2. add the new instance to the same dispatcher as is used by the owning class instance
3. initialize the new instance
4. connect the ports of the new instance to other ports based on defined connectors
5. start the new instance (to allow its statemachine to begin execute)

The implementation of 1) above depends on the multiplicity of the attribute. If the attribute has single multiplicity (i.e. 0..1), this part of the method is just a plain assignment of the member attribute. If the multiplicity is > 1 this part of the method will call 'add' on the attribute in order to add the new instance last in the list. It is required that the collection type used for the attribute implements the `java.util.List` interface so that the 'add' method can be called.

The implementation of 4) depends on whether or not connectors are connected to the attribute. It also depends on the multiplicity of the other attribute to which the connector is connected. If no connectors are connected

to the attribute, this part of the method is empty. Otherwise the 'connect' method on the Connector attribute is called in order to connect the ports as specified by the Connector. In case of single target attribute multiplicity this is done by one single call of 'connect' per outgoing connector. In case of non-single target attribute multiplicity there will be a for-loop generated for each outgoing connector in which 'connect' is called once for each instance in the target attribute of the connector. It is required that the collection type used for the target attribute implements the `java.lang.Iterable` interface so that the foreach Java construct can be used for iterating over the instances.

The generated utility methods are illustrated by the examples below. They assume the presence of two active classes D and E, each having a port called ‘p’.

**Example 466: ‘insert’ method generation (attributes with single multiplicity connected by a connector)**

**UML**

```
active class C
{
  public D[0..1] p1;
  public E[0..1] p2;
  connector c from p1.p to p2.p with Sig;
}
```

**Java**

```
class C extends DispatchableClass
{
  Connector c = new Connector();

  public void p1_insert(D p)
  {
    p1 = p; // p1 has single multiplicity
    addToCurrentDispatcher(p);
    p.init();
    if (p2 != null)
      c.connect(p.p, p2.p_in); // p2 has single
multiplicity
    p.start();
  }

  public void p2_insert(E p)
  {
    p2 = p; // p2 has single multiplicity
    addToCurrentDispatcher(p);
    p.init();
    if (p1 != null)
      c.connect(p1.p, p.p_in); // p1 has single
```

```
multiplicity
    p.start();
  }
}
```

**Example 467: 'insert' method generation (attributes with non-single multiplicity connected by a connector)**

## UML

```
active class C
{
  public D[*] p1;
  public E[*] p2;
  connector c from p1.p to p2.p with Sig;
}
```

## Java

```
class C extends DispatchableClass
{
  Connector c = new Connector();

  public void p1_insert(D p)
  {
    p1.add(p); // p1 has non-single multiplicity
    addToCurrentDispatcher(p);
    p.init();
    for (E i : p2) // p2 has non-single multiplicity
    {
      c.connect(p.p, i.p_in);
    }
    p.start();
  }

  public void p2_insert(E p)
  {
    p2.add(p); // p2 has non-single multiplicity
    addToCurrentDispatcher(p);
    p.init();
    for (D i : p1) // p1 has non-single multiplicity
    {
      c.connect(i.p, p.p_in);
    }
    p.start();
  }
}
```

### Inserting an active instance into a dynamic collection attribute

In order to make use of the 'insert' utility method that is generated for a dynamic collection attribute the UML TOR library contains an operation `tor::insert<Any>(Any)`. Calls to this operation are translated into calls of the corresponding utility method in generated Java code.

**Example 468: Translation of calls to tor::insert**

**UML**

```
part Ping[*] p1;
…
// Insert a new instance of Ping into p1.
tor::insert<p1>(new Ping());
```

**Java**

```
…
p1_insert(new Ping());
```

Note that if the call to tor::insert is not made by the class that owns the dynamic collection attribute, an access prefix will be generated in Java. The access prefix is the same as would have been used in other references to the attribute from that context. See Non-local definition access from a statemachine implementation for more information.

## Class Composite Structure Initialization

When creating an instance of a class with an internal structure consisting of parts with a multiplicity that specifies an initial number of allocated instances, these initial instances will be automatically

1. created
2. added to the same dispatcher as is used by the containing class instance
3. initialized
4. connected to other attributes as specified by connectors in the containing class
5. started

Task 1) is taken care of by the generation of Java field initializers and a Java instance initializer block. Task 2 - 4) are taken care of by code generated into the 'init' method of the containing class. Task 5) is taken care of by code generated into the 'start' method of the containing class.

Note that if the containing class is not active, only task 1) is performed.

**Example 469: Generation of code for initializing the composite structure of a class**

**UML**

```
active class C
{
  part D[0..1] p1_single;
  part E[1] p2_single;
  part F[*]/4 p1_multi;
  part G[8] p2_multi;

  connector c1 from p1_single.port1 with sig to
p2_single.port2 with sig;
  connector c2 from p1_multi.port1 with sig to
p2_multi.port2 with sig;
}
```

**Java**

```
class C extends DispatchableClass
{
  D p1_single;
  E p2_single = new E();
  java.util.Vector<F> p1_multi = new
java.util.Vector<F>();
  java.util.Vector<G> p2_multi = new
java.util.Vector<F>();

  {
    for (int i = 0; i < 4; i++)
      p1_multi.add(new F());

    for (int i = 0; i < 8; i++)
      p1_multi.add(new G());
  }

  public void init()
  {
    // Code related to statemachine initialization for C
(if applicable)
    // (see Active Class)

    addToCurrentDispatcher(p1_single);
    addToCurrentDispatcher(p2_single);
    addToCurrentDispatcher(p1_multi);
    addToCurrentDispatcher(p2_multi);
```

```
    p1_single.init();
    p2_single.init();
    init(p1_multi); // Using generic TOR method for
multi-initialization
    init(p2_multi); // ----------------------"----------
---------------

    c1.connect(p1_single.port1, p2_single.port2_in);
    for (F i : p1_multi)
    {
      for (G i2 : p2_multi)
        {
          c2.connect(i.port1, i2.port2_in);
        }
    }

    super.init();
  }

  public void start()
  {
    p2_single.start();
    start(p1_multi);
    start(p2_multi);
    super.start();
  }

}
```

Note that there are two ways in UML to specify that an attribute with non-single multiplicity should have initial instances allocated.

- Specifying a fix constant value as multiplicity

  ```
  C[6] c; // 6 initial instances in c
  ```

- Specifying an initial count for the attribute

  ```
  C[*]/6 c; // 6 initial instances in c
  ```

When translating UML to Java there is no difference between these two ways of specifying initial instances. That is, generated Java code does not ensure that a constraint on the maximum number of instances is fulfilled.

# Translation Customization

The output of the Java code generator can be customized using agents. This gives the user a very precise control of the generated code.

The generated code can be customized by adding arbitrary text at certain locations in the generated files. This makes it possible to generate additional code, comments etc. in conjunction with the ordinary code that gets generated.

It is also possible to implement custom transformations for model constructs which are not directly handled by the Java code generator, or to modify the default transformations made by the code generator.

All these customization possibilities are based on tool events and agents triggered during code generation by these tool events. See Chapter 75, Agents for more information about these concepts.

## Adding Text During Code Generation

During code generation it is possible to add arbitrary text

- at the beginning or end of a generated file (see tool event JavaPrintFile)
- just before or just after the generation of a Java definition (see tool event JavaPrintDefinition)

**Example 470: Printing a custom header in generated Java files**

Assume we want to print a custom header in the form of a comment block, in all generated Java files. We can do this by defining an agent that triggers on the JavaPrintFile tool event on <<before processing>>. The implementation of the agent can for example be done using the Tcl script below:

```
proc PrintHeader { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  set buffer [lindex $ap 0]
  u2::AddSourceBufferText $buffer "// This is a
generated file! Do not edit!\\n"
}
```

# Implementing Custom Transformations

The tool event Transformation can be used to implement custom transformations in order to generate Java code for model constructs for which no standard Java transformation is available. It can also be used as a means for customizing the standard transformations.

**Example 471: Implementing a custom Java transformation**

Assume we want to support the singleton design pattern when generating Java code. At the model level we don't want to expose the implementation of this design pattern, but instead we just want to stereotype classes with a <<singleton>> stereotype to express that they should be generated as singleton classes.

We can implement this transformation by defining an agent that triggers on the Transformation tool event on <<before processing>>. The implementation of the agent can for example be done using the Tcl script below:

```
proc CustomTransformation { triggeredBy timing context
server agentParameters } {
  upvar 1 $agentParameters ap
  set messageList [lindex $ap 0]
  set roots [lindex $ap 2]
  set model [lindex $ap 3]
  foreach r $roots {
      if {[u2::HasAppliedStereotype $r "singleton"]} {
          u2::AddMessage $messageList "Transforming
singleton class" Information -subject $r

          ## Add attribute
          set type [u2::GetValue $r "Name"]
          set att [u2::Parse $model "private static part
$type m_inst;"]
          u2::SetEntity $r "OwnedMember" $att

          ## Add private constructor
          set constructor [u2::Parse $model "private
void $type ();"]
          u2::SetEntity $r "OwnedMember" $constructor

          ## Add instance method
          set method [u2::Parse $model "public static
$type instance(){return m_inst;}"]
          u2::SetEntity $r "OwnedMember" $method
      }
  }
}
```

Using this agent we have in effect implemented a new translation rule. Here is an example of what the result could look like:

**UML**

```
<<singleton>> class C{}
```

**Java**

```
public class C
{
    private static C m_inst = new C();

    private C() {}

    public static C instance()
    {
        return m_C;
    }
}
```

Note that since the agent gets triggered before the ordinary code generator transformations are performed it can use UML constructs in the transformation result. For example, the created attribute is a UML part, which will be transformed to an attribute with a ‘new’ initializer according to the translation rule described in Impact of aggregation kind.

# 44

# *Java Run-time Framework*

This chapter describes the Java run-time framework, Tau Object Run-time (TOR). The purpose and run-time semantics of all classes in the framework is described.

# Introduction

This chapter is a reference guide for Tau Object Run-time (TOR), an object-oriented UML run-time framework implemented in Java. TOR implements the run-time semantics of UML, both for structural and behavioral aspects.

TOR is implemented as a set of classes each providing a well specified service. Some classes are used in the framework itself, while others are used by the code generated with the Java code generator to transform a model to executable Java code.

Some of the classes are used when modeling; these "TOR Classes" are available as a TOR UML Model.

The framework is delivered as a set of Java source files. The files are listed in the List of Files section.

**Note**

*TOR also has a C++ implementation, which is used by code generated by the C++ Application Generator. The design of C++ TOR and Java TOR is very similar, which facilitates the creation of UML models that can target both the Java and the C++ platform. For more information about the C++ implementation of TOR see C++ Run-time Framework.*

## TOR Package

All definitions of TOR are made in a Java package called `com.telelogic.tau.tor`. There are also subpackages in this package where related classes are grouped.

## TOR UML Model

Parts of TOR are available as a model library, called 'tor', that is loaded automatically when the Java code generator is activated. All types and classes that can be used in the user model is included in this model.

The classes of this model are described in the TOR Classes section.

**Note**

*The TOR UML model used for Java code generation is the same one as is used for C++ code generation. However, some parts of that model are not necessary to use when targeting the Java TOR, and are therefore not supported by the Java code generator.*

## Building TOR

TOR is usually built automatically when using the Eclipse Integration for compiling and deploying Java code generated by Tau. Note that if the generated code does not depend on TOR constructs, TOR will not be built.

If you are building the generated Java code in some other way you can find the Java source files for TOR in the Tau installation at

```
addins\JavaApplication\Etc\TOR\com\telelogic\tau\tor
```

Note that you must use a Java SDK for Java 1.5 or later.

For convenience the source files have been prebuilt into a JAR file, tor.jar, located in the same folder.

# TOR Classes

This section contains an alphabetical list of the classes defined in TOR. The purpose and run-time semantics of each class is described. All classes are declared in the TOR Package (or in one of its subpackages).

- CompletedEvent
- Connector
- Dispatchable
- DispatchableClass
- Dispatcher
- DispatcherBehavior
- DispatcherData
- EntryPoint
- Event
- EventExecutor
- EventQueue
- EventReceiver
- ExitPoint
- InstanceManager
- InternalEvent
- Port
- Region
- RunInitialTransition
- State
- StateMachine
- Synthesized
- ThreadedDispatcher
- ThreadSafeEventQueue
- TimerEvent
- TimerObject
- TimerQueue
- TopRegion

## CompletedEvent

An internal Event generated by the framework when a State is finished. It is passed to the state and processed instantly. It is used to trigger any transitions without a triggering event, also known as trigger free transitions.

There exists a null event which is used for transitions with guards only. In comparison, the CompletedEvent triggers transitions without guards, that is transitions with no events and no guards.

## Connector

This class represents a UML connector. It will be instantiated once for each connector in the model. It contains a method ‘connect’ for connecting two ports by the connector. This enables the sending of events between the connected ports.

## Dispatchable

An abstract class representing an entity with the ability to both receive and execute events. The ability to receive events is represented by a realization of the EventReceiver interface, and the ability to execute events is represented by a realization of the EventExecutor interface.

A Dispatchable is associated with a Dispatcher and an EventQueue. When the dispatcher processes an Event from a queue, it asks the dispatchable to which the event is addressed to process the event. As a result, the Dispatchable returns a EventExecutor.EventAction indicating how the event is handled.

## DispatchableClass

A dispatchable class represents a UML active class. It inherits Dispatchable to get the capability of receiving and executing events that are dispatched to it.

A dispatchable class may have an associated StateMachine, known as the classifier behavior statemachine.

Events sent to an instance of a dispatchable class are passed on to the StateMachine of the instance.

If a UML class is made active, an inheritance to `DispatchableClass` will be added automatically in the model. This makes it possible to access important operations from DispatchableClass in the UML model.

### Instantiation of dispatchable classes

When an instance of a dispatchable class is created, it does not automatically start the execution of its behavior. It must first be initialized, after which it can be started.

During initialization, the classifier behavior statemachine of the dispatchable class gets initialized and prepared for execution. The initialization is normally done automatically by the code generated with the Java code generator in the `main` method. To manually initialize a dispatchable class, call the `init` operation:

```
public void init();
```

Starting an instance starts its classifier behavior state machine by posting a request (RunInitialTransition) for executing the initial transition. Once started, it is ready to receive events. Starting is often performed automatically by the code generated with the Java code generator in the `main` method. To manually start the statemachine call the `start` operation:

```
public void start();
```

Starting an already started instance has no effect.

**Note**

*Beware that "start" is a reserved word in UML textual syntax. It is therefore necessary to write it with single quotation marks ('start') when referring to the method above.*

## Dispatcher

A dispatcher is associated with an EventQueue and is responsible for processing Events placed in the queue. Events can be retrieved from the queue and processed one by one, or continuously as long as the queue is not empty.

A dispatcher is also associated with a TimerQueue where TimerObjects corresponding to currently active timers are located.

The code associating a dispatcher with a Dispatchable and then starting the dispatcher is normally generated automatically with the Java code generator in the `main` method, but it can also be done manually as described below.

To add a dispatchable to the dispatcher, call the `add` operation:

```
public void add(Dispatchable);
```

To remove a dispatchable from the dispatcher, call the `remove` operation:

```
public boolean remove(Dispatchable);
```

To start processing events from the event queue of the dispatcher, call the `run` operation. It retrieves events one by one from the event queue until the queue is empty, then it returns.

```
public void run();
```

To process events off the queue one by one, call the step operation. It retrieves and processes the next event in the queue.

```
public void step();
```

## DispatcherBehavior

A class representing the behavior of a ThreadedDispatcher, realizing a thread which waits for events to arrive to the event queue and uses the dispatcher to execute them. DispatcherBehavior is defined as a static nested class of ThreadedDispatcher.

## DispatcherData

A class containing the data needed by a ThreadedDispatcher (and defined as a static nested class of that class). The data is encapsulated in a DispatcherData in order to be safely accessible both by the thread that is launched by the ThreadedDispatcher and the calling thread.

## EntryPoint

An entry point is a named pseudo state used to enter a Region of a composite State or a StateMachine. It provides a means to enter the composite state or sub-state machine without revealing anything about its internals, for example the actual target state.

When an entry point is reached, i.e. a transition with an entry point as its target has been executed, the region owning the entry point is entered, and then the outgoing transition of the entry point is executed.

An entry point can have any number of incoming transitions, but only one outgoing transition.

## Event

The Event class is used to represent types and occurrences of all kinds of events in a system, for example; signals, asynchronous operation calls and timer time-outs. Subclasses of Event specifies event types and instances of these classes represent event occurrences.

Every event has a receiver that is represented by an object id. The InstanceManager is responsible for mapping such an object id to a pointer to a Dispatchable that is the real receiver of an event instance. The receiver is set internally by the framework.

## EventExecutor

This is an interface capturing the ability to execute events. The interface contains the following method that must be implemented by classes that implement EventExecutor:

```
public EventAction execute(Event event);
```

Implementations of this method should return an appropriate literal of the EventExecutor.EventAction enumeration to indicate how the event was handled by the EventExecutor.

Note that before an event can be executed, it must be received. Thus it is normally so that a class that implements the EventExecutor interface also implements the EventReceiver interface.

**Note**

*It is important to make a distinction between receiving and executing an event. An event is **received** when it is delivered by the framework to the receiver. An event is **executed** when behavior that is associated with the reception of the event is executed (typically a transition). Event reception must precede event execution, and there is typically some time interval between reception and execution.*

### EventExecutor.EventAction

An enumeration used to indicate how an event is handled by the receiver. The literals and their meaning are listed in the table below.

| Literal | Description |
|---|---|
| `NoMatch` | The event is not handled by the receiver. |
| `Defer` | The event is saved by the receiver. |
| `Consumed` | The event is consumed by the receiver. |

## EventQueue

An event queue is a queue of Events, implemented in Java using a LinkedList.

The framework automatically handles everything related to event queues, i.e. insertion and removal of events.

## EventReceiver

This is an interface capturing the ability to receive events. The interface contains the following method that must be implemented by classes that implement EventReceiver:

```
public boolean receive(Event e);
```

Implementations of this method should return true if the event was received, and false otherwise.

EventReceiver is also available in the TOR UML profile as an interface. This makes it possible to declare your own event receivers. Events can be sent to such event receivers by means of the sendTo utility function. One common use for this is when there are passive classes that need to receive signals. By letting the classes inherit the EventReceiver interface, and implementing the `receive` function, this becomes possible.

For information about the difference between receiving and executing an event see the note in EventExecutor.

## ExitPoint

An exit point is a named pseudo state used to leave a Region of a composite State or a StateMachine. It provides a means to leave the composite state or sub-state machine without knowing anything about the context in which it is instantiated, for example the target state of the outgoing transition.

When an exit point is reached, i.e. a transition with an exit point as its target has been executed, the exited region (owning the exit point) is left, and then the outgoing transition of the exit point is executed.

An exit point can have any number of incoming transitions, but only one outgoing transition.

## InstanceManager

This class is responsible for keeping maps between object ids and EventExecutors and EventReceivers. The maps are kept up-to-date when Dispatchables are created and deleted (by the garbage collector). The main use for the instance manager is to have a symbolic representation (an object id) of an instance, rather than a direct reference to the instance. This indirection is crucial in a multi-threaded system where instances are created and deleted independently from multiple threads. It is also necessary in a system that is distributed over multiple address spaces.

## InternalEvent

This class is a common base class for all events that are sent internally by the TOR framework, i.e. all events that do not correspond directly to user-defined events in the model.

## Port

This class represents a UML port. It is instantiated in generated code once for each enabled direction of the UML port. That is, if signals may flow both in to and out from the port, two attributes typed by this class will be generated.

Each port maintains a list of other ports to which it has been connected by means of Connectors.

## Region

The abstract Region class represents an orthogonal region of a State or a StateMachine. A region owns a set of states. A region keeps track of its current and previous states. The current state is the currently active state of the region. The previous state is the state that was active the last time the region was left.

A region can be entered and left as the result of a transition.

### Entering a region

A region can be entered in a number of different ways. The first time a region is entered, its initial transition is executed. Subsequently, when the region is entered, history information can be used to re-enter the previous state. A region can also be entered through an EntryPoint.

The current and previous states are updated whenever a state in the region is entered as a result of a transition.

### Leaving a region

A region is left when a transition with a target state in a different region is triggered, or when a transition with an ExitPoint as its target is triggered.

The current and previous states are updated whenever a state in the region is left.

### Finishing a region

A region is finished when a final state of the region is reached. Final states are not explicitly defined in the framework.

When a region is finished, trigger free transitions of the enclosing state(s) are evaluated.

## RunInitialTransition

An internal event that is sent to a DispatchableClass when it is started. When this event is executed, the initial transition of the class will be executed.

RunInitialTransition is defined as a static nested class of TopRegion.

## State

This is an abstract class that represents a state in a StateMachine. Generated classes corresponding to states in the user model inherit from this class.

A state is owned by a Region. A state can own a number of regions or a state machine, in which case it is referred to as a composite state.

### Transitions and event handling

A state has a set of incoming and outgoing transitions. Transitions are not represented by separate classes in the framework, instead they are represented by methods in the enclosing state machine.

When a state receives an event, it checks if there are any outgoing transitions triggered by the event. If there is a match and the guard expression is true, the transition is executed. Transitions are not ordered, the first matching one found by the framework will be executed.

If no match is found, the event is passed on to any parent of the state, i.e. another state or a state machine.

### Entering a state

A state is entered when a transition with the state as a target has been executed. When a state is entered any entry actions of the state are executed. Then if the state is a composite state, any owned regions or state machine is also entered.

The current and previous state of the owning region is updated when a state is entered.

### Leaving a state

A state is left when any of its outgoing transitions is triggered, or when any outgoing transition of a parent to the state is triggered. When a state is left any exit actions of the state are executed.

The current and previous state of the owning region is updated when a state is left.

## StateMachine

This is an abstract class that represents a state machine and its implementation. Generated classes corresponding to state machines in the user model inherit from this class.

**Note**

*A state machine as a list of top regions. Currently this list will always contain exactly one TopRegion which is defined in the generated subclass of StateMachine.*

A state machine can be associated with a DispatchableClass, as the classifier behavior of the class. An Event sent to the dispatchable class is passed on to the state machine and processed there.

A state machine can be owned by a State, in which case it is called a sub-state machine. Events sent to the state are passed on to the sub-state machine for processing.

Before a state machine is ready to process events it has to be initialized and started. This is done internally by the framework when the corresponding actions are preformed on the associated DispatchableClass or State.

## Synthesized

This is an annotation used for marking definitions that are synthesized during Java code generation. Changes to synthesized definitions are not considered when updating the UML model from changes made to generated Java files.

The Synthesized annotation also makes generated code more readable, since it becomes clear which parts of generated Java files that have a direct correspondance to UML model entities, and which parts that were added by the code generator.

## ThreadedDispatcher

This class is a threaded version of a Dispatcher. ThreadedDispatcher realizes a thread which will dispatch events to all Dispatchables that are added to it. This allows for flexible thread deployment of a model. Some examples:

- One thread per active class instance
  Each active class is then associated with exactly one ThreadedDispatcher. This can for example be modeled by letting each active class contain one ThreadedDispatcher as a part. In the constructor of the class, the newly created instance is added to the ThreadedDispatcher, and it is launched. The lifetime of the launched thread is tied to the lifetime of the ThreadedDispatcher, so when an instance of the active class is deleted, the thread will be ended.
- One thread per active class
  This can for example be modeled by adding a static ThreadedDispatcher attribute to the class. In the constructor of the class, the newly created instance is added to the ThreadedDispatcher.

- One thread for an arbitrary set of instances
  This can for example be modeled by adding an ThreadedDispatcher attribute in a singleton static class, and then add the instances it shall dispatch to it.

A ThreadedDispatcher owns a DispatcherData which comprises all data that needs to be accessed by both the launched thread and the caller thread. For example, the dispatcher data contains a Dispatcher and a ThreadSafeEventQueue. The encapsulation of the data in a DispatcherData instance ensures that the data can be accessed in a thread-safe manner.

The dispatcher and the event queue are automatically instantiated and associated. The dispatcher will process events off the event queue in a thread-safe manner.

To start a ThreadedDispatcher call

```
public void launch();
```

To stop a ThreadedDispatcher call

```
public boolean endThread();
```

This method will end the ThreadedDispatcher as soon as possible, but without interrupting the behavior triggered by the currently executed event. The method is synchronous, i.e. it will wait until the behavior triggered by the current event has run to completion. There is also an overloaded version of this method that allows for specifying a timeout value, to avoid waiting too long for the thread to end. If the thread has not been ended after the specified time has elapsed, the method will return false.

## ThreadSafeEventQueue

A thread-safe subclass of EventQueue used by ThreadedDispatcher. All operations performed on the queue are synchronized and a Semaphore is used for waiting on new events to arrive to the queue.

## TimerEvent

A timer event is a special kind of Event, representing a timer timeout event. It is associated with a TimerObject, on which timer actions such as `set` and `reset` can be performed.

### TimerObject

A timer object represents the declaration of a timer in a Dispatchable. It provides methods for setting and resetting a timer and for querying a timer whether it is currently active or not. The implementation of these methods uses an associated TimerEvent and TimerQueue to realize the timer semantics.

A timer object also holds the timeout time value when the timer is active.

### TimerQueue

A timer queue is a priority queue of TimerObjects corresponding to timers that are currently active. The timer objects are ordered with regards to their timeout times.

Every Dispatcher has a timer queue in which timer objects corresponding to active timers of managed Dispatchables are administered.

### TopRegion

A top region is a Region owned by a StateMachine. The owning state machine can be retrieved from a top region.

When a top region is finished the owning state machine is also finished if all its top regions are finished.

TopRegion is a subclass of Region.

## Utilities

There are a number of utility methods that are frequently used internally by the TOR framework and by the generated Java code. They can also be used by the user since they are defined in the TOR UML library. All utilities are declared as static methods of an abstract class Utilities.

### sendTo

A method used to send an Event to a receiver. The receiver can be specified as an EventReceiver:

```
public static boolean sendTo(Event e, EventReceiver r);
```

There is also an overloaded version which allows the event to be sent to a port, so that the receiver does not need to be explicitly specified, but instead can be located at run-time using the connector structure:

```
public static boolean sendTo(Event e, Port p);
```

The event is sent to the receiver for processing. A boolean value is returned to indicate if the event was received by the receiver or if it was lost.

In either case, the responsibility of the event is passed on to the receiver and the event should not be accessed after passing it to the `sendTo` method.

## setTimeUnit

A method used for setting the unit of time to be used by TOR.

```
public static void setTimeUnit(double seconds);
```

The time unit is specified in seconds. For example, to specify a time unit of 1 millisecond, call `setTimeUnit(0.001)`.

**Important!**

*At present the time unit is shared throughout the entire application. Do not change time unit from different threads and when timers already have been activated, to avoid unexpected results.*

# Operating System Abstraction Layer

TOR uses a separate abstraction layer to interface with functionality that traditionally is supplied by the underlying operating system, but which in the case of the Java platform is available in the form of Java library functionality. This section describes the classes defined in this layer.

All classes in the OS layer are defined in a package called `os`, which is a subpackage contained in the `TOR Package`. The fully qualified name is therefore `com.telelogic.tau.tor.os`.

Some of the classes in the OS layer are available in the TOR UML Model. For example, it is often necessary to make use of the thread synchronization primitives of TOR, such as a semaphore.

### Semaphore

A semaphore is used to access and wait for resources accessed by more than one thread. A semaphore can be retrieved and released. There are two ways of getting a semaphore, either wait until the semaphore is released so you can get it, or wait for a specified amount of time before moving on.

### Time

This class is a representation of time (both absolute time and relative time durations). The class contains various functions for getting the current time, adding and subtracting time values etc. The underlying Java representation is a `Date` object.

## Thread

This is a subpackage of the `com.telelogic.tau.tor.os` package. It contains classes and utilities for working with multiple threads in a generated Java application.

### Behaviour

A thread is represented by means of a Behaviour class which inherits from `java.lang.Thread`. The ThreadedDispatcher class uses a `Behaviour` subclass (DispatcherBehavior) in which the dispatcher is executed.

Just like in the `com.telelogic.tau.tor.os` package there is a static `Utilities` class which contains thread related utilities.

**suspend**

```
public static void suspend(Time timeout);
```

Suspends the current thread until the specified point in time (N.B. 'timeout' is an absolute time value).

**getCurrentThreadId**

```
public static long getCurrentThreadId();
```

Returns the ID of the currently executing thread.

# List of Files

This section describes the files of TOR delivered as a part of the Tau installation.

### Java source files

The complete TOR source code is included in the Tau installation and can be found in the following folder in the installation directory:

```
addins\JavaApplication\Etc\TOR\com\telelogic\tau\tor
```

These files are used when building generated Java applications which depend on TOR. See Building TOR for more information about how to build TOR.

The table below lists all the files and the TOR declarations they contain.

| File | TOR declaration |
|---|---|
| CompletedEvent.java | CompletedEvent |
| Connector.java | Connector |
| Dispatchable.java | Dispatchable |
| DispatchableClass.java | DispatchableClass |
| Dispatcher.java | Dispatcher |
| EntryPoint.java | EntryPoint |
| Event.java | Event |
| EventExecutor.java | EventExecutor |
| EventQueue.java | EventQueue |
| EventReceiver.java | EventReceiver |
| ExitPoint.java | ExitPoint |
| InstanceManager.java | InstanceManager |
| InternalEvent.java | InternalEvent |
| Port | Port |
| Region.java | Region |
| State.java | State |

| File | TOR declaration |
|---|---|
| StateMachine.java | StateMachine |
| Synthesized.java | Synthesized |
| ThreadedDispatcher.java | ThreadedDispatcher, DispatcherBehavior, DispatcherData |
| ThreadSafeEventQueue.java | ThreadSafeEventQueue |
| TimerEvent.java | TimerEvent |
| TimerObject.java | TimerObject |
| TimerQueue.java | TimerQueue |
| TopRegion.java | TopRegion |
| Utilities.java | sendTo, setTimeUnit |
| os/Semaphore.java | Semaphore |
| os/Status.java | Status enumeration used as return type of various OS-related methods. |
| os/Time.java | Time |
| os/thread/Behaviour.java | Behaviour |
| os/thread/Utilities.java | suspend, getCurrentThreadId |

# 45

# *Eclipse Integration*

The Tau Eclipse Integration extends the UML tool set with Java support and provides an integration to the Eclipse Java IDE. The integration consists of a number of commands added to the UML tool set and to Eclipse to facilitate seamless round-trip engineering for Java.

Eclipse projects can be created from existing UML projects and projects in Eclipse can be imported into UML. Once a project is present in both tools it is kept synchronized. The integration also facilitates navigation between the tools.

The integration is built on top of the available Java Support.

# Installing the Eclipse Integration

### Eclipse integration components

The Eclipse integration consists of two components:

- An add-in called EclipseIntegration
- An Eclipse plug-in called TauG2Integration

Both components have to be correctly installed in order for the integration to work properly. The EclipseIntegration add-in is automatically installed by the Tau installer, but the Eclipse plug-in is not.

### Eclipse integration plug-in

To install the TauG2Integration plug-in into Eclipse:

1. Make sure that Eclipse is properly installed and not running.
2. Locate the plug-in installation file:

   ```
   C:\Program Files\Telelogic\TAU_4.2\integrations\Eclipse
   com.telelogic.taug2integration_4.2.0.jar
   ```

3. Copy the `.jar` file to the "`plugins`" folder of the Eclipse installation.
4. The plug-in will be running the next time Eclipse is started.

**Note**

*The **EclipseIntegration** and the **JavaApplication** Add-Ins are not designed to be used both at the same time. Some of the commands are identical, while some exist only in one of the addins. Finally, some commands have a slightly different implementation. To avoid confusion, only one of these add-ins should be activated for any project.*

### Activating the integration in Tau

The EclipseIntegration add-in must be activated for every project you want to use the Eclipse integration in. To activate it:

1. From the **Tools** menu select Customize.
2. Click the Add-Ins tab and check the **EclipseIntegration** add-in.
3. Click **OK**.

**Note**

*When activating the add-in, or loading a project with the add-in activated, the entire Java Runtime Libraries might be loaded as a profile (depending on the setting of the option Load rt.jar on startup. Since this profile is large this operation may take some time. When loading the profile Tau will not respond to any interaction.*

# Working with Eclipse

## Workflow scenarios

The Eclipse integration is primarily intended to be used in two different scenarios, depending on if the starting point is java code or UML models:

- Starting with UML. You start by creating a UML model inside Eclipse using the Eclipse New wizard (compare section Create a UML project in Eclipse). The UML modeling constructs are used to do an analysis and design model. When you consider the design to be sufficiently stable you generate the corresponding Java code from this model and continue with the implementation keeping the Java code and UML model in synch. (compare section Create a Java project in Eclipse).
- Starting with Java. You have an existing Eclipse Java project (or create a new project) and you want to create a corresponding UML model to visualize and analyze the static structure of the application. This is done as described in section Create a project in Tau. When the UML model is created it will be kept in synch with the Java code as described in section Model and Code Synchronization.

A variant of the first scenario is to not store the UML model inside the Eclipse workspace. This is also supported.

## Create a UML project in Eclipse

To create a new UML project in Eclipse:

1. Select the File->New->Project command in Eclipse.
2. Select UML project (found in the Tau folder) as the project type and click “Next”
3. Select “Java Code Generation” or some other relevant kind of UML project in the second page of the wizard.

4. The third page in the wizard gives you a possibility to change details of the project. In most cases the default values do not need to be changed.
   - Select to add a UML file to the project
   - Provide a name for the UML file
   - Choose the location of the project
   - Decide to create an empty top-level UML package

Ones the project is created you can start the editing of the UML model by double-clicking on the ttp file in the "Package Explorer"

## Create a Java project in Eclipse

To generate Java code and a corresponding Eclipse project from an existing UML model:

1. Open the project in Tau and activate the EclipseIntegration add-in on the project.

**Note**

*When the Java Runtime Libraries are loaded this may take several minutes.*

2. From the **Eclipse** menu, select Create Eclipse Project.

This generates Java code from the entire project, opens and connects to Eclipse, creates a corresponding Eclipse project and sets up up a link between the projects. The projects are now setup for seamless round-trip engineering. For details on the available Eclipse related commands, see "Tau to Eclipse" on page 1474.

Whenever you execute the Update Source Code command from the UML tool set, the Eclipse project is synchronized if Eclipse is running. The Synchronize with Eclipse command updates the source code, but it also attempts to start Eclipse with the corresponding Eclipse project if Eclipse is not already running. See also section Model and Code Synchronization for more synchronization options.

The Locate in Eclipse command is used to navigate from a model element to the Java source code in Eclipse.

## Create a project in Tau

To generate a UML model from your Java code in Eclipse:

1. Open the project in Eclipse and select the **Package Explorer**.
2. From the **Tau** menu, select Create Corresponding Tau Project. In the dialog that appears you can select the location of the Tau project if needed (the default location is the project directory in the Eclipse workspace). You can also choose to automatically generate diagrams in the created UML.

The command will the open and connect to Tau, create a new empty project and reverse-engineer the Java source code into a UML model. It also sets up a link between the projects.

The projects are now setup for seamless round-trip engineering. For details on the available Tau related commands, see "Eclipse to Tau" on page 1477.

**Note**

*The loading of the new Tau project may take several minutes since the Java Runtime Libraries are loaded into Tau.*

To synchronize changes made in the Java code in Eclipse, use the Synchronize with Tau command. All new classes and files are detected and added to the UML model and changes are merged into the model. See also section Model and Code Synchronization for more synchronization options.

The Locate in Tau command is used to navigate from the Java source code to the corresponding model element.

## Model and Code Synchronization

The java source code and the UML model are kept synchronized. You can control when the synchronization is done by changing java settings for the UML model. See section Synchronizing Model and Source Code for more details.

# Tau to Eclipse

## Communication

All commands communicating with Eclipse require that the location of the Eclipse executable has been stored by the EclipseIntegration addin. The first time any of the commands are executed you will be prompted to specify where the Eclipse executable is located. This information is then stored as Eclipse location in the Eclipse Options, and can be changed with the Change Eclipse directory command.

**Note**

*When establishing the communication between Tau and Eclipse temporary files may need to be generated. On Unix it is therefore important that the TEMP environment variable has been set up.*

## Connecting to Eclipse

Tau communicates with Eclipse through a socket, and almost all commands require that the Eclipse executable is running. If Eclipse is not running, Tau automatically attempts to start Eclipse. If this fails a dialog is displayed after a time-out. You can then select to try to connect again or abort the command.

**Hint**

*If the connection fails, make sure that the TauG2Integration plug-in has been properly installed.*

## Commands in Tau

### Importing JAR files

JAR files can be imported into the UML project. This is accomplished using the Import Wizard. See section Importing JAR Files for details.

### Export

Java files can be generated from a UML model in the UML tool set, no matter how the model was created to begin with. Java files are generated when you export a package to Java source code. From the **Eclipse** menu select **Export** and point to **Package** in the submenu.

### Update Model

Once the Java files have been exported the model is kept synchronized by using the **Update Model** command. The Locate in Eclipse command is used to open the `.java` file in Eclipse if the project is connected to an Eclipse project.

The command will do an incremental update of the model based on changes in the java files.

### Force Update Model

The command will update the model by re-reading all java files and updating the model correspondingly.

### Update Source Code

Once the Java files have been exported the source code is kept synchronized by using the **Update Source Code** command. The Locate in Eclipse command is used to open the `.java` file in Eclipse if the project is connected to an Eclipse project.

- Select **Update Source Code** from the **Eclipse** menu.

This updates the Java files for all Java packages in the model.

### Force Update Source Code

The command will update the source code by re-generating all java files.

### Change Eclipse directory

Tau connects to Eclipse by launching the Eclipse executable. If the executable is moved or there are multiple installations of Eclipse, the wanted executable can be chosen here.

- Select **Change Eclipse directory** from the **Eclipse** menu.

### Create Eclipse Project

To create an Eclipse project from an existing UML model:

1. Select **Create Eclipse Project** from the **Eclipse** menu.

This starts up Eclipse and creates a project with the same name as the active project in Tau. The location is determined by the current Eclipse workspace location.

Java source code is generated for all packages in the active project. The default project source code location of Eclipse is used. (This is the same folder as the Eclipse project by default.) Java syntax is also enabled for all the packages.

The location of the source code is added to the list of Synchronized Target Directory. This ensures that changes made in the file system are automatically reflected in the model when updating the model from source code.

### Synchronize with Eclipse

To synchronize a project that has been exported to Eclipse, or created from Eclipse:

- Select **Synchronize with Eclipse** from the **Eclipse** menu.

This updates the Java files for all Java packages in the model and refreshes the corresponding Eclipse project.

This command is only available if a connection exists between the Tau project and an Eclipse project. See Eclipse integration components.

### Locate in Eclipse

To locate a model element in the corresponding source code in Eclipse:

1. Select a single element in the Model View or in a diagram.
2. Select **Locate in Eclipse** from the **Eclipse** menu.

This selects the element in Eclipse. If the element is a package, the package is selected in the Package Explorer otherwise the corresponding Java source file is opened and the element is selected in the editor, as well as in the Package Explorer and in the Outline View.

### Use Java Syntax

The Java support extends the tool with Java Syntax when this selection is checked.

# Eclipse to Tau

All commands require that the Tau installation has been stored in the Eclipse workspace. The first time any of the commands are executed you will be prompted to specify where the Tau installation is located. This information is then stored in Eclipse, and you should not have to specify it again. If you need to change it for some reason, you can Set Tau Location.

**Note**

*When establishing the communication between Eclipse and Tau temporary files may need to be generated. On Unix it is therefore important that the TEMP environment variable has been set up.*

## Eclipse command list

### Set Tau Location

To set or change the location of the Tau installation:

1. Select **Set Tau Location...** from the **Tau** menu.
2. Select the Tau installation folder, typically:
   `C:\Program Files\Telelogic\TAU_4.2`
3. Click **OK.**

### Create Corresponding Tau Project

To create a Tau project from a project in Eclipse:

1. Select **Create corresponsing Tau Project...** from the **Tau** menu.
2. Select a folder for the Tau project files.
3. Click **OK**.

A new project is created and loaded into Tau. One `.u2` file is created and all source files in the Eclipse project are synchronized into the model.

The location of the source code is added to the list of Synchronized Target Directory in Tau. This ensures that changes made in the file system are automatically reflected in the model when updating the model from source code.

### Synchronize with Tau

To synchronize a project that has been exported to Tau, or created in Tau and then exported to Eclipse:

1. Select **Synchronize with Tau** from the **Tau** menu.

This updates the model corresponding to all Java files of the project. New packages, for example classes or Java files, added in Eclipse are added to the model. Changes to existing elements are detected and merged into the model. Deleted elements are deleted from the model.

For details on the file to model mapping, see "Synchronizing Model and Source Code" on page 1341.

### Locate in Tau

To locate a model element in Tau from Eclipse:

1. Select an element in the Package Explorer, the Outline View or in the Java editor.
2. Select **Locate in Tau** from the **Tau** menu.

This starts Tau and navigates to the corresponding model element using the Model Navigator. If the model element has one and only one corresponding presentation element, the diagram containing this presentation element is displayed.

# Eclipse Options

Eclipse related options are saved in a textual format in a file called:

```
eclipseOptions.ini
```

located in the `EclipseIntegration` addin folder, typically found in:

```
C:\Program Files\Telelogic\TAU_4.2\addins\EclipseIntegration
```

This file is created on demand the first time an option is set by the integration. Each options is stored in key/value pairs separated with a '=' character. Options are separated with new line characters. Example:

```
key1=value1
key2=value2
```

To change an option, open the file in a text editor and change the value. If the key does not yet exist, add a new line with both the key and the value.

**Note**

*Always make a backup copy of the options file before editing it manually.*

Currently there are two options:

- Eclipse location
- Platform options

### Eclipse location

Specifies the location of the Eclipse executable. This option is set when Eclipse is launched from Tau for the first time.

It is stored under the key `ExecutablePath` and points to the executable file of Eclipse, typically:

```
ExecutablePath=C:\Program Files\eclipse\eclipse
```

### Platform options

Specifies the platform options used when launching Eclipse. Normally no platform options are passed to Eclipse by the integration. If you wish to launch Eclipse with platform options, for example to use a different workspace than the default, you have to set this option.

It is stored under the key `PlatformOptions`, for example:

```
PlatformOptions=-data c:/temp
```

The value is passed unchanged to Eclipse as platform options. The value should therefore be specified as when starting Eclipse manually.

**Note**

*Forward slashes must be used in any paths specified in the PlatformOptions value.*

# *UML and C#*

The chapters that are listed under UML and C# describe how to generate a C# application from a UML model, and how to import existing C# code into Tau.

The C# support in Tau is only supported for Windows operating systems.

# 47

# *C# support*

This section describes how to use the C# support in Tau. It extends Tau with abilities to import an existing C# application into UML, and to generate a C# application from the UML model. It also contains support for keeping the model and code synchronized when any of these change.

It is assumed that a separate development environment is used for compiling, debugging and running the generated C# application. An integration with Microsoft Visual Studio is provided with the Tau installation as described in Visual Studio .NET Integration, but the general-purpose C# support described in this chapter can be used together with any C# development environment.

# Using the C# Support

## Creating a C# Project

A Tau C# project can either be created as a new project from scratch, or an existing Tau project can be turned into a C# project.

To create a new C# project use the following procedure:

1. Choose the command **File->New**...
2. Choose **UML for C# Code Generation** as the project type when creating the project
3. Complete the wizard using desired settings.

For details on how to create a project, see "Working with Projects" on page 35 in Chapter 4, *Introduction to Tau 4.2*

To activate the C# support for an existing project do the following:

1. From the **Tools** menu select Customize.
2. Click the Add-Ins tab and check the CSharpApplication add-in.
3. Click **Close**.

Regardless of how the C# project was created you should now see some new C# Specific Libraries in the Model View under the Library node. There is also a new C# Menu added to the menu bar.

## C# Specific Libraries

The following C# specific libraries are available in a C# project:

### TTDCSharp

This library contains all stereotypes that are used with the C# support. Tagged values for these stereotypes constitute various options to the C# tools. The library also exposes an API to the C# related tool functionality by means of agents. These agents can be used from the public APIs in order to invoke C# related commands programmatically.

### TTDCSharpPredefined

This library contains a number of stereotypes that are used to represent C# specific language constructs that cannot be expressed in terms of standard UML constructs.

For example, in standard UML the concept of partial classes does not exist. A class must be fully defined in one and only one location. However, in C# a class may be defined partially in multiple locations using the `partial` keyword. In order to represent partial types the TTDCSharpPredefined library contains a <<partial>> stereotype.

The library also contains UML representations of built-in C# types, such as string, int and bool.

### TTDCSharpRuntime

This library is a UML representation of the Microsoft.NET main assembly known as 'mscorlib'. It shows up as two library packages called "System" and "Microsoft". It contains a number of fundamental definitions used extensively in almost all C# programs, and is therefore available as a pre-imported library. All other assemblies that you want to use from your C# application should be imported by means of the .NET Assembly Importer to become available for use.

## C# Menu

The C# menu contains commands that allow you to generate C# code from the UML model. The commands also help you keep the UML model and the generated C# code synchronized when one of these has been changed.

All commands in this menu operate globally on everything that is present in the user model (shown below the Model folder in the Model View). In order to apply any of these commands selectively use the corresponding command in the context menu of the selected element.

### Update model

This command updates the UML model with changes made in generated C# files. This command is only meaningful if C# source files have already been generated for the model (see Generating C# Code for more information). Only files that have been changed since the last time they were generated from the model will be processed.

Note that this command shall not be used to import existing C# code that has not been generated from the model. In order to import such code use the C# importer described in Importing Existing C# Code.

### Force update model

This command is identical with Update model with the only difference that all generated C# source files will be processed, also those that have not been changed since the last time they were generated from the model. If changes have been made to C# files without modifying their timestamp the Update model command will not detect that these files have changed. In that case you can use the Force update model command instead.

### Update source code

This command updates generated C# files with changes made in the UML model. If no C# files have been generated, they will be generated as described in Generating C# Code. Only those files will be updated that need to be so based on changes that have been made in the model since the last time these files were updated or generated.

### Force update source code

This command is identical with Update source code with the only difference that all C# source files will be updated or generated. This command is useful if changes have been made to UML model elements without modifying their timestamps. This can happen due to limitations in the model change detection mechanism used by Tau.

# Generating C# Code

C# source code is generated from the UML model using the Update source code or Force update source code commands. These commands generate C# source files based on a model-to-file mapping specification.

## Model-to-File Mapping

A model-to-file mapping specification specifies how to map model elements to C# source files. Such a specification consists of C# file artifacts (defining which C# files to generate) and <<manifest>> dependencies from these artifacts to definitions in the model (defining which definitions to generate into each file).

It is recommended to place the model-to-file mapping specification in a separate package in the model, and to save this package in its own file. Thereby you get a clear separation between the model itself, which defines the logics and behavior of the C# application, and the model-to-file mapping, which specifies which C# source files the application is made up of.

Following the above recommendation leads to a UML model with the following structure:

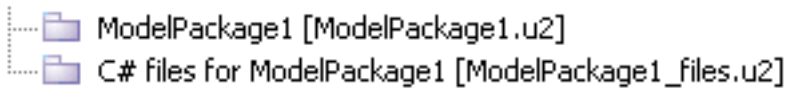



*Figure 247: Structure of a UML model for C# code generation*

There may be any number of top-level model packages in the model. Each top-level model package corresponds to a C# software component, such as an assembly or executable. For each top-level model package there is one package containing the model-to-file mapping specification for that model package.

**Note**

*During code generation the C# code generator adds information to the model-to-file mapping package. For example, it updates timestamps on the C# file artifacts and adds information that makes it possible to navigate between the model and the generated C# code. It is thus required that the model-to-file mapping package is writable. By saving the model-to-file mapping package in its own file code generation can be performed without modifying the file that contains the model package.*

### Default model-to-file mapping

If no user-defined model-to-file mapping exists the C# code generator will first generate a default model-to-file mapping prior to code generation. It will be placed in a separate package, next to the corresponding model package. The model-to-file mapping package is generated according to the following rules:

1. Each definition contained in a package, with the exception of delegates, are generated into its own C# file.
2. The name of a C# file is the fully qualified name of the definition it contains.
3. Delegates are generated into the same C# file as the definition in which it is defined.

**Example 472: Default model-to-file mapping**

**UML**

```
package P {
  class Class1 {
    void Operation1( long p1);
    delegate void Delegate1( long a);
  }
  enum DataType1 {
    Literal1
  }
}
```

**C#** (in file `"P.Class1.cs"`)

```
namespace P {
  class Class1
  {
    void Operation1(long p1)
    {
    }

    delegate void Delegate1(long a);
  }
}
```

**C#** (in file `"P.DataType1.cs"`)

```
namespace P {
  enum DataType1
  {
    Literal1
  }
```

```
}
```

If a partial model-to-file mapping exists, but more definitions have been added later on in the model, the existing model-to-file mapping package will be updated with a file mapping for the missing definitions.

### Location of generated files

The model-to-file mapping also defines the location of generated files in the file system. By default the files will be generated in the same folder as the .u2 file in which the corresponding UML definition is located.

To generate one particular file in a different location you may modify the `path` tagged value of the file artifact.

To generate all files in a different location you may apply the stereotype `<<cSharpCodePath>>` ("C# code path") and specify a folder in its `path` tagged value. This stereotype can either be applied on the model-to-file mapping package, or on the Model node. In the former case all relative paths of file artifacts contained in that model-to-file mapping package will then be interpreted as relative to that path when generating the C# files. In the latter case the option applies to all model-to-file mapping packages, unless any of them has the `<<cSharpCodePath>>` applied.

## Navigating to and from Generated C# Files

To open a generated C# file you may double-click on the file artifact that represents it in the model. You may also use the **Goto Source** command that is available in the context menu of any generated definition, to navigate directly to the location (or locations - in some cases one UML definition can end up in multiple C# files) of that definition in the generated C# files.

When navigating to a generated C# file it will be opened in Tau's built-in text editor (or in Visual Studio if the Visual Studio Integration has been installed and activated).

It is also possible to navigate in the opposite direction, from the C# code that was generated, to the corresponding UML model entity. This is done by a specific command in the Visual Studio Integration, and by the **Goto Source** command that is available in the context menu of Tau's built-in text editor.

### Translation Rules

When generating C# code from a UML model, translation rules are used that define the mapping for each kind of UML entity to C# code. These translation rules are described in UML to C# Mapping. Note that certain UML entities have no defined mapping to C#. Such entities can currently not be translated to C# and will be ignored by the C# code generator if encountered.

### Compiling, Running and Debugging Generated C# Code

It is assumed that a separate development environment is used for compiling, running and debugging the generated C# application. Although any C# development environment can be used, the C# support is primarily designed to be used with Microsoft Visual Studio. A special integration with this development environment is available, which facilitates the deployment of generated C# code considerably (see the chapter about the Visual Studio Integration).

## Importing Existing C# Code

This section describes how to import existing C# source code into a UML model. Note that if the C# code you want to import has been built as an assembly, you may consider to import that assembly into Tau instead of the source code. Doing so allows you to look at the imported C# module as a black box from your UML model, which can be visualized and accessed. See .NET Assembly Importer for more information. However, if the purpose of the import is to reverse engineer or analyze the complete C# module in UML, and you want to import also the implementation of C# methods, then you should use the C# code importer described in this section.

### Using the C# Import Wizard

Existing C# code is imported into Tau using the C# Import Wizard. Follow these steps:

1. Select the Model node in the Model View and perform the command **File->Import...**
2. Select **Import C#** in the Import dialog and click OK.

3. Select whether you want to do the import based on a Visual Studio project (.csproj file) or if you want to specify individual C# files (or folders containing C# files). If you choose to import a Visual Studio project you also have to specify which project configuration to use, in case there are more than one in the project. All C# files that are included in the selected configuration will then be imported, with the settings specified in the .csproj file for that configuration.
4. On the next wizard page you can specify some additional options for the import:
   – The option "Generate diagrams" will cause the importer to create a diagram for each imported C# file. The diagram will show all packages, classes, interfaces etc. contained in the file and their relationships.
   – The option "Import now" can be unselected if you want to specify additional advanced options (such as preprocessor settings) before performing the import. See Advanced Import Options for more information.
5. Click "Finish" to complete the Import Wizard.

## Advanced Import Options

The C# Import Wizard only allows you to set a limited number of common options to the C# importer. In order to specify more advanced options follow these steps:

1. Select the top-level import package that was created by the Import Wizard.
2. Open the Properties Editor and select the `<<cSharpImportSpecification>>` stereotype as filter. This stereotype will not be available if you made the import based on a Visual Studio project file. In that case all options for the import are taken from that project file.
3. When you have specified appropriate import options, perform the import by selecting the command **Update Model** that is available in the context menu on the import package.

## Result of C# Import

The result of the import is found under the top-level import package created by the Import Wizard. It will contain UML definitions corresponding to all C# definitions defined in the global namespace. The created package will be given the same name as the C# project file that was imported. If individual C# files were imported the package will be called `ImportedDefinitions_X`, where `X` is a number appended to make the name unique.

When translating C# code to UML a number of translation rules are used. These rules are the same that are used when Generating C# Code, but applied in reverse. See UML to C# Mapping for more information.

In addition to the import package the C# importer will also create a package that contains a model-to-file mapping specification. The purpose of this package is to allow the imported C# code to be regenerated using the C# code generator. The structure of the model-to-file package is therefore identical with the one used by the C# code generator (see Model-to-File Mapping).

## Navigating to and from Imported C# Files

To open an imported C# file you may double-click on the file artifact that represents it in the model. You may also use the **Goto Source** command that is available in the context menu of any imported definition, to navigate directly to the location (or locations - in some cases one UML definition may originate from multiple C# files) of that definition in the imported C# files.

When navigating to an imported C# file it will be opened in Tau's built-in text editor (or in Visual Studio if the Visual Studio Integration has been installed and activated).

It also is possible to navigate in the opposite direction, from the C# code that was imported, to the corresponding UML model entity. This is done by a specific command in the Visual Studio Integration, and by the **Goto Source** command that is available in the context menu of Tau's built-in text editor.

# Synchronizing Model and Source Code

When C# source code has been generated from the model, or when C# source code has been imported to the model, Tau keeps track of the relationship between the code and the model by means of the model-to-file mapping. Be-

sides from providing full navigability from model to code and vice versa the model-to-file mapping also allows for synchronization between the model and the code in case either of these has been changed. Changes to a C# application can thus be made either in the UML model, or in the C# code.

## Automatic vs. Manual Synchronization

The update of the model and source code can be done in two different ways; automatically when the source code or model is changed or manually using special update commands.

### Automatic Synchronization

By default automatic synchronization is turned on. If it has been turned off, it can be enabled by turning on one or both of the following options which control how the automatic synchronization is applied:

- Automatic model update
- Automatic source generation

The details of these settings are described in C# Settings.

### Manual Synchronization

To manually update the C# source code to match what is in the model use the command Update source code or Force update source code.

To manually update the model to match what is in the C# source code use the command Update model or Force update model.

# UML to C# Mapping

This section describes the mapping between the UML and C# languages. The mapping is presented in the form of translation rules from UML constructs to C# constructs. These rules are used when Generating C# Code. The rules can also be applied in reverse in order to define a translation from C# constructs to UML constructs. Applied in that direction these rules are used when Importing Existing C# Code.

Being a modeling language UML is a more comprising language than C#. UML constructs that are not mentioned explicitly in the translation rules below will not be translated to C# when generating code.

# General Translation Rules

This chapter describes general translation rules that apply to many kinds of translated entities.

### Names of definitions

The name of a generated C# definition will be the same as the name of the corresponding UML definition.

Note that no name mangling takes place to guarantee that the generated C# name is a valid C# identifier. In particular your model should not contain

- UML definitions without names, such as operation parameters.
- UML definitions with names that are C# keywords.

### Predefined UML and C# types

Although some UML predefined types have a mapping to a corresponding C# type it is recommended to only use C# predefined types when modeling for C#. UML representations for C# predefined types can be found in the profile `TTDCSharpPredefined`.

The `TTDCSharpPredefined` profile also contains UML representations of the C# array constructs. The `SArray` type represents a single-dimensional C# array, while the `MArray` type represents a multi-dimensional C# array. These types have operations corresponding to the operations that are available on C# arrays.

**Example 473:** ———————————————————————————

**UML**

```
class X {
  Charstring v;
  SArray<int> x;
  MArray<int,2> y;
}
```

**C#**

```
class X
{
    string v;
    int[] x;
    int[,] y;
```

```
}
```

The representation of pointer types (pointers to unmanaged types) makes use of a subset of the `TTDCppPredefined` library, which is the library used for the representation of predefined C/C++ types. For example, this library contains the 'void*' type and the CPtr template which are used for the representation of pointer types in C#.

### Comments

Model comments attached to UML definitions are translated to C# documentation style comments (starting with triple slash (///) and enclosed in an XML tag). Note that the C# compiler only supports documentation style comments for certain kinds of definitions (refer to the C# compiler documentation for more information). Model comments present on other kinds of definitions will not be translated to C#.

The following XML tags are supported in the translation:

- <param>, for operation or delegate parameters
- <summary>, for other definitions

Ordinary annotation comments that are defined in UML textual syntax using /* */ or // are not translated to C#. However, if a model comment starts with // or /* it will be generated as an ordinary C# comment, and not as a documentation style comment.

**Example 474: Translation of comments**

The parameter 's' below has a comment "Param comment" defined in the model (although not displayed in the textual syntax).

**UML**

```
class X comment "//Class comment" {
  void foo(string s) comment "Summary comment";
}
```

**C#**

```
//Class comment
class X
{
    ///<summary>
    ///Summary comment
    ///</summary>
```

```
    ///<param name="s">Param comment</param>
    void foo(string s) {}
}
```

## Package

A package is translated to a namespace.

If the package has the stereotype `<<TTDCppPredefined::globalNamespace>>` applied, it represents the global namespace of C#. No namespace will then be generated for it.

**Note**

*If this stereotype is applied on more than one Package in a model, it is your responsibility to make sure that names within all these packages are unique. It is therefore recommended to use at most one Package for representing the global namespace.*

**Example 475:**

**UML**

```
package P {}
```

**C#**

```
namespace P {}
```

## Class and Interface

A UML class is translated to a C# class.

If the UML class is defined as abstract the C# class will also be abstract. If the UML class is defined as finalized the C# class will be declared as sealed.

A UML interface is translated to a C# interface.

**Example 476: Translation of classes and interfaces**

**UML**

```
class C {}
finalized class FinalImpl {}
interface Ifc {}
```

**C#**

```
class C {}
sealed class FinalImpl {}
interface Ifc {}
```

### Inheritance

UML generalization relationships (inheritance or realization) is translated to C# inheritance (of class or interface).

**Example 477:**

**UML**

```
class C : BaseClass, Ifc1, Ifc2 {}
```

**C#**

```
class C : BaseClass, Ifc1, Ifc2 {}
```

### Partial types

A UML class or interface (and also Datatype) may have the <<partial>> stereotype applied. In that case the corresponding C# class, interface or struct will be defined with the **partial** keyword. This allows you to split its definition over multiple source files. The source files you want to use for each member of the class, interface or struct are specified in the Model-to-File Mapping using <<manifest>> dependencies from the C# file artifacts that represent the source files to the member definitions.

## Datatype

A datatype is translated to a C# enum if it contains at least one literal. If it does not contain any literals it is translated to a C# struct.

**Example 478:**

**UML**

```
datatype D1 {}
enum D2 { L1, L2 }
```

**C#**

```
struct D1 {}
enum D2 { L1, L2 }
```

## Stereotype

A stereotype that inherits the `TTDCSharpPredfined::CSAttribute` stereotype is translated to a class that inherits `System.Attribute`.

**Example 479:**

**UML**

```
public stereotype S : CSAttribute
{}
```

**C#**

```
public class S : System.Attribute
{}
```

## Syntype

A syntype that is defined in a package or a class is translated to a C# “using alias” directive.

**Example 480:**

**UML**

```
syntype MyString = System::String;
```

**C#**

```
using MyString = System.String;
```

## Dependency

An <<access>> dependency between two packages is translated to a using declaration in the namespace that is the translation of the package that owns the dependency.

**Example 481:**

**UML**

```
package P1 <<access>> dependency to P2 {

}
package P2 {

}
```

**C#**

```
namespace P1
{
  using P2;
}

namespace P2 {}
```

An <<access>> dependency from a C# file artifact to a package is translated to a using declaration at the beginning of the corresponding C# file.

# Operation

A UML operation is translated to a C# method. An operation body for the UML operation is translated to a corresponding C# method body. Note that if the UML operation has no specified body, an empty C# method body will be generated.

A derived operation called '[]' with specified 'get' and/or 'set' accessors is translated to a C# indexer.

**Example 482:**

**UML**

```
class X {
  void Operation1() { }
  public <<Derived="true">> string '[]'(int index)
    get {return "First";}
    set {}
}
```

**C#**

```
class X {
  void Operation1() { }
  public string this[int index]
```

```
  {
    get {return "First";}
    set {}
  }
}
```

### Parameter

A UML parameter is translated to a C# parameter. The direction of the UML parameter is translated according to the following table:

| UML Direction | C# Parameter Kind |
|---|---|
| **in** (or unspecified direction) | ordinary parameter |
| **inout** | **ref** parameter |
| **out** | **out** parameter |
| operation return | operation return |

**Example 483**

**UML**

```
bool op(bool p1, in bool p2, inout bool p3, out bool
p4);
```

**C#**

```
bool op(bool p1, bool p2, ref bool p3, out bool p4)
{}
```

### Variable number of parameters

A UML parameter stereotyped by the <<ellipsis>> stereotype is translated to a C# parameter preceeded with the `params` keyword.

**Example 484**

**UML**

```
void op( <<ellipsis>> SArray<int> a);
```

**C#**

```
void op(params int[] a)
{}
```

### Virtual, redefined and finalized operations

A virtual operation is translated to a virtual method.

A redefined operation is translated to an override method.

A finalized operation is translated to a sealed method.

**Example 485:**

**UML**

```
virtual void op1();
redefined void op2();
finalized void op3();
```

**C#**

```
virtual void op1() {}
override void op2() {}
sealed void op3() {}
```

### Type conversion operators

An operation stereotyped by `<<implicitTypeConvOperator>>` is translated to an implicit C# type conversion operator.

An operation stereotyped by `<<explicitTypeConvOperator>>` is translated to an explicit C# type conversion operator.

In both cases the return type of the operation designates the target type of the conversion operator.

**Example 486:**

**UML**

```
class C {
    static <<implicitTypeConvOperator>> int implicit(C
x);
    static <<explicitTypeConvOperator>> bool explicit(C
x);
}
```

**C#**

```
class C
{
     public static implicit operator int(C x) {}
     public static explicit operator bool(C x) {}
}
```

## Delegate

A UML delegate is translated to a C# delegate. Parameters of the delegate are translated in the same way as operation parameters (see Parameter).

**Example 487**

**UML**

```
class B {
  delegate void Delegate1(bool p1,string p2);
}
```

**C#**

```
class B
{
  delegate void Delegate1(bool p1, string p2);
}
```

## Attribute

A non-derived UML attribute is translated to a C# field.

A derived UML attribute with specified ‘get’ and/or ‘set’ accessors is translated to a C# property.

**Example 488: Translation of UML attributes**

**UML**

```
class Class1 {
  string a;
  string / b
  get
  {
    return "foo";
  }
  set
```

```
    {
      a = value;
    };
  }
```

**C#**

```
  class Class1
  {
    string a;
    string b
    {
      get
      {
        return "foo";
      }
      set
      {
        a = value;
      }
    }
  }
```

---

If the UML attribute is typed by the `Event` type from `TTDCSharpPredefined` it is translated to a C# event.

**Example 489:** ---

**UML**

```
public delegate void D(object 'sender', EventArgs e);
public class SampleEventSource {
  public Event<D> SampleEvent;
}
```

**C#**

```
public delegate void D(object sender, EventArgs e);
public class SampleEventSource
{
  public event D SampleEvent;
}
```

---

### Constant attribute

A constant UML attribute is translated to a C# constant.

**Example 490:**

**UML**

```
const long x = 4;
```

**C#**

```
const long x = 4;
```

### Attributes with non-single multiplicity

If a UML attribute has an informal non-single multiplicity it is translated to a C# field typed by the type that is the translation of the UML attribute type. The informal multiplicity is thus not present in the C# translation.

If the attribute instead has a formal multiplicity the `<<containerType>>` stereotype is used to specify which container type to use in the C# translation. See The <<containerType>> stereotype for more information.

When a new C# project is created `<<containerType>>` is by default applied on the Model level specifying System::Collections::Generic::List<Any> as the default container type to use.

**Example 491:**

In this example `a1` and `a3` has formal multiplicity while `a2` has informal multiplicity. Both have non-single multiplicity. `a3` is contained in a package which specifies a custom container type to use.

**UML**

```
class X {
  B [*] a1;
  System::Collections::Generic::List<B> {[*]} a2;
}
<<containerType (. Type = MyOwnContainerType .) >>
package P {
  class Y {
    B [*] a3;
  }
}
```

**C#**

```
class X {
  System.Collections.Generic.List<B> a1;
  System.Collections.Generic.List<B> a2;
```

```
}
namespace P {
  class Y {
    MyOwnContainerType a3;
  }
}
```

## Association

Unnamed uni-directional associations are represented as attributes in the UML model. The translation of such attributes thus follows the rules in Attribute.

## Template

A UML template definition is translated to a corresponding C# definition with generic parameters. Constraints on UML template parameters are translated to corresponding constraints on the C# generic parameters.

**Example 492:**

**UML**

```
template <class T atleast B >
class Class1 {
  T a;
}
```

**C#**

```
class Class1<T> where T : B
{
  T a;
}
```

## Expression

Several UML expressions have a defined mapping to C# expressions. However, bear in mind that the type of a UML expression might be a predefined UML type while the type of a C# expression might be a predefined C# type. Predefined UML types are not always fully compatible with predefined C# types. A certain level of compatibility exists, namely that predefined C#

types can be constructed from predefined UML types. However, the set of operators that are available on predefined types are not identical in the UML and C# languages.

If you get semantic errors in your model caused by type incompatibilities between predefined types you can choose to use an informal UML expression. In such an informal expression a C# expression can be inlined, and that expression text will be copied to the generated C# file verbatim by the code generator. An informal UML expression is type compatible with all types, which means that you sometimes must use a type cast (or declare an intermediate attribute) to avoid ambiguities in calls to overloaded operations.

The translation of (non-informal) UML expressions are specified in the table below.

| **UML Expression** | **C# Expression** | **UML Example** | **C# Example** |
|---|---|---|---|
| parenthesis expression | parenthesis expression | `(a + b)` | `(a + b)` |
| unary expression | unary expression | `-a` | `-a` |
| this expression | this expression | `this.x` | `this.x` |
| binary expression | binary expression | `a = b` | `a = b` |
| index expression | indexer access | `obj[10]` | `obj[10]` |
| create expression | use of the **new** operator | `new C()` | `new C()` |
| conditional expression | use of the conditional (**?:**) operator | `a ? b : c` | `a ? b : c` |
| real value | double literal | `3.14` | `3.14` |
| integer value | int literal | `8` | `8` |
| charstring value | string literal | `"Tau"` | `"Tau"` |
| call expression | invocation expression | `foo()` | `foo()` |
| field expression | member access expression | `a.b` | `a.b` |

| UML Expression | C# Expression | UML Example | C# Example |
|---|---|---|---|
| instance expression | attribute | `S(. .)` | `[S()]` |
| identifier | simple name | `x` | `x` |
| base expression | **base** | `base` | `base` |
| value expression | **value** | `value` | `value` |
| list expression | array initializer | `{1,2}` | `{1,2}` |
| delegate implementation expression | anonymous method expression | `delegate() {return;}` | `delegate(){return;}` |

The C# language contains more expressions than those listed in the above table. These are represented in UML as informal expressions.

### Special call expressions

Calls of certain UML operations are translated in a special way.

| Called UML operation | C# Expression | UML Example | C# Example |
|---|---|---|---|
| Predefined::cast | cast expression | `cast<C>(x)` | `(C) x` |
| TTDCSharpPredefined::typeof | use of **typeof** operator | `typeof(long)` | `typeof(long)` |
| Predefined::is | use of **is** operator | `is<MyC>(x)` | `x is MyC` |
| Predefined::as | use of **as** operator | `as<MyC>(x)` | `x as MyC` |
| TTDCSharpPredefined::Default | default value expression | `Default<C>()` | `default(C)` |
| TTDCppPredefined::GetValue | pointer member access | `p.GetValue().x` | `p->x` |

## Action

Several UML actions have a defined mapping to C# action code. However, bear in mind that the action semantics of the UML and C# languages are not identical, and that the C# code generator currently does not perform any

transformations in order to adjust for the different semantics. Because of that it is often best to specify behavior of operations that are to be translated to C# using informal an UML action. In such an informal action C# code can be inlined, and that code will be copied to the generated C# file verbatim by the code generator.

The C# importer imports all C# action code as informal UML actions.

**Example 493: Using informal UML action with inline C# code**

**UML**

```
void hi() {
[[System.Console.WriteLine("Hi there!");]]
}
```

**C#**

```
void hi()
{
System.Console.WriteLine("Hi there!");
}
```

The translation of (non-informal) UML actions are specified in the table below. Examples of the translation rules are only specified where there is a difference between the UML and C# syntaxes, or when it is not obvious what kind of action that is referred to.

| UML Action | C# Statement | UML Example | C# Example |
|---|---|---|---|
| compound action | compound statement | | |
| continue action | continue statement | | |
| break action | break statement | | |
| if action | if statement | | |
| loop action | for, while or do statement depending on kind of loop action | | |

| UML Action | C# Statement | UML Example | C# Example |
|---|---|---|---|
| expression action | expression statement | `foo(3);`<br>`x = 10;` | `foo(3);`<br>`x = 10;` |
| definition action | declaration statement | `{`<br>`string s;`<br>`}` | `{`<br>`string s;`<br>`}` |
| try action | try statement | | |
| throw action | throw statement | | |
| decision action | switch statement | | |
| return action | return statement | | |
| join action | goto statement | `join L;` | `goto L;` |

UML actions not mentioned in the above table will not be translated to C#.

# C# Settings

Each C# model has a set of C# specific settings stored in the project file. To view or edit these settings:

1. Select the **Model** node in the Model View.
2. Right-click and select **Properties...**
3. Click the **Stereotypes...** button in the Properties Editor and apply the <<C# Settings>> stereotype.
4. In the **Filter** drop-down, select **C# Settings.**

The following settings are available:

- Automatic model update
  - If this setting is turned on the model will automatically be updated when the C# source files change, e.g. when modifying and saving the files in an external text editor. The default is that the automatic model update is turned on.
- Automatic source generation
  - If this setting is turned on the C# source files will automatically be updated when the model is changed. The update will happen a while after some model editing has been completed. The default is that the automatic source generation is turned on.

- Support roundtrip
  - By default the support for roundtrip is turned on. You may turn it off if you don't intend to make any changes to the generated C# files that should be propagated back to the model. In that case all C# source files will be completely regenerated each time code generation is performed.

# 48

# *Visual Studio Integration for C#*

The integration between Tau and Visual Studio facilitates building and debugging a generated C# application. For more information refer to the documentation of the Visual Studio Integration.

# *UML and C++*

This chapters that are listed under UML and C++ describe how a UML project is turned into a C++ application.

# 50

# *C++ Support in Tau*

This section is an introduction to the C++ support in Tau and a guide for using the tool to manage various C++ usage scenarios.

- Visualization of existing C++ code
- UML - C++ roundtrip engineering
- Application generation for advanced UML concepts
- Migration of existing C++ applications to Tau
- Using Tau managed C++ code in a C++ development environment
- Accessing C++ APIs from the Tau UML Environment.
- Tracing execution of Tau generated applications

# Overview

## Key capabilities

The basic functionality of the C++ support in Tau is illustrated in Figure 248 on page 1516

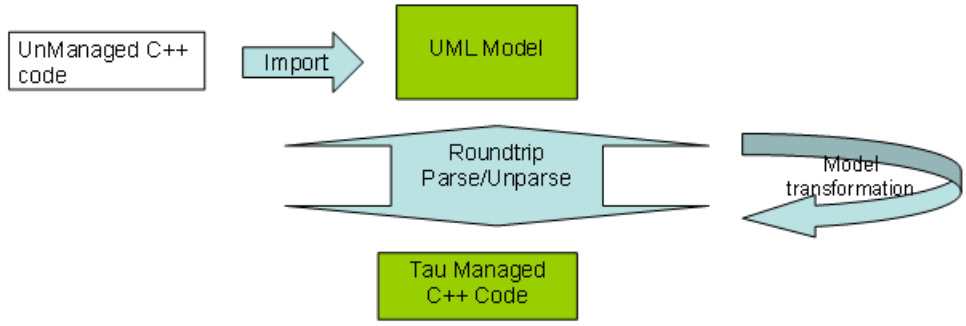



*Figure 248: C++ roundtrip*

The key capabilities are the roundtrip parse/unparse of C++ code and the model transformations. The roundtrip parse-unparse enables a one-to-one mapping between C++ syntax and certain concepts in a UML model. This is used to keep C++ files synchronized with a UML model. The model transformations makes it possible to use more advanced UML concepts for which there is no one-to-one mapping to C++, for example the ability to generate C++ code from UML state machine definitions.

## External C++ and roundtrip

Tau includes a general-purpose C/C++ Import that can be used to import C++ APIs to a UML model. The main difference between this and the roundtrip parse/unparse is the difference of focus: C/C++ Import is intended to support the C++ language, including preprocessing directives and macros. The import will make some transformations during the import process, for example an expansion of macros. The main focus of C/C++ Import is to make the APIs of C++ libraries available to UML developers. The roundtrip C++ parse/unparse functionality is focussed on preserving a tight synchronization of C++ code and the UML model. To accomplish this, a subset of C++ is defined for which a one-to-one mapping to C++ can be maintained. Only concepts within this subset can be used in the roundtrip parse/unparse.

## Using C++ in Tau

There are several different scenarios in which these basic capabilities can be used:

- Visualization of existing C++ code
- UML - C++ roundtrip engineering C++ code
- Application generation for advanced UML concepts (for example based on UML state machines and composite structure definitions)

In addition there are special cases that require some extra consideration:

- Migration of existing C++ applications to Tau
- Using Tau managed C++ code in a C++ development environment
- Accessing C++ APIs from the Tau UML Environment
- Tracing execution of Tau generated applications

# C++ Usage Scenarios

## Visualization of existing C++ code

This is an example of how Tau can assist a C++ developer. In this scenario C/C++ Import is used together with the built-in ability to provide an auto layout of diagrams that exist in Tau.

Consider a set of C++ header files that contain a number of C++ class definitions. To visualize classes that are defined in header files:

- Use the Import Wizard to import the C++ files into a UML model (**File** menu, **Import** command). Select **C/C++** and choose the files to import. In simple cases it is possible to directly import the file, resulting in a root level package named **ImportedDefinitions** with definitions from the imported files.
- In some cases it is necessary to provide some extra information to the import process, for example include paths. To do this you cancel the selection of the **Import now** choice in the wizard and change the settings for the import as described by the properties for the **ImportedDefinitions** package. You can then finalize the import by right-clicking the **ImportedDefinitions** package and from the shortcut menu select the **Import C/C++** menu choice.
- To visualize the imported classes in a diagram select what classes to display using the Show elements command that is available from the shortcut menu in the diagram, or use the Create Presentation dialog to create the element and the diagram.

### Navigation from model view to source code

The C++ Code Generator enables model to code navigation. If there exist header or implementation files as a result of code generation, navigating from a UML element to the corresponding lines in code is done by right-clicking on it in the model view and choosing **Go to source**.

**Note**

*With the Visual Studio integration, two-way navigation is at hand, you can also Locate in Tau from C++ source code.*

# UML - C++ roundtrip engineering

The roundtrip features of Tau are described by the following example. The examples models a simple “Hello World...” style application using Tau.

### Creating the project

Create a new project in Tau, using the menu **File -> New** and by selecting the project type **UML for C++ Code Generation** in the project wizard.

The next step is to create a simple UML model with a class “Hello” and one operation “PrintIt” as in Figure 249 on page 1519.

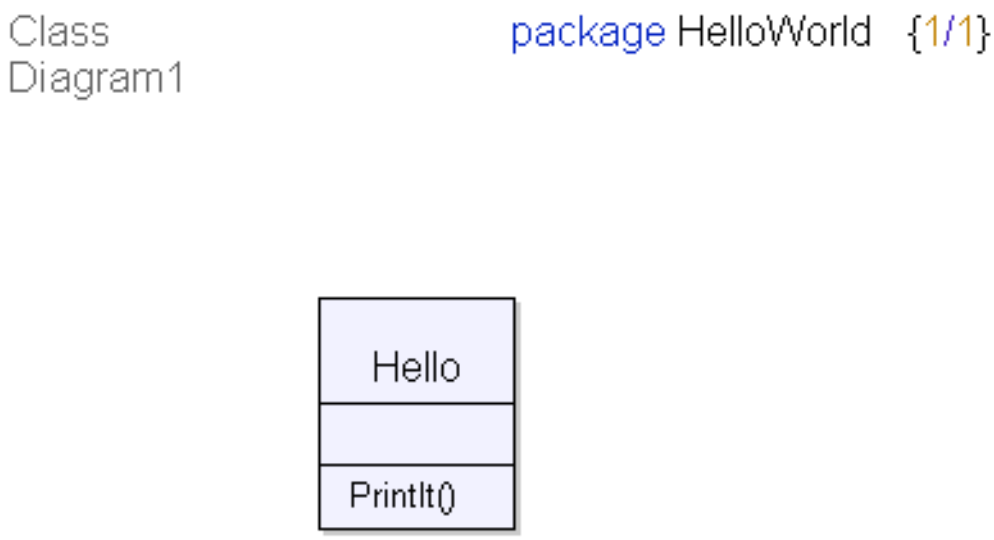



*Figure 249: Class diagram example*

### Generating C++ code

To generate C++ code choose the **Generate Configuration** command from the **Build** menu. When the Build Wizard prompts for input the only information you need to specify is to select the package you just created as the Build Root. All elements within this package will be included in the generated C++ code.

After the wizard is closed, C++ files corresponding to the UML model are generated. The generated files will be in a package “Result of C++ Code Generation”. The file mapping used by the code generator is to produce one header file and one source file for each class, thus there are now two files, `Hello.h` and `Hello.cpp`.

By double-clicking on the `Hello.h` file artifact in the model view, the text editor is opened on that file. You can now verify that the class definition is generated as expected:

```
#ifndef GEN_IzuOjIq0WdSL_45GMYmEjyT3hE
#define GEN_IzuOjIq0WdSL_45GMYmEjyT3hE
#include "torAnnotations.h"

namespace HelloWorld {
    class Hello {
        void PrintIt();
    };
}

#endif/*<GENERATED> GEN_IzuOjIq0WdSL_45GMYmEjyT3hE */
```

The generated header file contains the line:

```
#include "torAnnotations.h"
```

The `torAnnotations.h` file contains macros that are used when doing roundtrip on certain UML constructs, for example interfaces which otherwise would be transformed to classes and not interfaces. The file is used to increase the precision in the mapping to UML. It is always generated to support for example that the user could manually add interfaces and push them into the model.

### Applying changes to the generated C++ code

You can now modify the file and propagate the changes back to the UML model.

Add an attribute to the class, say “i”.

```
#ifndef GEN_IzuOjIq0WdSL_45GMYmEjyT3hE
#define GEN_IzuOjIq0WdSL_45GMYmEjyT3hE
#include "torAnnotations.h"

namespace HelloWorld {
    class Hello {
        int i;
        void PrintIt();
    };
```

```
}

#endif/*<GENERATED> GEN_IzuOjIq0WdSL_45GMYmEjyT3hE */
```

### Propagating changes back to model

To propagate the changes back to the UML model:

Save the `Hello.h` file and select **Update Configuration** from the **Build** menu.

In the Model View you can check that the class `Hello` now has the extra attribute "`i`".

Some aspects to be aware of when using the roundtrip features to maintain C++ code:

- Only a subset of C++ syntax can be used in roundtrip. This subset covers the most commonly used C++ language features. This subset is described in the C++ Textual Syntax.
- To use un-supported constructs, "user sections" can be created. See the section on Comments in the C++ Textual Syntax.

By default, the **Update** operation will only consider files that has changed since the most recent update or code generation. To bypass this and force an update use the corresponding **Full Update** operation.

### Automatic model and code synchronization

In the scenarios described above manual commands were used for synchronizing the model with the code and vice versa. It is also possible to do this synchronization automatically, so that each time the model is changed (and saved) the code will be regenerated, and each time a generated file is changed (and saved) the model will be updated. This feature is by default turned off, but it can be enabled by following these steps:

1. Select the build artifact in the Model View.
2. Choose **Properties** in the context menu.
3. Select 'C++ Application Generator' in the **Filter** list of the Properties Editor.
4. To enable automatic synchronization of the code, turn on the option **Automatic code generation**.

5. To enable automatic synchronization of the model, turn on the option **Automatic model update**.

**Note**

*If the option for automatic model update is enabled, the **Support roundtrip** option must also be enabled.*

## Application generation for advanced UML concepts

As mentioned earlier in this chapter, you can use the UML - C++ roundtrip engineering features of Tau to handle all common concepts like classes, operations, data type that have a one-to-one mapping between their UML and C++ representations. However if you want to use more advanced UML concepts you can benefit from the model transformations and the related run-time framework supplied as a part of the C++ support.

### Model transformations

Some of these transformations are performed interactively in the model during editing. for example when a class is marked as **Active**. A class marked as Active will automatically inherit from “DispatchableClass” (part of the supplied run-time framework). Most transformations will however be performed on a copy of the UML model during code generation and will not be visible in the UML model.

### Run-time framework

The run-time framework is delivered both as a UML model that contains all aspects of the framework that are relevant from a UML point of view, as well as C++ source code. You can find the UML model in the **Libraries** section in the Model View under the heading **tor** (short form for Tau Object Run-Time).

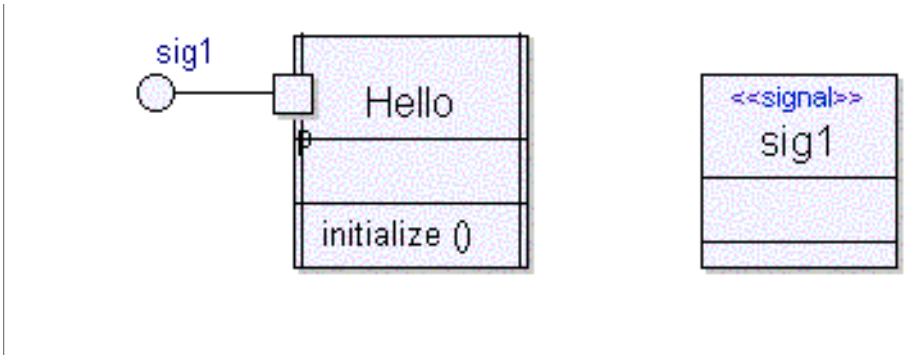



*Figure 250: Signal definition in a class diagram.*

Mark the Hello class from the previous example as Active. Add a signal "sig1" to the model (Figure 250 on page 1523) and add a state machine to the "Hello" class (Figure 251 on page 1523).

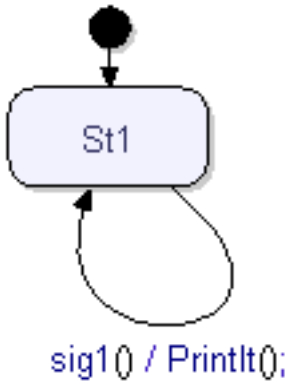



*Figure 251: State machine example.*

### Generating C++ code

When you generate code from this model, the transformations will create substantially more C++ code than in the roundtrip example earlier. Open the `Hello.cpp` file (e.g. by double-clicking on the corresponding file artifact in the **Result of C++ Code Generation** package) and have a look at the generated code.

Some aspects worth noting in the file is the class `Hello_initialize` that is the C++ representation of the state machine and the function `HelloWorld::Hello_initialize::trans_St1_sig1` that contains the C++ code corresponding to the transition in the simple state machine you created.

To make the program complete you must implement the "PrintIt" operation. A `main()` function will be generated automatically. For a more complex case it is possible to write your own `main()` function.

The "PrintIt" operation can be defined as follows:

```
void PrintIt() {
    std::cout << "Hello World\n";
}
```

An interesting aspect of this function is that it uses `std::cout`, available in UML by activating the `CppStdLibrary` add-in. This function is also available in C++ by including `<iostream>`.

Access to external C++ headers from the model can be done in two ways:

- Import the API as described in section Accessing C++ APIs from the Tau UML Environment.
- Use the functions in target code statements (for example statements surrounded by `[[ ... ]]`) that will not be checked by the UML environment.

If you plan to make substantial use of an API the import method is preferable, but in simple situations the target code approach is often easier. When the inline code approach is used you must define the files to include during the make process, by defining an «`include`» dependency between UML artifacts that represent the files in the model. The `include` statements to imported headers will be added automatically.

In the current example you can do as shown in Figure 252 on page 1525

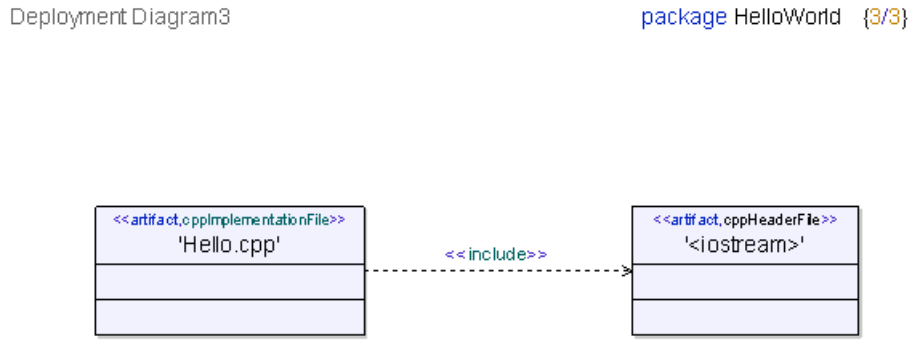



*Figure 252:An «include» dependency example*

You can now regenerate the code for the application. If you choose the command **Build->**Build Configuration to do this, you will also invoke compilation and linking of the program. The end result is an executable, found in the Target Directory specified for the files generated by the C++ Application Generator.

If you have not specified the Target Directory, it will be in a directory named after the build artifact, located below the directory containing the project file. If you start the application from a console window you should now see the familiar “Hello World!” printed on the screen.

**See also**

Chapter 54, C++ Run-time Framework for more details about the run-time model.

Chapter 52, C++ Application Generator Reference for a detailed description of the UML concepts that are supported at C++ code generation and their mapping to C++ constructs.

## Accessing C++ APIs from the Tau UML Environment

It is common that Tau is used to develop some parts of an application and other tools can be used for other parts. It is important to be able to access the definitions from all parts of the application also in the UML model, which is supported in Tau when using the C/C++ Import.

# Using Tau managed C++ code in a C++ development environment

### Using Visual Studio with Tau

Tau includes a Visual Studio Integration, containing support for creating Visual Studio projects directly from Tau, easy navigation from source code in Visual Studio to UML models and other features.

**Note**

*The generated C++ code is not Visual Studio specific, any C++ compiler can be used with Tau.*

### Workflow

A possible workflow for using Tau together with a C++ development environment could be:

- Perform the analysis phase of the application using UML in Tau (potentially also preceded by a requirements analysis using Telelogic DOORS).
- Refine the analysis model to a design model in UML
- Generate C++ code from the model
- Include the Tau generated files in your preferred C++ development environment
- Use the roundtrip features and/or the model transformation features to update the C++ files
- Use the make and debug features of the C++ development environment to finalize and deploy your application.

# Migration of existing C++ applications to Tau

Tau includes support for the migration of an existing C++ application to UML. This is based on using C/C++ Import to import the C++ files and then use the roundtrip facilities and code generator to regenerate new C++ files.

The header files of an external application can be imported. Both definitions and behaviour code that is present in the header files can be imported. It is currently not possible, in the general case, to import C++ implementation files.

### How to import C++ files

Essentially the workflow is the following:

1. Use **File->Import** to start the Import Wizard.
2. Select the files you want to import.
3. Make sure to enable the **Import action code** and **Generate artifacts** options.
4. In simple cases you may now perform the import by leaving the **Import now** check box enabled and finish the wizard. Continue with step 7)
   In case you need to set more import options than what can be done from the import wizard, disable the **Import now** check box and finish the wizard. This will cause a package called "ImportedDefinitions" to be created on root level in your model.
5. (Optional) Open the property pages for the ImportedDefinitions package and uncheck the "**Add source file references to enable navigation from the UML model to the C++ source**" check box. Doing this means losing the possibility to navigate back to the original files, but there will be less confusion when you have regenerated new files based on the model. If you don't perform this step there will, after code generation, be source references both to the original and the generated files.
6. Finish the import using the command **Import C/C++** available in the shortcut menu for the **ImportedDefinitions** in the Model View.
7. You should now be able to find the definitions from the imported files in the **ImportedDefinitions** package that was created by the importer. In this package there are also file artifacts representing the imported files, and a build artifact to be used for regenerating the files with the C++ Application Generator.
8. (Optional) Reorganize the imported definitions into various packages and manifest them on different files.
9. Use the **Build->Generate** command on the generated build artifact to regenerate C++ from the files you imported. Note that by default the new files will be placed in a different directory than the original files. This can be changed by setting the **Target Directory** option of the build artifact. You may also want to set appropriate build settings, such as preprocessor directives, before performing the code generation.

### Restrictions

There are some restrictions in the migration support. These restrictions may make it necessary to do modifications to the C++ code before importing it and impose changes that should be applied to the UML model before regenerating C++ code from it:

- Only header files can be imported. Importing implementation files sometimes work, but often leads to duplicate definitions in UML since different implementation files typically include the same header files. Using the importer option "**Do not import definitions from included header files"** may help to avoid this problem. In any case, if you attempt to import implementation files, do it in a separate pass after the header files have been imported.
- The imported files go through a Preprocessor and it is the resulting preprocessed C++ that is imported to UML. Macros, conditional compilations and other preprocessor directives will not be traceable in the resulting UML model.
- Comments from the C++ code are not imported for the same reason.

## Tracing execution of Tau generated applications

When generating C++ code it is possible to add tracing functionality. This is called instrumenting the code and it generates lines that will produce meta data when the application is running. This data will either feed Tau with information needed to visualize the execution in a sequence diagram, or it will be piped to a log file.

### Workflow

1. Enable instrumentation in the build artifacts C++ Application Generator properties
2. Generate code.
3. Activate the trace by adding lines to the code, preferably early in the main function. Example:

```
int main() {
    //<GENERATED>
TOR_INSTRUMENTATION(TOR_GEN_Mxon1Vw89tiLYt_459WVk8GBGI);
    //</GENERATED>
tor::meta::eventManager.createNewHostTracer();
tor::meta::eventManager.createNewLogFile("C:/log");

    //Your code...
```

```
}
```

4. Build and run the application. Follow the execution in a Tau sequence diagram.

**Note**

*Tracing functionality runs in its own thread. To cope with this, make sure that you are building a multi-threaded application.*

5. Control the Tau Trace functionality at run-time with the Visual Studio integration.

**Note**

*Using the Visual Studio run-time control does not require any hand written code (step 3).*

**Instrumentation API**

For sequence diagram trace, create or delete host tracers:

```
tor::meta::eventManager.createNewHostTracer();
tor::meta::eventManager.deleteAllHostTracers();
```

Creating and deleting log files:

```
tor::meta::eventManager.createNewLogFile("C:/log");
tor::meta::eventManager.deleteAllLogFiles();
```

Enable and disable the instrumentation:

```
tor::meta::eventManager.setActive(true);
tor::meta::eventManager.setActive(false);
```

# Getting started with the C++ support

To get started with the C++ support there exists some C++ examples in the Tau installation. To start your development using one of these examples simply choose the **Templates** tab in the **New Wizard** (which is started using the **File->New** command).

Among the templates are a number of C++ examples. These are identified by their respective name, as the example names are constructed to contain a prefix of language and code generator stereotype (Cpp for C++ Application Generator). Select the example of your choice and you have a running UML/C++ application that can be modified and extended to suit your needs.

# 51

# *C++ Textual Syntax*

The supported C++ textual syntax is defined for use with the round-trip feature of the C++ Application Generator.

The Complete listing is only available in the on-line help file.

# 52

# *C++ Application Generator Reference*

This chapter is a reference guide to the C++ Application Generator.

# General

The C++ Application Generator provides services that are scheduled to interact with the code generation, extends a copy of the original model when needed, parses the C++ files produced from a previous code generation pass in order to merge user written and code generated by the C++ Application Generator, builds the structure of the code and finally writes the code on files.

## C++ Application Generator add-ins

In order to create models intended to generate code with the C++ Application Generator, the Add-Ins described below have to be enabled (from the **Tools** menu, select Customize). These add-ins are automatically enabled when creating a new "UML for C++ Code Generation" project, but have to be manually enabled otherwise.

### CppGen

This add-in loads the profile `TTDCppAppGen`, which contains a UML specification of the C++ Application Generator in the form of model transformations and dependencies between these transformations. The profile also contains stereotypes for describing the options to the code generator.

Normally you do not have to be concerned about this profile. It provides a means for the advanced user to customize the way the UML model is translated into C++ code.

### CppTypes

This add-in loads the profile `TTDCppPredefined`, which contains UML representations of the C++ fundamental types (such as `int, bool, char` etc.). In order to use fundamental C++ types in your model, you may refer to these types. The profile also contains stereotypes which are used to represent C++ constructs that cannot be natively expressed in UML. For example, there is a stereotype `«inline»` which can be applied to an operation to specify that it should be generated as an inline declared C++ function.

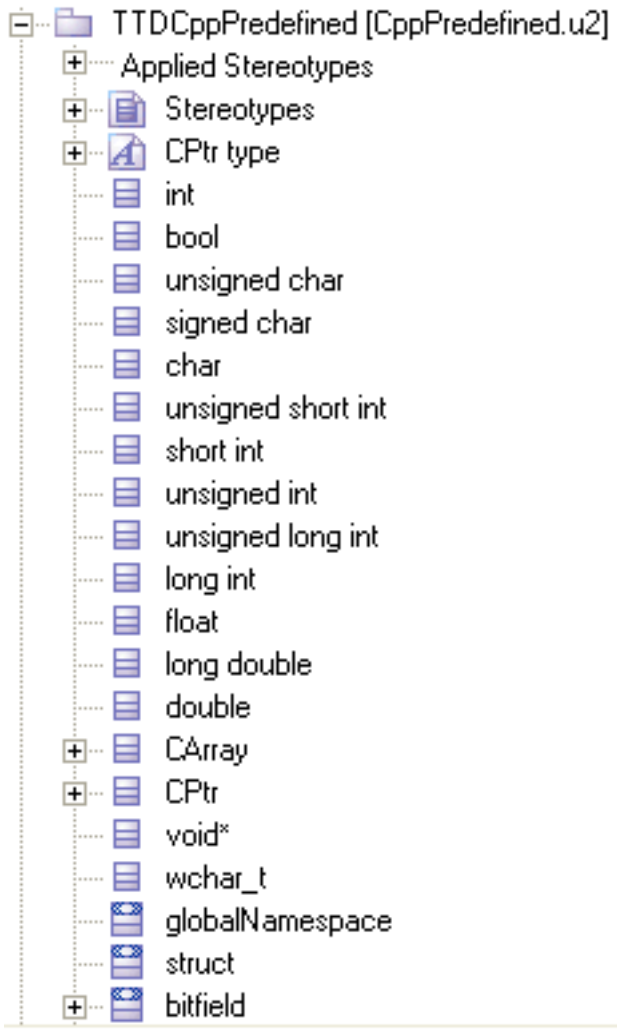



*Figure 253 Parts of the TTDCppPredefined profile*

## Basic principle of the C++ Application Generator

The C++ Application Generator is based on the principle of generating code through model transformations followed by a pretty printing step (this is also referred to as unparsing). The purpose of the model transformations is to transform all non-trivial UML constructs into trivial constructs, which can be handled by the C++ unparser. The model transformations are performed “in-place” rather than building new result models after each transformation.

The C++ Application Generator is started by the Application Builder as a separate executable according to the ordinary Application Builder procedures. As input from Application Builder, the C++ Application Generator receives:

1. A «build» artifact. This artifact contains Tagged values for the usual Application Builder stereotypes (Target Directory etc.) as well as for the C++ Application Generator stereotype (containing all the Translation Options).
2. A list of entities to build. These are either the entities selected by the user ("build selection") or the entities manifested by the Build Artifact ("build artifact" or "Build Configuration"). The C++ Application Generator iterates over these entities, and for each entity the corresponding «file» artifact (see "Model-to-File Mapping" on page 1536) is located. The resulting set of «file» artifacts represents the set of target C++ files to generate (or update).

## Document structure

The following chapters describe the subset of UML that can be translated to C++. UML language constructs not mentioned are ignored during translation.

For each supported UML construct a translation rule is given. If there are exceptions to the rule, these are also mentioned.

Each supported UML construct can be classified as either trivial or non-trivial. Trivial constructs can be directly mapped to C++ by the C++ Unparser, while non-trivial constructs first must be transformed into trivial constructs before the C++ Unparser can generate C++ for them. Round-tripping is supported for trivial constructs only.

For most translation rules examples are given using concrete UML and C++ syntax. The purpose of each example is only to illustrate the translation rule at hand, not to give a precise description of how the generated code will look like.

# Model-to-File Mapping

A UML model may contain a specification on how to map model elements to generated C++ files. Such a specification consists of artifacts representing the source files (both header and implementation files), and dependencies

from such artifacts to model elements that are to be generated in those source files. The dependencies are stereotyped by the predefined 'manifest' or 'manifest implementation' stereotypes, and the artifacts are stereotyped by a derivate of the 'file' stereotype containing a 'path' attribute for specifying the name and location of the file.

For clarity, the artifacts are presented in a folder of the containing package instead of intermixed with other model elements.

In the context of C++ code generation, two derivates of the 'file' stereotypes are considered:

**cppHeaderFile**

Represents a C++ header file

**cppImplementationFile**

Represents a C++ implementation file

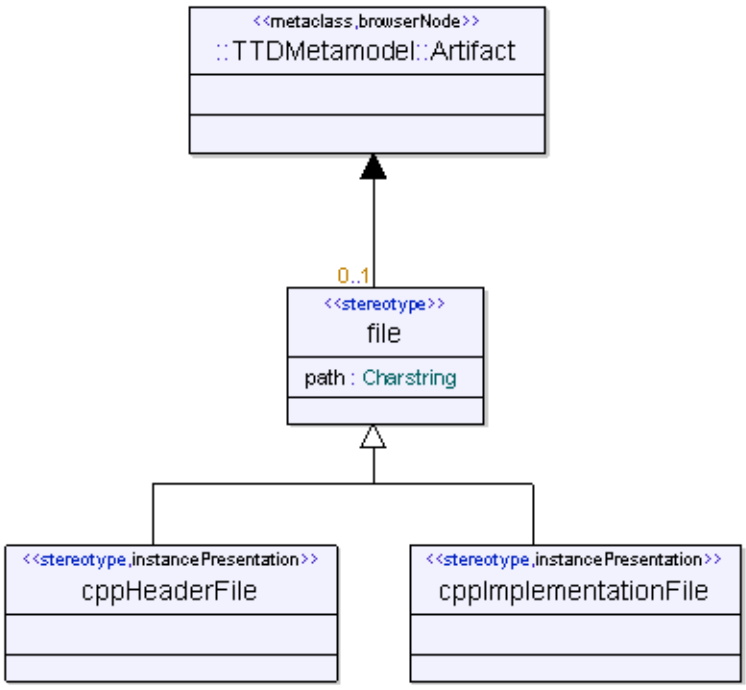



*Figure 254 File artifacts of interest for C++ translation*

**When the model-to-file mapping has been explicitly specified by the user, using artifacts and stereotypes as described above, and the option to generate a default model-to-file mapping is not turned on, then the only source files that will be generated are those described in that mapping.**

Each header file will contain all entities that have been specified as manifested by that file.

Each implementation file will contain all entities that have been specified as manifested by that file, and also the implementations of the entities that have been specified as manifested through a manifest implementation relationship by that file.

**Example 494:**

**UML**

```
<<cppHeaderFile (. path = "MyClass.h".)>> artifact
MyClassHeader
<<manifest>> dependency to MyClass;
<<cppImplementationFile (. path = "MyClass.cpp".)>>
artifact MyclassImpl <<'manifest implementation'>>
dependency to MyClass;
class MyClass {
    int foo(){
        return 14;
    }
}
```

**C++** (in file `"MyClass.h"`)

```
class MyClass {
  int foo();
};
```

**C++** (in file `"MyClass.cpp"`)

```
#include "MyClass.h"
int MyClass::foo(){
  return 14;
}
```

Although `MyClass::foo` is “inline-defined” in UML, its C++ implementation will go into the implementation file and not into the header file. The `«inline»` stereotype can be used to specify that a function should be inline, in which case the implementation will be placed in the header file.

If an entity, that has been selected by the user to be built, is not specified to be manifested in a C++ header or implementation file, it will not be translated. However, for convenience the C++ Application Generator has an option to create a default model-to-file mapping. This option is on by default, so if you have made an explicit model-to-file mapping it is a good idea to turn this option off.

**The default model-to-file mapping is generated according to the following:**

1. A header file and an implementation file is generated for each package.
2. A header file and an implementation file is generated for each structured classifier (class, choice or interface) contained in a package.
3. A header file is generated for a stand-alone state machine (a state machine owned by a package). If it has a state machine implementation, an implementation file is also generated.
4. The name of a synthesized artifact in a default model-to-file mapping is the same as the name of the entity for which the file is generated

**Note**

*When an entity is already manifested in a file (directly, not indirectly through composition) the default model-to-file mapping will not create another file mapping for that entity.*

**Entities in a model are then translated into these files according to the following:**

1. An entity for which a header file artifact is produced is translated into that header file, by adding a `«manifest»` dependency from the header file artifact to the entity.
2. An entity for which an implementation file artifact is produced is translated into that implementation file, by adding a «manifest implementation>> dependency from the implementation file artifact to the entity.
3. Other entities are translated into the same file as their composition owners, that is to say that translation is recursive with regards to composition.

**The names of the generated files are typically specified explicitly by means of a tagged value for `file::path`. If it is unspecified (which for example is the case in a default model-to-file mapping), the name of the artifact that represents the file is used instead.**

**Example 495: Default model-to-file mapping**

**UML**

```
package test {
    class S {
        Integer Operation1() {
            return 5;
        }
    }
    Integer var;
    extern int var2;
}
```

**C++** (in file `"test.h"`)

```
namespace test {
  extern tor::Integer var;
  extern int var2;
};
```

**C++** (in file `"test.cpp"`)

```
#include "test.h"
namespace test {
  tor::Integer var;
};
```

**C++** (in file `"S.h"`)

```
namespace test {
  class S {
    tor::Integer Operation1();
  };
};
```

**C++** (in file `"S.cpp"`)

```
#include "S.h"
tor::Integer test::S::Operation1() {
  return 5;
}
```

## Include protection

**Each generated C++ header file is protected from multiple inclusions by generation of a conditional compilation macro.**

The default name of the macro is a unique identifier, based on the GUID of the artifact representing the header file.

**Example 496:**

**UML**

```
class X {
}
```

**C++** (in file `"X.h"`)

```
#ifndef GEN_Wr5SkVKjOm5LrSHaaV45sx9V
#define GEN_Wr5SkVKjOm5LrSHaaV45sx9V
class X {
};
#endif //GEN_Wr5SkVKjOm5LrSHaaV45sx9V
```

It is possible to modify the format of the include protection by setting the properties “includeProtectionBegin” and “includeProtectionEnd” of the file artifact that represents the header file. In the Properties Editor these properties are presented as “Include Protection First String” and “Include Protection Last String” respectively. These properties can also be set on the build artifact and will then control the format of the include protection for all header files that are generated (except those that override this setting by having these properties explicitly specified).

The include protection properties are text strings that may contain the following codes for substitution:

| Code | Will be replaced by |
|---|---|
| `%%` | A single ‘%’ character. |

| Code | Will be replaced by |
|---|---|
| %f | The base name of the file (the initial path removed) with any '.' replaced with an underscore. E.g. "my/path/Xyzzy.h" is transformed into "Xyzzy_h". |
| %F | As %f but any lowercase letters are converted to upper case. E.g. "my/path/Xyzzy.h" is transformed into "XYZZY_H". |
| %g | A string based on the GUID of the artifact that represents the header file. |

The default values for the include protection properties are:

"#ifndef GEN_%g\n#define GEN_%g\n" for "includeProtectionBegin"

"#endif // GEN_%g" for"includeProtectionEnd"

**Note**

*Don't forget the trailing newline (\n) in the string specified for the property "includeProtectionBegin". Without it the generated code will be appended to the #define which typically leads to compilation errors.*

# General Translation Rules

This chapter describes general translation rules that apply to many kinds of translated entities.

## Name of definitions

**The name of a C++ definition is the same as the name of the UML definition from which it is translated.**

An exception occurs if the UML definition has an ANSI name specified (by means of the ansiName stereotype). Then the C++ definition will get that name instead.

**If no ANSI name is specified, or an ANSI name is specified that in fact is not ANSI, the C++ name is adjusted to become ANSI.**

If the C++ name to use (obtained from one of the translation rules above) is not a legal C++ identifier (because it is a keyword, or a name starting with a digit), it will be prefixed by a user-configurable prefix that by default is

“Name_”. If it instead is illegal because it contains characters not allowed in a C++ identifier (for example a whitespace), each such character will be replaced with an ASCII string representation of the illegal character.

**Example 497: Translation of names**

**UML**

```
class volatile {
}
<<ansiName(.name = "XYZ".)>> class '1åäö' {
}
<<ansiName(.name = "1åäö".)>> class C {
}
<<ansiName(.name = "1par".)>> class D {
}
part D 'an attribute';
```

**C++**

```
class Name_volatile {
};
class XYZ {
};
class tau_003100e500e400f6_tau {
};
class Name_1par {
};
D an_32_attribute;
```

# Type of typed definitions

## Impact of aggregation kind

**The type reference of a typed definition (that is a definition that has a type, for example an attribute, parameter, or a syntype) is translated to a corresponding type reference in C++.**

The rules are as follows:

**A typed UML entity of reference or shared type (the typed entity has “reference” or “aggregation” as aggregation kind) is translated into a C++ definition with:**

- a C++ pointer type specifier, if the type of the entity is not a datatype (or a syntype of a datatype).

- no type specifier at all, if the type of the entity is a datatype (or a syntype of a datatype).

**A typed UML entity of part type (the typed entity has "composition" as aggregation kind) is translated to no type specifier at all in C++ (the definition gets "value" type).**

An exception to this rule applies if the Multiplicity of the part is optional or non-single (compare Combined impact of multiplicity and aggregation kind).

**A UML entity typed by an instantiation of the `TTDCppPredefined::CPtr` template is translated into a C++ definition with one pointer type specifier for each (nested) instantiation of that template.**

**Example 498: Translation of type reference**

**UML**

```
class D {
    E a1;
    shared E a2;
    part E a3;
    CPtr<E> a4;
    CPtr<CPtr<E> > a5;
}
```

**C++**

```
class D {
  E* a1;
  E* a2;
  E a3;
  E* a4;
  E** a5;
};
```

### Predefined types

**References to UML predefined types are translated to references to C++ types according to the table below:**

| UML pre-defined type | C++ type | Comment |
|---|---|---|
| `Boolean` | `tor::Boolean` | Defined as '`bool`'. |
| `Character` | `tor::Character` | Defined as 'unsigned char' in ASCII compilation and as '`wchar_t`' in Unicode compilation. |
| `Integer` | `tor::Integer` | By default defined as 'int', but could optionally be defined as '`long int`', '`long long int`', or as an Integer class depending on how big integral numbers it needs to represent. |
| `Natural` | `tor::Natural` | By default defined as 'unsigned int', but could optionally be defined as '`unsigned long int`', '`unsigned long long int`', or as a Natural class depending on how big integral numbers it needs to represent. |
| `Real` | `tor::Real` | By default defined as 'double', but could optionally be defined as 'float', 'long double', 'long float', or as a Real class depending on how big real numbers (and with how big precision) it needs to represent. |

| UML pre-defined type | C++ type | Comment |
|---|---|---|
| `Charstring` | `tor::Charstring` | Defined as a class that inherits the STL class `std::string` (`wstring` in Unicode configuration), and which implements the standard collection interface (described below). |
| `String` | `tor::Charstring` | String is just a syntype of Charstring in UML. |
| `String` | `tor::String` | See Collections and impact of multiplicity. |

The target C++ types (tor::…) are defined in the TOR header torTypes.h.

**All fundamental C++ types are available at UML level through the 'TTDCppPredefined' profile package. Usages of these types are translated to the corresponding C++ fundamental type.**

**Example 499: Translation of predefined types**

**UML**

```
Boolean b;
Any a;
TTDCppPredefined::float f;
```

**C++**

```
tor::Boolean b;
tor::Any a;
float f;
```

### Collections and impact of multiplicity

**The predefined UML collection type is the String type. References to it are translated to references to a '`tor::String`' type, which is defined as a class that inherits a container class from the STL (`std::vector` by default).**

`tor::String` has an interface that corresponds closely to the interface of the predefined UML String type. The implementation of the interface calls functions inherited from the base class container.

**Example 500: Multiplicity**

Entities with Multiplicity greater than 1 are translated in the same way as entities typed by the String type.

**UML**

```
Character str[*];
String<Integer> istr;
class C {
  C [10] lst;
  C [1, 5..10] lst2;
}
```

**C++**

```
tor::String<tor::Character> str;
tor::String<tor::Integer> istr;
class C {
  tor::String<C*> lst;
  tor::String<C*> lst2;
};
```

**Note**

*The* `TTDCppPredefined` *package contains a* `CArray` *datatype, which corresponds directly to the built-in array construct of C/C++. It could be used as an alternative to String (for example for performance reasons).*

### Combined impact of multiplicity and aggregation kind

As described above, an attribute with "composite" aggregation kind (a part) is represented as a C++ value. Thus, the lifetime relationship between the part and its owner is automatically enforced by the C++ language (that is when an instance of the owning class is deleted, the part it owns will also be deleted).

However, this is only true if the part has single, non-optional Multiplicity (i.e. the multiplicity is exactly 1). In the case of non-single multiplicity, or optional multiplicity, the lifetime relationship is not automatically enforced by the C++ language. The C++ Application Generator thus generates additional code in this case to make sure the correct lifetime semantics is obtained.

**Note**

*The UML semantics of a part with non-single multiplicity is that it is the entities that are contained in the collection (and not the collection itself) that should have a lifetime relationship with the owning instance*

This is solved by the following translation rules:

**A part with non-single multiplicity is translated to an attribute typed by the `tor::String` type, where the element type is a pointer to the translation of the attribute type. If the attribute is typed by a datatype (or a syntype of a datatype), the type translation is used as is, without adding a pointer specifier. The same thing will happen if the element type is an instantiation of the predefined `Value` template.**

In addition, code is added to the destructor of the containing class. This code consists of one stack-allocated variable for each part. The variable is typed by `tor::MultiDeleter`, which is a class that deletes all elements of the String in its destructor. Thus it is guaranteed that when execution leaves the destructor, the elements of the String will be automatically deleted.

**Note**

*The code for implementing a container with part semantics will not be added in case the multiplicity is specified to be informal (see Informal multiplicity and custom container types).*

**Example 501:** ———————————————————————————

**UML**

```
class C {
  part C [*] c_list;
  part String<C> c_list2;
}
```

**C++**

```
class C {
  tor::String<C*> c_list;
```

```
  tor::String<C*> c_list2;

  ~C() {
    tor::MultiDeleter<C> deleter_c_list(c_list);
    tor::MultiDeleter<C> deleter_c_list2(c_list2);
  }
};
```

**A part with single optional multiplicity is translated to an attribute of pointer type. If the attribute is typed by a datatype (or a syntype of a datatype), the type translation is used as is, without adding a pointer specifier.**

Code will also be added in the constructors of the owning class for initializing the pointers to 0, and in the destructor to ensure that when it is deleted, pointers corresponding to parts with non-single multiplicity, will be deleted too. The destructor code is generated in a way similar as for parts with non-single multiplicity. The only difference is that the destructor variable is typed by the class `tor::SingleDeleter` instead.

**Example 502:**

**UML**

```
class C {
  part D [0..1] opt_d;
}
```

**C++**

```
class C {
  D* opt_d;

  C() : opt_d(0) { }

  ~C() {
    tor::SingleDeleter<D> deleter_opt_d(opt_d);
  }
};
```

The table below summarizes the translation of attributes with different combinations of aggregation kind, type and multiplicity:

| Multi-plicity | Part (typed by class) | Reference (typed by class) | Typed by datatype |
|---|---|---|---|
| `0..1` | Pointer variable<br>Initialization to 0 in constructor<br>Delete in destructor | Pointer variable | Value variable |
| `1` | Value variable | | |
| `*`<br>`0..n`<br>`n..m`<br>where m, n > 1 | Container of pointers variable<br>Delete of all container elements in destructor | Container of pointers variable | Container of values variable |

# Initial instances

**If an attribute specifies an initial number of instances, code will be added in the constructors of the owning class which will create the initial instances according to the table below:**

| Multiplicity / Initial cardinality | Part (typed by class) | Reference (typed by class) | Typed by datatype |
|---|---|---|---|
| `0..1 / 1` | Set the attribute to an initially created instance. | Set the attribute to NULL. | N/A |
| `m..n / m`<br>where m, n > 1 | Insert m number of initially created instances in the container. | Insert m number of NULLs in the container. | Insert m number of default literals of the datatype in the container. |

When no user-defined constructors exist, a default constructor will be created. The same translation rule apply if the multiplicity specifies a closed range (with the special case where the lower bound and upper bounds are equal, that is to say that one single value is specified as multiplicity), even if an initial number of instances has not been specified.

**Example 503: Translation of initial instances**

**UML**

```
class C {
  part D [0..1]/1 opt_d;
  part D [4]/2 d2;
  D [1] d3;
  D [8] d4;
  Integer [7] i1;
}
```

**C++**

```
class C {
  D* opt_d;
  tor::String<D*> d2;
  D* d3;
  tor::String<D*> d4;
  tor::String<tor::Integer> i1;

  C() : opt_d(new D), d2(true), d3(0), d4(8), i1(7) {
    tor::initString<D>(d2, 2);
  }
  ~C() {
    if (opt_d)
      delete opt_d;
  }
};
```

**Note**

*The constructor code for adding initial elements in the container will not be added in case the multiplicity is specified to be informal (see Informal multiplicity and custom container types for more information).*

## Informal multiplicity and custom container types

By specifying the Multiplicity of a structural feature as informal, it is possible to use a different container type than the default String container.

**If a structural feature has informal multiplicity the C++ Application Generator assumes it does not know the semantics of the specified container type. Thus it will not generate code for initializing the container with an initial number of instances (if an initial number of instances has been specified). Nor will it generate code for deleting all instances of the container (if the container is a part).**

**Example 504: Translation of an attribute with a custom container type**

**UML**

```
class A {
    public MyContainer<B> {[*]} m_b;
    public part MyContainer<B> {[*]} m_p / 2;
    public MyContainer<B> {[*]} m_r / 4;
}
```

**C++**

```
class A {
public:
    MyContainer<B*> m_b;
    MyContainer<B*> m_p;
    MyContainer<B*> m_r;
};
```

**Note**

*There is no constructor in* `A` *for initializing* `m_p` *with 2 initial instances of* `B` *and* `m_r` *with 4 NULL entries, and there is no destructor for deleting all instances of* `m_p`.

# Comments

**A Comment is translated to a C++ source code comment (formatted with the <MODEL> tag) placed just before the translation of the model element to which the comment is attached.**

If more than one comment is attached to a model element, the corresponding C++ comments are separated by an empty line in the generated code.

**Example 505:**

**UML**

```
package P1 comment "This is package P1" comment "ver. 1"
{
  class C {

    }
}
```

**C++**

```
/*<MODEL> This is package P1 */
```

```
/*<MODEL> ver. 1 */
namespace P1 {
  class C { ... };
};
```

If the translator option Support roundtrip is turned off, <MODEL> markers will not be printed.

## External definition

**An external Definition is translated to a corresponding C++ 'extern' declaration.**

The meaning of 'external' is thus the same both in UML and C++. An entity has to be declared as external in a UML model, if the entity is used in the model, but defined outside it (for example in a library).

**Example 506:**

**UML**

```
DD extern var;
```

**C++**

```
extern DD var;
```

**Note**

*The C++ Application Generator will sometimes generate 'extern' C++ definitions although the corresponding UML definition is not marked as external. For example, a non-constant attribute will be translated to a C++ variable which will be declared as 'extern' in the header file where it is manifested. For more information see Attribute.*

## Non-name based references

**If a reference is not setup by name (but GUID or pointer) it is converted to a name-based reference. A minimal relative qualifier is added to the reference to make sure it binds to the same target as before.**

This translation rule is currently applied to a limited set of references (those references that are typically edited using graphical syntax in the editors).

# Markers for synthesized entities

**All parts of a generated file which corresponds to entities that are synthesized during the UML to C++ translation are enclosed within <GENERATED> markers.**

**Example 507: Synthesized Entities**

**UML**

```
class X {
  const Integer i = 14;
}
```

**C++**

```
class X {
  const tor::Integer i;
// <GENERATED>
  public:
  X() : i(14) { }
// </GENERATED>
};
```

The <GENERATED> markers are intended as a help to indicate which parts of a file that have no direct equivalence in the UML model, but are synthesized during the translation. Avoid changing code in these sections, except for modifying the code formatting or adding comments.

**Note**

*For brevity reasons all markers are excluded from the examples in this document.*

If the translator option Support roundtrip is turned off, <GENERATED> markers will not be printed.

# Package

**A package is translated to a namespace.**

**If the package has the stereotype `TTDCppPredefined::globalNamespace` applied, it represents the global namespace of C++. No namespace will then be generated for it.**

**Note**

*If this stereotype is applied on more than one Package in a model, it is your responsibility to make sure that names within all these packages are unique. It is therefore recommended to use at most one Package for representing the global namespace.*

**Example 508:**

**UML**

```
package P1 {
  class C {

    }
}
<<globalNamespace>> package P2 {
    class C {

    }
}
```

**C++**

```
namespace P1 {
  class C {  };
};
class C { };
```

# Dependency

Dependencies are used in many ways in a UML model and are often to be interpreted as informal. However, there are certain specific usages of dependencies that are visible in generated C++ code. These usages of dependencies are described in this chapter. Dependencies that do not fall into the categories mentioned below are not translated to C++.

# Include dependency

**A dependency stereotyped by the predefined 'include' stereotype, where the supplier of the dependency is an artifact representing a file, is translated to a #include of that file.**

If the client of the dependency is another artifact representing a file, the #include directive is generated at the beginning of that file.

**Example 509:**

**UML**

```
<<cppHeaderfile (. path = "sif.h".)>> artifact
SomeIncludeFile;

<<cppHeaderfile (. path = "..\..\include\generic.h".)>>
artifact GenericTypes <<include>> dependency to
SomeIncludeFile
<<manifest>> dependency to X;

class X {
}

<<cppHeaderfile (. path = "IRunnable_interface.h".)>>
artifact IRun_ifc
<<manifest>> dependency to IRunnable;

interface IRunnable <<include>> dependency to
GenericTypes {
}
```

**C++** (in file "`..\..\include\generic.h`")

```
#include "sif.h"

class X {
};
```

**C++** (in file "IRunnable_interface.h")

```
#include "..\..\include\generic.h"
UML_INTERFACE IRunnable {
};
```

It is not necessary to manually specify dependencies for each required #include. The C++ Application Generator will automatically compute the minimal set of #includes needed by analyzing in which files definitions that are

used from one compilation unit will be generated. However, the mechanism for specifying a #include dependency manually is needed in the following situations:

1. When definitions from a header file is used in inline C++ code, since such code is not interpreted at UML level.
2. In order to include external source files in C++ implementations.
3. To specify the use of a precompiled header for a C++ implementation file

Example 510 on page 1557 shows how to specify the use of a precompiled header for an implementation file.

**Example 510:**

**UML**

```
<<cppHeaderfile (. path = "StdAfx.h", precompiled = true
.)>> artifact PrecompiledHeader;

<<cppImplementationFile (. path = "foo.cpp" .)>>
artifact foo_impl <<manifest>> dependency to foo,
<<include>> dependency to PrecompiledHeader;

bool foo(){
  return false;
}
```

**C++** (in file “foo.cpp”)

```
#include "StdAfx.h"

bool foo(){
  return false;
}
```

# Access dependency

**A dependency stereotyped by the predefined «access» stereotype, where the supplier of the dependency is a package P (not stereotyped by the «'global namespace'>> stereotype), is translated to a 'using namespace N', where N is the translation of P. If the supplier of the dependency is another kind of entity, a 'using' declaration is generated instead.**

**Example 511:**

**UML**

```
package P1 <<access>> dependency to P2
<<access>> dependency to P3::E {
  C var;
}

package P2 {
  class C { }
  class D { }
}

package P3 {
  class E { }
}
```

**C++**

```
namespace P1 {
  using namespace P2;
  using P3::E;

  C var;
};

namespace P2 {
  class C {};
  class D {};
};

namespace P3 {
  class E {};
};
```

## Import dependency

**A dependency stereotyped by the predefined «import» stereotype, where the supplier of the dependency is a package (not stereotyped by the «'global namespace'>> stereotype), is translated to a 'using namespace' for the namespace that corresponds to the supplier package.**

**Note**

*This means that the C++ Application Generator will treat import dependencies exactly like access dependencies.*

**Example 512:**

**UML**

```
package P1 <<import>> dependency to P2 {
}
```

**C++**

```
namespace P1 {
  using namespace P2;
};
```

## Friend dependency

**A dependency stereotyped by the «friend» stereotype from TTDCppPredefined, where the client of the dependency is a Class or Choice, is translated to a friend declaration.**

The friend declaration is placed in the C++ translation of the Class or Union.

**Example 513:**

**UML**

```
class Class1 <<friend>> dependency to foo(Integer) {}
void foo( Integer);
```

**C++**

```
void foo(tor::Integer par0);
class Class1 {
    friend void ::foo(tor::Integer par0);
```

```
};
```

# Structured Classifier

A Structured Classifier is translated according to the following rules:

**A Class is translated to a class, or to a struct if it has the stereotype `TTDCppPredefined::struct` applied.**

**An Interface is translated to an abstract class, that is a class with all its member functions being pure virtual.**

**A `choice` is translated to a union.**

Other kinds of structured classifiers are not translated.

**Example 514:**

**UML**

```
class C { }
<<struct>> class S { }
interface I {
  int foo( char);
}
choice U1 {}
```

**C++**

```
class C { };
struct S {};
UML_INTERFACE I {
  int foo(char) = 0;
}
union U1 {};
```

`UML_INTERFACE` is a macro defined in the TOR header torAnnotations.h. It is defined to `class`. If the translator option Support roundtrip is turned off, the `class` keyword is used instead of this macro.

# Attribute

A non-constant attribute is translated according to the following rules. For constant attributes see Constant attribute.

**An Attribute owned by a Structured Classifier is translated to a member variable of the class, struct or union that is the translation of the structured classifier.**

**An Attribute owned by a Package is translated to a variable scoped by the namespace that is the translation of the package.**

Unless an attribute in a package is declared as "external" it will yield both a variable declaration (in the header file generated for the package) and a variable definition (in the source file generated for the package), if the variable is manifested in a C++ implementation file that is. The variable declaration in the header file will always be external to allow the header file to be included from several different compilation units without link errors.

**Example 515:**

**UML**

```
package test {
    class S {
        Boolean m_b;
    }
    Integer var;
}
```

**C++** (in file "test.h")

```
namespace test {
  class S {
    tor::Boolean m_b;
  };
  extern tor::Integer var;
};
```

**C++** (in file "test.cpp")

```
tor::Integer test::var;
```

# Attribute default value

**The default value of a non-static Attribute owned by a Structured Classifier is translated to an initialization of the corresponding member variable in the initializer list of all constructors.**

**The default value of an Attribute owned by a Package, or a Static attribute, is translated to an initialization of the corresponding variable definition.**

**Example 516:** ———————————————————————————————

**UML**

```
package test {
    class S {
        Boolean m_b = true;
        public S() {
           // Some initialization
        }
    }
    Integer var = 14;
}
```

**C++** (in file test.h)

```
namespace test {
  external tor::Integer var;
};
```

**C++** (in file test.cpp)

```
tor::Integer test::var = 14;
```

**C++** (in file S.h)

```
namespace test {
  class S {
    tor::Boolean m_b;
    public:
    S();
  };
};
```

**C++** (in file S.cpp)

```
test::S::S() : m_b(true) {
```

```
}
```

**Note**

*When an attribute of a structured classifier has a default value, but there are no constructors defined for the structured classifier, then an explicit parameter-less constructor with public visibility will be created for the C++ class, in order to specify the initialization of the member variable.*

A default value specified for an attribute implies an Assignment, so the translation rule for assignments applies also in this case.

## Attribute visibility

**The visibility of an Attribute owned by a Structured Classifier is translated to a corresponding visibility for the corresponding member variable.**

Public, protected and private visibility has the same semantics in UML and C++. However, UML has an additional visibility kind known as package visibility.

**In the C++ translation package visibility is translated as public visibility.**

If the visibility is unspecified in UML, it defaults to public or private visibility (depending on the context). In the translation the visibility is explicitly specified anyway, if required by the rules of C++ (also in C++ the default visibility is dependent on context). For example, an attribute with unspecified visibility defined in a class will be private in C++.

A small difference between UML and C++ is that in UML the visibility is specified for every attribute while in C++ the visibility can be specified once for all the member variables that follows. Since the latter approach often is considered improving code readability, several consecutive attributes with the same visibility will only yield one “visibility declaration” for the corresponding C++ member variables.

**Example 517:**

**UML**

```
class S {
    public Boolean m_b;
    public Character m_c;
```

```
    protected int m_i;
    private bool m_bo;
    package char m_ch;
}
```

**C++**

```
class S {
public:
  tor::Boolean m_b;
  tor::Character m_c;
protected:
  int m_i;
private:
  bool m_bo;
public:
  char m_ch;
};
```

---

## Static attribute

**A static attribute owned by a Structured Classifier is translated to a corresponding static member variable.**

**Example 518:** ———————————————————————————————

**UML**

```
class S {
    Boolean m_b1;
    static Boolean m_b2;
}
```

**C++** (in S.h)

```
class S {
  tor::Boolean m_b1;
  static tor::Boolean m_b2;
};
```

**C++** (in S.cpp)

```
tor::Boolean S::m_b2;
```

---

## Constant attribute

**A constant Attribute owned by a Structured Classifier is translated to a member constant.**

**A constant Attribute owned by a Package is translated to a non-member constant.**

Note that contrary to a variable a C++ constant by default has internal linkage. To specify that a constant attribute shall be translated to a C++ constant with external linkage, the 'external' property should be set to true on the UML attribute.

**Example 519:**

**UML**

```
<<globalNamespace>> package P {
  class S {
      const Boolean m_b1 = true;
  }
  const double e = 2.78; // Internal constant
  const Integer extern b = 6; // External constant
}
```

**C++** (in header file)

```
class S {
  const tor::Boolean m_b1;
  S() : m_b1(true) {}
};
const double e = 2.78;
extern const tor::Integer b;
```

**C++** (in implementation file)

```
const tor::Integer b = 6;
```

**Note**

*This translation rule can be combined with the one for static attributes to yield a constant member in C++.*

## Bitfield

**An Attribute with the `TTDCppPredefined::bitfield` stereotype applied is translated to a bitfield.**

**Example 520:**

**UML**

```
<<struct>> class S {
    int <<bitfield(. 2 .)>> m1;
    bool <<bitfield(. 1 .) m2;
}
```

**C++**

```
struct S {
  int m1 : 2;
  bool m2 : 1;
};
```

The size of the bitfield is specified as a tagged value.

# Operation

**An Operation owned by a Structured Classifier is translated to a member function of the class, struct or union that is the translation of the structured classifier.**

**An Operation owned by a Package is translated to a function scoped by the namespace that is the translation of the package.**

**Note**

*Constructors and destructors are special kinds of operations both in UML and C++. They are consequently translated according to the same rules as ordinary operations (a constructor called 'initialize' in UML will be called the same name as the owner class, struct or union according to the rules of C++).*

**Example 521:**

**UML**

```
package test {
```

```
        class S {
            int foo();
            S(bool); // Constructor
            initialize(); // Constructor
            finalize(); // Destructor
        }
        Integer open();
    }
```

**C++**

```
    namespace test {
      class S {
        int foo();
        S(bool);
        S();
        ~S();
      };
      tor::Integer open();
    };
```

---

# Operation parameters

**A Parameter to an Operation is translated to a formal parameter of the function that is the translation of the operation.**

**Parameters with direction kind 'in/out' or 'out' are translated to parameters of reference type.**

**The type of the first parameter with direction kind 'return' (there should at most be one) is translated to a return type for the function that is the translation of the operation. If there is no return parameter, a void function is generated.**

**Example 522:** ———————————————————————————————

**UML**

```
    int foo(int p1, in int p2, inout int p3, out int p4);
```

**C++**

```
    int foo(int p1, int p2, int& p3, int& p4);
```

The general rules for typed entities (compare with "Type of typed definitions" on page 1543) apply also on the return parameter.

---

### Parameter default value

**A default value for an Operation Parameter is translated to a corresponding default value for the function parameter that is the translation of the Parameter.**

**Note**

*When a parameter with a default value is a non-trailing parameter it is not possible to give the corresponding C++ parameter a default value, according to the rules of C++.*

**Example 523:** ---

**UML**

```
int foo(int p1 = 4, int p2, int p3 = 5);
MyClass bar();
part MyClass func();
```

**C++**

```
int foo(int p1, int p2, int p3 = 5);
MyClass* bar();
MyClass func();
```

---

The default value may be any Expression.

A default value specified for a parameter implies an Assignment, so the translation rule for assignments applies also in this case.

### Parameter multiplicity

The impact of a multiplicity specified for a parameter is in general the same as for any definition (see "Collections and impact of multiplicity" on page 1547). However, parameter multiplicity is also used to specify that a parameter is optional (0 is then included in the specified multiplicity ranges).

**Note**

*When an optional parameter is a non-trailing parameter it is not possible to give the corresponding C++ parameter a default value, according to the rules of C++.*

**Example 524:**

**UML**

```
void f(int p1[0..1], int p2, MyClass p3[0..1], MyClass
p4[0..*]);
```

**C++**

```
void f(int p1, int p2, MyClass* p3 = 0,
       String<MyClass*>* p4 = 0);
```

## Abstract operation

**An abstract Operation owned by a Structured Classifier is translated to a pure virtual member function of the class, struct or union that is the translation of the structured classifier.**

Other abstract operations are translated as ordinary operations.

**Example 525:**

**UML**

```
class S {
    abstract int f1();
}
class D : S {
    redefined int f1(); // Redefines S::f1
}
```

**C++**

```
class S {
  virtual int f1() = 0;
};
class D : public S {
  virtual int f1();
};
```

# Virtual, redefined or finalized operation

**A virtual, redefined or finalized Operation owned by a Structured Classifier is translated to a virtual member function of the class, struct or union that is the translation of the structured classifier.**

The specification that an operation is redefined or finalized is thus not visible in the C++ translation.

**Example 526:**

**UML**

```
class S {
    int f1();
    virtual int f2();
    virtual int f3();
    virtual int f4();
}
class D : S {
    int f1(); // Hides S::f1
    virtual int f2(); // Hides S::f2
    redefined int f3(); // Redefines S::f3
    finalized int f4(); // Redefines S::f4
}
```

**C++**

```
class S {
  int f1();
  virtual int f2();
  virtual int f3();
  virtual int f4();
};
class D : public S {
  int f1();
  virtual int f2();
  virtual int f3();
  virtual int f4();
};
```

## Exception specification

**An exception specification (a 'throw' declaration) for an Operation, is translated into an exception specification for the function that is the translation of the operation.**

**Example 527:**

**UML**

```
void foo() throw Exc1, Exc2;
void <<TTDCppPredefined::noException>> bar();
```

**C++**

```
void foo() throw (Exc1, Exc2);
void bar() throw ();
```

As shown in Example 527 on page 1571 there is a special stereotype `TTDCppPredefined::NoException` which can be used to specify that an operation throws no exceptions at all.

## Operation reference

**An interface representing an operation reference is translated to a typedef of a function pointer type corresponding to the signature of the 'call' operation of the interface.**

Calls to the 'call' operation are translated into calls on the corresponding function pointer.

**Example 528:**

**UML**

```
<<operationReference>> interface IFoo {
    Integer call(Boolean);
}

Integer foo(Boolean);

Integer main() {
    IFoo i = operation foo(Boolean);
    return i.call(true);
}
```

**C++**

```
typedef tor::Integer (*IFoo)(tor::Boolean);
tor::Integer foo(Boolean);

tor::Integer main() {
    IFoo i = foo;
    return i(true);
}
```

# Generalization

**A Generalization between two Structured Classifiers is translated to an inheritance between the classes, structs or unions that are the translation of the structured classifiers.**

**Note**

*For* `EventClasses` *(for example Signals) special translation rules for generalization apply.*

Generalizations between operations are not translated.

By default the inheritance will be public and non-virtual, but by applying the Stereotypes `TTDCppPredefined::inheritanceVisibility` and/or `TTDCppPredefined::virtualInheritance` it is possible to specify also private or protected inheritance and virtual inheritance.

**Example 529:**

**UML**

```
class S {
}
class D : S {
}
```

**C++**

```
class S {
};
class D : public S {
};
```

# Association

Unnamed uni-directional associations are represented as attributes in the UML model.
The translation of such attributes thus follows the rules in Attribute.

**Example 530:**

**UML**

```
class T {
    U b1;
}
class U {

}
association A {
    from T a2;
    from U a1;
    to a1;
    to a2;
}
association B {
    from T b2;
    to b1;
    to b2;
}
```

**C++**

```
class T {
  U* b1;
};
class U {
};
class A {
  T* a2;
  U* a1;
};
class B {
  T* b2;
};
```

# Syntype

**A syntype is translated to a C++ type definition.**

The data constraint of the syntype will not be translated to C++.

**Example 531:**

**UML**

```
syntype X = Integer;
```

**C++**

```
typedef tor::Integer X;
```

# Datatype

A datatype is mapped to a plain enum. This means that if the datatype has operations, these are not translated to C++.

# Informal Definition

**An informal definition is translated into a verbatim copy of the definition text.**

If the informal definition contains a reference to a UML definition, it is translated according to the ordinary rule for an Identifier before the expression is copied into the generated C++.

**Example 532:**

**UML**

```
[[int status = reinterpret_cast<#(Integer)>(STATUS);]];
```

**C++**

```
/*<TARGET>*/
int status = reinterpret_cast<tor::Integer>(STATUS);
/*</TARGET>*/
```

# Expression

**UML Expressions are translated to C++ by translating each part of the expression individually. Constant expressions are not evaluated during translation (although it would be possible in most cases).**

The translation of most UML expression is straight-forward, as shown in the table below:

| UML Expression | C++ Expression | UML Example | C++ Example |
|---|---|---|---|
| Parenthesis expression | Parenthesis expression | `(a+b)` | `(a+b)` |
| Unary expression | A unary expression, where the C++ operator to use is decided by the specified UML operator (see "References to predefined UML definitions" on page 1577). | `not m_bOk`<br>`++var` | `!m_bOK`<br>`++var` |
| Binary expression | A binary expression, where the C++ operator to use is decided by the specified UML operator (see "References to predefined UML definitions" on page 1577). | | |
| This expression | 'this' expression | | |
| Index expression | Subscripting operator ('[]') | `coll[4]` | `coll[4]` |
| Create expression | 'new' operator | `new C(1,2)` | `new C(1,2)` |

| UML Expression | C++ Expression | UML Example | C++ Example |
|---|---|---|---|
| Conditional expression | Conditional expression | `b ? x1 : y1` | `b ? x1 : y1` |
| Real value | real literal | `Real a = 3.14;` | `tor::Real a = 3.14;` |
| Integer value | integer literal | `Integer a = 4;` | `tor::Integer a = 4;` |

The translation of remaining expressions is described in the rest of this chapter.

## Identifier

**An identifier is translated in the same way as the name of the definition to which it is bound.**

This rule applies both when the identifier is part of an expression and when it represents a reference (compare Name of definitions).

**Example 533:**

**UML**

```
<<ansiName(.name = "XYZ".)>> class '1ääö' {
    public void g();
}
'1ääö' volatile = new '1ääö'();
void foo(){
    volatile.g();
}
```

**C++**

```
class XYZ {
};
XYZ* Name_volatile = new XYZ();
void foo(){
  Name_volatile->g();
}
```

The translated identifier will contain scope qualifiers if the UML identifier has scope qualifiers.

**Note**

*As a consequence of somewhat different scope rules, a scope qualifier is sometimes added in C++ although it would not be needed in UML. The opposite could also happen; if a package stereotyped by «*`globalNamespace`*» is referenced in the scope qualifier, that reference is removed.*

### References to predefined UML definitions

**An exception to the rule above applies when the identifier represents a reference to a predefined UML entity (an entity contained in the predefined package).**

The reason is that the predefined package is not translated to C++. If the referenced definition is a type, the translation follows the rule in "Predefined Types". If the referenced definition is an operation, the translation is made according to the table below.

| Predefined UML operation | C++ operator |
|---|---|
| `reinterpret_cast` | `reinterpret_cast` |
| `ASSERT` | `ASSERT`<br>`It is assumed that this function (or macro) is defined in a file included by the user.` |
| `Boolean::'='` | `operator=(tor::Boolean, tor::Boolean)` |
| `Boolean::not` | `operator!(tor::Boolean)` |
| `Boolean::and`<br>`Boolean::'&&'` | `operator&&(tor::Boolean, tor::Boolean)` |
| `Boolean::or`<br>`Boolean::'\|\|'` | `operator\|\|(tor::Boolean, tor::Boolean)` |
| `Boolean::equal`<br>`Boolean::'=='` | `operator==(tor::Boolean, tor::Boolean)` |
| `Boolean::'!='` | `operator!=(tor::Boolean ,tor::Boolean)` |
| `Boolean::xor` | `operator^(tor::Boolean, tor::Boolean)` |
| `Boolean::'=>'` | `tor::implies(tor::Boolean, tor::Boolean)` |
| `Integer::'='` | `operator=(tor::Integer, tor::Integer)` |

| Predefined UML operation | C++ operator |
|---|---|
| Integer::'-' | operator-(tor::Integer) |
| Integer::'-' | operator-(tor::Integer, tor::Integer) |
| Integer::'+' | operator+(tor::Integer) |
| Integer::'+' | operator+(tor::Integer, tor::Integer) |
| Integer::'++' | operator++(tor::Integer) |
| Integer::'--' | operator--(tor::Integer) |
| Integer::'*' | operator*(tor::Integer, tor::Integer) |
| Integer::'/' | operator/(tor::Integer, tor::Integer) |
| Integer::mod | tor::mod(tor::Integer, tor::Integer) |
| Integer::rem | operator%(tor::Integer, tor::Integer) |
| Integer::power | tor::power(tor::Integer, tor::Integer) |
| Integer::equal<br>Integer::'==' | operator==(tor::Integer, tor::Integer) |
| Integer::'!=' | operator!=(tor::Integer, tor::Integer) |
| Integer::'<' | operator<(tor::Integer, tor::Integer) |
| Integer::'>' | operator>(tor::Integer, tor::Integer) |
| Integer::'>=' | operator>=(tor::Integer, tor::Integer) |
| Integer::'<=' | operator<=(tor::Integer, tor::Integer) |
| Character::'=' | operator=(char, char) |
| Character::'==' | operator==(tor::Character,<br>tor::Character) |
| Character::'!=' | operator!=(tor::Character,<br>tor::Character) |
| Character::'<' | operator<(tor::Character,<br>tor::Character) |
| Character::'>' | operator>(tor::Character,<br>tor::Character) |
| Character::'<=' | operator<=(tor::Character,<br>tor::Character) |
| Character::'>=' | operator>=(tor::Character,<br>tor::Character) |

| Predefined UML operation | C++ operator |
|---|---|
| `Real::'='` | `operator=(tor::Real)` |
| `Real::'-'` | `operator-(tor::Real, tor::Real)` |
| `Real::'+'` | `operator+(tor::Real, tor::Real)` |
| `Real::'*'` | `operator*(tor::Real, tor::Real)` |
| `Real::'/'` | `operator/(tor::Real, tor::Real)` |
| `Real::'=='` | `operator==(tor::Real, tor::Real)` |
| `Real::'!='` | `operator!=(tor::Real, tor::Real)` |
| `Real::'<'` | `operator<(tor::Real, tor::Real)` |
| `Real::'>'` | `operator>(tor::Real, tor::Real)` |
| `Real::'<='` | `operator<=(tor::Real, tor::Real)` |
| `Real::'>='` | `operator>=(tor::Real, tor::Real)` |
| `is` | `tor::is(ARG arg)` |
| `as` | `tor::as(ARG arg)` |

**Note**

*The implementations of tor::is and tor::as are based on the C++* `dynamic_cast` *operator. Thus it is required that ARG is a polymorphic type (i.e. has at least one virtual operation). There is currently no semantic check that will detect attempts to use these operations on non-polymorphic types. If your argument types are non-polymorphic you can always add an empty virtual operation in order to make them polymorphic*

If the referenced definition is a constant, the translation is made according to the table below.

| Predefined UML constant | C++ constant |
|---|---|
| `Real PLUS_INFINITY` | `PLUS_INFINITY` It is assumed that this constant (or macro) is defined in a file included by the user. |
| `Real MINUS_INFINITY` | `MINUS_INFINITY` It is assumed that this constant (or macro) is defined in a file included by the user. |

**Note**

*When the reference has a scope qualifier containing a reference to the Predefined package explicitly (for example* `Predefined::Integer`*), that reference will be removed in the C++ transformation.*

### References to predefined C++ definitions

**Another exception to the identifier translation rule above applies when the identifier represents a reference to a predefined C++ entity (an entity contained in the** `TTDCppPredefined` **package).**

For most of the definitions in `TTDCppPredefined` the ordinary translation rule applies (since most have the same name as the corresponding built-in definitions in C++).

The exceptional cases are some operations listed in the table below:

| TTDCppPredefined operation | C++ operator | UML Example | C++ Example |
|---|---|---|---|
| `CPtr::'new[]'` | `new[] operator` | `CPtr<int> p = CPtr<int>::'new[]'(9);` | `int* p = new int[9];` |
| `CPtr::'delete[]'` | `delete[] operator` | `CPtr<int>::'delete[]'(p);` | `delete p[];` |

In addition to these operations, references to the `CPtr` and `CArray` templates are translated into C++ pointer and array type specifiers (compare with Example 533 on page 1576 and "Impact of aggregation kind" on page 1543).

**Note**

*When the reference has a scope qualifier containing a reference to the* `TTDCppPredefined` *package explicitly (for example* `TTDCppPredefined::int`*), that reference will be removed in the C++ transformation.*

**References to external C/C++ definitions**

**Yet another exception to the identifier translation rule applies when the identifier represents a reference to an external C/C++ entity (for example imported into the UML model by C/C++ Import), which has a different name in UML than in the target code.**

Since UML can support any name for an identifier, the effect of this exception is only to remove the enclosing apostrophes if a name contains these.

## Informal expression

**An informal expression is translated into a verbatim copy of the expression text.**

If the informal expression contains a reference to a UML definition, it is translated according to the ordinary rule for an Identifier before the expression is copied into the generated C++.

**Example 534:**

**UML**

```
int ptrSize = [[sizeof(void*)]];
int p2 = [[#(ptrSize) * 8 - 4]];
```

**C++**

```
int ptrSize = /*<TARGET>*/ sizeof(void*) /*</TARGET*>/;
int p2 = /*<TARGET>*/ ptrSize * 8 - 4 /*</TARGET*>/;
```

## Call expression

**A UML call expression is translated to a C++ function call.**

The actual arguments in a call expression are in general any Expression.

**Example 535:**

**UML**

```
void f(int p2, int p3, MyClass p4[0..1]);
void foo(){
    f(4, 3, new MyClass());
}
```

**C++**

```
void f(int p2, int p3,
       MyClass* p4 = 0);
void foo(){
  f(4, 3, new MyClass());
}
```

An operation call causes implicit assignments of actual arguments to formal operation parameters. The translation rule for an Assignment thus apply also in this case.

## Field expression

**A UML Field expression is translated to a call of the C++ member access operator '.' (if the operand type is a reference or value type in C++) or '->' (if the operand type is a pointer type in C++).**

## Assignment

An assignment is represented as a Binary expression and is translated as such. However, some additional translation rules apply for assignments, and these are described here.

**If the translation of the right-hand side of a UML assignment has value type in C++ while the translation of the left-hand side has pointer type in C++, the address-of operator (&) is applied on the right-hand side of the corresponding C++ assignment. Similarly, if the translation of the right-hand side of a UML assignment has pointer type in C++, while the translation of the left-hand side has value type in C++, the indirection operator (*) is applied on the right-hand side of the corresponding C++ assignment.**

This translation rule makes sure that an assignment between a part and a reference (or vice versa) that is legal in UML also will be legal in C++.

**Example 536:**

**UML**

```
C c = new C();
part C cv;
cv = c;
C cref;
cref = cv;
```

**C++**

```
C* c = new C();
C cv;
cv = *c;
C* cref;
cref = &(cv);
```

This translation rule is applicable also on implicit assignments:

1. The assignment of a default value to an attribute or a parameter.
2. The assignment of an actual operation argument to a formal operation parameter in an operation call. This also applies for assignment of an actual constructor argument to a formal constructor parameter.
3. The assignment of a return value to an operation return parameter in a return action.

## Charstring and Character values

**A Charstring or Character value is by default enclosed in the _T macro. This macro is defined in the TOR header "torTypes.h", and expands to nothing in an ASCII configuration, and to L in a wide-character (e.g. Unicode) configuration. The purpose is thus to allow the generated code to be compiled in both ASCII and non-ASCII configurations.**

There is a translation option Enable non-ASCII compilation that can be turned off (it is turned on by default) to avoid adding the _T macro around Charstring and character values. With this option turned off it is still possible to use multibyte strings using the wchar_t type.

**Example 537:**

**UML**

```
Charstring s = "zenith";
wchar_t[*] str = "angst";
Character c = 'x';
wchar_t cc = 'y';
```

**C++**

```
tor::Charstring s = tor::Charstring(_T("zenith"));
wchar_t str[] = L"angst";
tor::Character = _T('x');
wchar_t cc = L'y';
```

As seen in Example 537 on page 1584, a Charstring value is translated to a construction of a `tor::Charstring` object (possibly after enclosing it in the _T macro). The reason is that the type of a UML Charstring literal should be `tor::Charstring` and not `const TCHAR*` as would otherwise be the case.

Without the above translation rule usage of binary operators, where both the right-hand side and the left-hand side are typed by const TCHAR* in C++, would lead to application of the wrong operator. For example, comparison would be between the pointer values rather than between the contents of the UML Charstring variable values. The explicit construction of a `tor::Charstring` object for each UML Charstring literal solves this problem.

## TimerActive expression

**A TimerActive expression is translated to a call of the `isActive` function on the timer variable corresponding to the timer that is referenced by the expression.**

**Example 538:**

**UML**

```
timer T1 (Integer i);

void foo() {
  if (active(T1 (2))) {
    ...
  }
}
```

**C++**

```
class T1 : public virtual tor::TimerEvent {
public:
  typedef tor::TimerEvent theTimerEvent;
  tor::Integer i;
  T1(tor::Integer i) : i(i) {}

  static inline bool isTypeOf(tor::Event* e) {
    return tor::cast<T1*>(e) != 0;
  }

};

tor::TimerObject timer_T1(*this);

void foo() {
  if (timer_T1.isActive()) {
    ...
  }
}
```

**Note**

*The only purpose of the actual arguments that is used in the TimerActive Expression in UML is to identify which (possibly overloaded) version of a timer that shall be queried for its activeness. These actual arguments are thus not visible in the C++ translation.*

# Template

**A UML Structured Classifier Template is translated to a corresponding C++ class, struct or union template.**

**Example 539:**

**UML**

```
template <class T, const int x >
class CSL
{
    T mv = x;
    T op1(int p1 = x);
}
```

**C++**

```
template <class T, int x >
class CSL {
  T mv;
  CSL() : mv(x) {}
  T op1(int p1 = x);
};
```

**A UML Operation Template is translated to a C++ function template.**

**Example 540:**

**UML**

```
template <class T, T x >
void foo(T p1 = x);
```

**C++**

```
template <class T, T x >
void foo(T p1 = x);
```

Inside a template definition the normal translation rules apply. However, when translating references to formal template parameters those translation rules that depend upon knowledge about properties of the corresponding actual template parameter in a template instantiation are disabled. This is because it is in general impossible to know, from the template definition, how the template will be instantiated. One example of such a translation rule that is disabled for references to formal template parameters is the rule to add a

pointer declarator on UML references (see "Impact of aggregation kind" on page 1543). By specifying an Atleast constraint on a formal template parameter you can put a constraint on the corresponding actual template parameter, so that references to the formal template parameter can be translated in a more appropriate way.

## Template instantiation

**An instantiation of a UML template is translated to an instantiation of the C++ template that is the translation of the UML template.**

Actual template type parameters are ordinary references to types, so the rules in Impact of aggregation kind for adding pointer declarators apply also in this case. In order to pass a reference type as an actual template parameter by "value" rather than by reference, the predefined `Value` template can be used.

**Example 541:** ———————————————————————————

This example instantiates the template `CSL` from Example 539 on page 1586.

**UML**

```
CSL<MyClass, 4> v1;
CSL<Value<MyClass>, 4> v2;
```

**C++**

```
CSL<MyClass*, 4> v1;
CSL<MyClass, 4> v2;
```

———————————————————————————————————————

Another way to avoid the addition of a pointer declarator for an actual template type parameter is to specify in the definition of the corresponding formal template parameter that it is a reference type. This is done by using an Atleast constraint.

## Atleast constraint

**If a formal template parameter has an 'atleast T' constraint, where T is a type, references to that parameter within the template definition will be treated like a reference to T.**

**Example 542:**

By specifying ‘atleast Any’ on a formal template parameter the corresponding actual template parameter is constrained to be a reference type. Thus the C++ Application Generator can translate a reference to such a formal template parameter as it would translate a reference to a reference type (i.e. to add a pointer declarator).

**UML**

```
template <type Type atleast Any>
class buffer {
    public Type [*] variable;
    public <<inline>> buffer(Natural n) {
        for (Natural i = 0; i < n; i++) {
            Type t = new Type();
            variable.append(t);
        }
    }
}
```

**C++**

```
template <class Type>
class buffer {
    public:
    tor::String<Type *> variable;
    inline buffer(tor::Natural n) {
        for (tor::Natural i = 0; i < n; i++) {
            Type *t = new Type;
            variable.append(t);
        }
    }
};
```

Without the ‘atleast Any’ constraint on `Type`, no pointer declarator would have been added on references to `Type` within the template definition.

An ‘atleast’ constraint may also have an impact on the translation of a Template instantiation.

**Note**

*UML has several more features in its template construct than the ones described above. For example it is possible to specify a prototype for a template argument. These constructs have no correspondence in C++ and are not translated.*

# Action

**A UML action is translated to a C++ statement.**

The translation of most UML actions is straight-forward, as shown in the table below:

| UML Action | C++ Statement | UML Example | C++ Example |
|---|---|---|---|
| Delete action | delete statement | `delete T;` | `delete T;` |
| Compound action | compound statement | `{ v = v + 1; }` | `{ v = v + 1; }` |
| Continue action | continue statement | | |
| Break action | break statement | | |
| If action | if statement | | |

**If a Label is attached to an Action, it is translated to a C++ label just before the statement that is the translation of the action.**

**Example 543:**

**UML**

```
void foo(){
    LABEL: Integer v;
    goto LABEL;
}
```

**C++**

```
void foo(){
  LABEL:
  tor::Integer v;
  goto LABEL;
}
```

# Definition Action

**A Definition Action where the associated definition is an Attribute is translated to a local variable definition.**

**Example 544:**

**UML**

```
void foo(){
    Integer v = 4;
}
```

**C++**

```
void foo(){
  tor::Integer v = 4;
}
```

If the associated definition is not an Attribute, the definition action will not be translated. The C++ Application Generator will yield a warning in this case:

```
Warning: Local definitions other than local variables
are not supported. The local definition 'name' will not
be translated to C++.
```

# Expression Action

The translation of an Expression Action depends on what kind of expression that is associated with the action.

**An Expression Action with an associated Empty Expression, is translated to an empty C++ statement.**

**An Expression Action with an associated Informal expression, is translated to a copy of the informal text.**

**An Expression Action with an associated Call expression, is translated by appending a semicolon (;) to the translation of the call expression.**

**An Expression Action with an associated Create Expression, is translated by appending a semicolon (;) to the translation of the create expression.**

**Example 545:**

**UML**

```
void foo(){
     ;
     [[##ifdef _WIN32]];
     open(true);
     new Integer;
     [[##endif]];
}
```

**C++**

```
void foo(){
  ;
#ifdef _WIN32
  open(true);
  new tor::Integer;
#endif
}
```

As can be seen in Example 545 on page 1591 the normal rules for indentation are overruled for the case with an informal expression starting with '#' followed by a character other than '(' (since the informal text then specifies a preprocessor directive).

# Try Action

**A Try Action with an associated catch clause, is translated to a try statement with a catch clause.**

# Throw Action

**A Throw Action is translated to a throw statement.**

The expression of a Throw Action specifies which exception to throw. It can either

- specify a new exception to throw
- specify an already thrown-and-caught exception to “re-throw”.

**Example 546:**

**UML**

```
void foo(){
    try {
        compute();
    }
    catch (tor::EMultiplicityOutOfRange e) {
        throw e;
    }
    catch (Any a) {
        throw a;
    }
    throw MyClass(14);
}
```

**C++**

```
void foo(){
  try {
    compute();
  }
  catch (tor::EMultiplicityOutOfRange e){
    throw e;
  }
  catch (...) {
    throw;
  }
  throw MyClass(14);
}
```

# Loop Action

**A Loop Action is translated to a while statement, a do-while statement or a for statement.**

These three statements are all variants of the same construct, and which statement that will be used is determined by the syntax variant used in the UML model.

**Example 547:**

**UML**

```
Integer a = 0;
for (Integer i = 0; i < 10; i = i + 1) {
    a = a + 1;
}
while (a > 0)
    a = a - 1;
do {
    a = a + 1;
} while (a < 10);
```

**C++**

```
tor::Integer a = 0;
for (tor::Integer i = 0; i < 10; i = i + 1) {
    a = a + 1;
}
while (a > 0)
  a = a - 1;
do {
a= a+ 1;
} while (a < 10);
```

# Stop Action

**A Stop Action is translated to a call of the “finish” function on the `statemachine` class that corresponds to the `statemachine` implementation in which the stop action is contained.**

**Example 548:**

**UML**

```
stop;
```

**C++**

```
finish();
```

# NextState Action

## Normal NextState Action

**A “normal” NextState Action (a nextstate action that specifies a state) is translated to a call of the “enter” function on the state variable corresponding to the state specified as the next state.**

**Example 549:**

**UML**

```
nextstate idle;
```

**C++**

```
m_s_idle->enter();
```

**If the nextstate action specifies a 'via' entry connection point, the address of the member variable that is the translation of the entry connection point is passed as argument to the 'enter' call.**

**Example 550:**

**UML**

```
nextstate idle via Cin;
```

**C++**

```
m_s_idle -> enter(&m_s_Idle->m_sm->Cin);
```

### History NextState Action

**A history NextState Action is translated to a call of the "`enterHistory`" function of the top region of the `statemachine` implementation that contains nextstate action.**

**Example 551:**

**UML**

```
nextstate -;
```

**C++**

```
theTopRegion::enterHistory();
```

### Deep History NextState Action

**A deep history NextState Action is translated to a call of the "`enterHistory`" function of the top region of the `statemachine` implementation that contains the nextstate action, with the "`deepHistory`" flag set to true.**

**Example 552:**

**UML**

```
nextstate ^-;
```

**C++**

```
theTopRegion::enterHistory(true /* deepHistory */);
```

## Signal Sending Action

**A Signal Sending Action is translated to a call of a "`send`" or "`sendTo`" function, with a dynamically created signal instance as argument.**

The function "`tor::sendTo`" is used when the destination is explicitly specified (either as a `DispatchableClass` or as a Port). Otherwise the "send" function of the owner Dispatchable class is called, which will dispatch the signal according to the current architecture description.

**Example 553: Signal sending with and without explicit destination**

**UML**

```
output X.sig1;
output sig2(4);
```

**C++**

```
tor::sendTo(new sig1(), X);
m_owner->send(new sig2(4));
```

**Example 554: Signal sending via port**

**UML**

```
   port port1 out with PingSignal;
...
   output PingSignal via port1;
...
```

**C++**

```
...
   sendTo( new PingSignal(), &(m_owner->port1) );
...
```

If a signal sending action specifies the sending of more than one signal, there will be one "`send`" or "`sendTo`" call for each sent signal.

# Decision Action

### A Decision Action is translated to an if-statement, with one branch for each decision answer.

The reason for translating it to an if-statement instead of a switch-statement, is that switch-statements in C++ are much more constrained than UML Decision Actions.

**Example 555:**

**UML**

```
switch (e) {
  case <10 : {
     i = 1;
     break;
```

```
    }
    case >=10 :
       break;
    default : {
       i = 3;
    }
  }
  switch (b < 8) {
    case true: {
      i = 1;
    }
    case false: {
      i = 0;
      break;
    }
  }
```

**C++**

```
  if (e < 10)
  {
    i = 1;
    goto GEN_9AnS2IFnpu0Lxq66AVx4TFzV;
  }
  else if (e >= 10)
    goto GEN_9AnS2IFnpu0Lxq66AVx4TFzV;
  else
  {
    i = 3;
  }
  GEN_9AnS2IFnpu0Lxq66AVx4TFzV:
  if ((b < 8) == true)
  {
    i = 1;
  }
  else if ((b < 8) == false)
  {
    i = 0;
    goto GEN_kpCCZVljYHVLaWD9DIlWN0DE;
  }
  GEN_kpCCZVljYHVLaWD9DIlWN0DE:;
```

---

Break actions within a ‘case’ clause are replaced with ‘goto’ statements which transfer control to the action that follows after the decision action. In case no action follows after the decision action an empty action is inserted which contains the label referenced by such ‘goto’ statements.

Note that the semantics for executing ‘case’ clauses is different in UML and C++. In C++ a trailing break statement must be used to prevent “falling through” into the next ‘case’ clause. In UML there is no semantics of “falling

through", which means that it is not necessary to use a trailing break action for this purpose. However, it is still useful to use such a break action to break out from the case clause before reaching its last action.

# Return Action

**A Return Action that is contained in an OperationBody is translated to a return statement.**

**A Return Action that is contained in a transition is translated to a call of the "finish" function of the top region of the state machine implementation that contains the transition.**

**Example 556: Return action within a transition**

**UML**

```
start {
    return;
}
```

**C++**

```
void initialTransition( ) {
    theTopRegion::finish();
}
```

**If the Return Action specifies a 'via' exit connection point, the address of the member variable that is the translation of the exit connection point, is passed as argument in the call of 'finish'.**

**Example 557: Return via connection point**

Return action within a transition specifying an exit connection point.

**UML**

```
start {
    return Cout;
}
```

**C++**

```
void initialTransition( ) {
    theTopRegion::finish(&Cout);
```

```
}
```

## Join Action

**A Join Action is translated to a goto statement.**

Compare with Example 543 on page 1589 for a simple example involving a join action.

An exception to the above rule applies if the join action is contained in a transition and references a label that is not defined within that same transition. Since different transitions are translated to different C++ functions, the default translation rule would translate such a join to a non-local goto, which is not legal C++.

**A Join Action that is contained in a transition, and which references a label defined in another transition, is translated into a call of the transition function that is the translation of the transition in which the label is contained. The call is followed by an immediate return statement.**

Non-local joins will not be supported, unless the referenced label is the label of a label transition.

**Example 558:**

**UML**

```
void foo(Integer i) {
X:
  if (i == 1) {
    i = i + 1;
    join X;
  }
}
```

**C++**

```
void foo(tor::Integer i) {
X:
  if (i == 1) {
    i = i + 1;
    goto X;
  }
```

```
}
```

## Timer Set Action

**A Timer Set Action is translated to a call of the "set" function on the timer variable corresponding to the timer that is set.**

The first argument in the call is the specified time-out expression. If no time-out expression is specified in the Timer Set Action, there must be a default time-out expression specified in the timer definition which then is used instead. The second argument in the call is a dynamically created instance of the `tor::TimerEvent` derivate class that is the translation of the timer definition. Actual timer parameters are translated to actual constructor parameters in the creation of this instance.

**Example 559:**

**UML**

```
timer T1 (Integer i);
timer T2 () = 15;

void foo() {
  set T1 (5) = now + 12;
  set T2;
}
```

**C++**

```
class T1 : public virtual tor::TimerEvent {
public:
  tor::Integer i;
  T1(tor::Integer i) : i(i) {}

  static inline bool isTypeOf(tor::Event* e) {
    return tor::cast<T1*>(e) != 0;
  }
};

class T2 : public virtual tor::TimerEvent {
public:
  static inline bool isTypeOf(tor::Event* e) {
    return tor::cast<T2*>(e) != 0;
  }
};

tor::TimerObject timer_T1;
tor::TimerObject timer_T2;
```

```
void foo() {
  timer_T1.set(tor::os::Time(tor::os::Time::now().to_dou
ble() + 12), new T1(5));

  timer_T2.set(tor::os::Time(tor::os::Time::now().to_dou
ble() + 15, true), new T2());
}
```

## Timer Reset Action

**A Timer Reset Action is translated to a call of the "reset" function on the timer variable corresponding to the timer that is reset.**

**Example 560:**

**UML**

```
timer T1 (Integer i);

void foo() {
  reset T1 (2);
}
```

**C++**

```
class T1 : public virtual tor::TimerEvent {
public:
  tor::Integer i;
  T1(tor::Integer i) : i(i) {}

  static inline bool isTypeOf(tor::Event* e) {
    return tor::cast<T1*>(e) != 0;
  }
};

tor::TimerObject timer_T1;

void foo() {
  timer_T1.reset();
}
```

**Note**

*The only purpose of the actual arguments that is used in the Timer Reset Action in UML is to identify which (possibly overloaded) version of a timer that shall be reset. These actual arguments are thus not visible in the C++ translation.*

# Internals

The Internals construct is not part of standard UML, but is a Telelogic specific extension originally introduced for supporting convenient design of components. It has several similarities with the Bridge design pattern, but is not exactly the same. For C++ generation, Internals are ignored. The purpose of an Internals can be achieved by using the Bridge pattern in the UML design (which then of course is handled by the C++ Application Generator).

# Operation Body

**An operation body in UML is translated to a C++ function body.**

Although an operation body is directly owned by its operation in UML, it will not be so in C++, since it is not usually desired to put an implementation in a header file (with the exception of inline operations).

**Example 561:** ———————————————————————————

**UML**

```
class C {
  void foo(){
    return;
  }
  void bar();
  void <<inline>> inl() {
    return;
  }
}
void C::bar(){
  return;
}
```

**C++** (in file C.h)

```
class C {
  void foo();
  void bar();
  inline void inl() {
    return;
  }
}
```

**C++** (in file C.cpp)

```
void C::foo(){
  return;
```

```
}
void C::bar(){
  return;
}
```

# Signal

**A Signal is translated to a C++ class that inherits from the run-time-class** `tor::Event`**.**

**Example 562:**

**UML**

```
signal sig_Ping;
```

**C++**

```
class sig_Ping : public virtual tor::Event {
  static inline bool isTypeOf(tor::Event* e) {
    return (tor::cast<sig_Ping*>(e) != 0);
  }
};
```

The name of a signal class is the translation of the name of the corresponding signal (compare "Name of definitions" on page 1542). However, since a signal is an event class there may be more than one signal in the same UML scope having the same name (overloading). If that is the case, all overloaded signals with the same name will have their names suffixed with the types of the signal parameters. See Example 563 on page 1604.

**Note**

*The static '*`isTypeOf`*' function determines if an event is dynamically typed by the signal corresponding to the class. The implementation of this function is based on the* `tor::cast` *function (which by default uses* `dynamic_cast`*).*

# Signal parameter

**If a signal has parameters the class that is the translation of the signal gets a constructor with one parameter for each signal parameter. In addition there will be one public attribute for each signal parameter. The name, type, multiplicity etc. of a constructor parameter and class attribute are all identical to the signal parameter for which they were generated.**

If a signal parameter has no name the corresponding constructor parameter and class attribute will get the name "`parX`", where X is the zero-based index of the signal parameter.

**Example 563:**

**UML**

```
signal sig_Ping(Charstring, Boolean b, Integer i = 3);
signal sig_Pong(Integer k);
signal sig_Pong(Boolean b);
```

**C++**

```
class sig_Ping : public virtual tor::Event {
public:
  tor::Charstring par0;
  tor::Boolean b;
  tor::Integer i;
  sig_Ping(tor::Charstring par0, tor::Boolean b,
tor::Integer i = 3) :
    par0 (par0), b (b), i (i) { }
};
class sig_Pong_Integer : public virtual tor::Event {
public:
  tor::Integer k;
  sig_Pong_Integer(tor::Integer k) :
    k (k) { }
};
class sig_Pong_Boolean : public virtual tor::Event {
public:
  tor::Boolean b;
  sig_Pong_Boolean(tor::Boolean k) :
    b (b) { }
};
```

**Note**

*The initializer of the generated constructor will assign the value of each constructor parameter to the corresponding class attribute.*

# Timer

**A Timer is translated to a C++ class that inherits from the run-time-class `tor::TimerEvent`.**

This is done in exactly the same way as when translating a Signal. However, in addition there will be a definition of a timer variable, typed by `tor::TimerObject`, in the translation of the definition that owns the Timer.

The name of the timer variable is the same as the name of the Timer, but a "timer_" prefix is added.

**Example 564:**

**UML**

```
timer T (Integer a) = 15;
```

**C++**

```
class T : public virtual tor::TimerEvent {
public:
  tor::Integer a;
  T(tor::Integer a) :
    a (a) { }

  static inline bool isTypeOf(tor::Event* e) {
    return tor::cast<T*>(e) != 0;
  }

};

tor::TimerObject timer_T;
```

**Note**

*The default timeout value (15 in Example 564 on page 1605) is not directly visible in the C++ translation result. However, it is used in the translation of a Timer Set Action which do not specify a timeout value.*

# State Machine

A UML state machine is modeled as a special kind of operation, but is translated in a special way.

The focus here is on the common case when the state machine is contained in a class (when it is the "classifier behavior"). However, a state machine may also be declared as a separate definition outside a class (for example in a package), or as an inline state machine of a composite state. Compare with "State machine for defining a composite state" on page 1618 for how such a state machine is translated.

In the UML model a clear distinction is made between a state machine (the signature) and a state machine implementation (the implementation). In the C++ translation this distinction is not so clear. In fact, almost all interesting properties of a state machine are parts of its implementation, so this chapter will also cover the translation of state machine implementations.

**A UML state machine is translated to a C++ class that inherits from the run-time class "`tor::StateMachine`", and the run-time class "`tor::TopRegion`".**

The name of a state machine class is "C_SM", where C is the name of the owner class, and SM is the name of the state machine.

**Note**

*It is possible to use a simple state machine as the implementation of an ordinary operation. If such a state machine implementation has no states, it will be translated as an ordinary operation body (the actions of the start transition will be put in the Operation Body.*

**If a UML class contains at least one state machine that is a constructor, the corresponding C++ class will inherit from the run-time class "`tor::DispatchableClass`".**

A generated Dispatchable class contains the following members:

- A member variable "`m_sm`" typed by the state machine class (which is the classifier behavior).

- A redefinition of the virtual function "`tor::DispatchableClass::init`", with an implementation that creates an instance of the state machine class and stores it in the "`m_sm`" member variable. It also calls the inherited implementation.
  If the UML class already has an '`init`' operation another one will of course not be generated. The C++ Application Generator then assumes that the existing '`init`' operation performs the initialization of the Dispatchable class in some custom way.

**Note**

*The '`init`' function also is used for the purpose of Initialization of static structure.*

- A redefinition of the virtual function "`tor::DispatchableClass::finish`", with an implementation that deletes the "`m_sm`" member variable. It also calls the inherited implementation.
  If the UML class already has an '`finish`' operation another one will of course not be generated. The C++ Application Generator then assumes that the existing '`finish`' operation performs the finalization of the Dispatchable class in some custom way.
- A redefinition of the virtual function "`tor::DispatchableClass::start`", with an implementation calls the inherited implementation in order to start the state machine of the Dispatchable class. The implementation also starts each active instance that is a part of the owning Dispatchable class instance. This means that when an instance of an active class is started, all contained instances will also be recursively started.
  If the UML class already has a '`start`' operation another one will of course not be generated. The C++ Application Generator then assumes that the existing '`start`' operation performs the starting of the state machine (and contained instances) in some custom way.
- A redefinition of the virtual function "`tor::DispatchableClass::receive`", with an implementation that just calls the inherited function.
- A redefinition of the virtual function "`tor::DispatchableClass::getClassifierBehavior`", with an implementation that returns the "`m_sm`" member variable.
- A `typedef` of "`tor::DispatchableClass`" which is called "`theDispatchableClass`".

- Friend declarations of the `statemachine` class and all state classes contained in the `statemachine` class.

A generated `statemachine` class contains the following members:

- A member variable called "`m_owner`" typed by the corresponding Dispatchable class.
- A constructor. The implementation of this constructor initializes the "`m_owner`" member.
- A virtual destructor. The implementation of this destructor calls 'finish' to finalize the Dispatchable class before destruction.
- A "`getDispatchableClass`" function with an implementation that returns the owner variable as a `tor::DispatchableClass`.
- Two type definitions for the inherited `tor::StateMachine` and `tor::TopRegion`. These type definitions are called "`theStateMachine`" and "`theTopRegion`".
- A redefinition of the virtual function "`tor::StateMachine::init`", with an implementation that calls the inherited function "`addRegion`" with "this" as argument.
- Friend declarations of all contained state classes.

**Example 565:**

**UML**

```
active class C {
    statemachine initialize {
        ...
    }
}
```

**C++**

```
class C : public virtual tor::DispatchableClass {
protected:
  typedef tor::DispatchableClass theDispatchableClass;
public:
  C() : m_sm(NULL) {}
  virtual ~C()
  {
    finish();
  }
  virtual void init() {
    m_sm = new C_initialize(this);
    theDispatchableClass::init();
  }
```

```
  virtual void finish() {
    if (m_sm) {
      delete m_sm;
      m_sm = NULL;
    }
  }
  virtual void start() {
    theDispatchableClass::start();
  }
  virtual bool receive(tor::Event* e) {
    return theDispatchableClass::receive(e);
  }
  virtual tor::StateMachine* getClassifierBehavior {
    return m_sm;
  }
  friend class C_initialize;
  C_initialize* m_sm;
};

class C_initialize : public virtual tor::StateMachine,
                     public tor::TopRegion {
public:
  typedef tor::StateMachine theStateMachine;
  typedef tor::TopRegion theTopRegion;
  C_initialize(C* owner) : m_owner(owner) {}
  virtual ~C_initialize() {}
  virtual void init() {
    theStateMachine::addRegion(this);
  }
  virtual tor::DispatchableClass* getDispatchableClass()
{
    return m_owner;
  }
protected:
  C* m_owner;
};
```

## State

**A state is translated to a class. The name of the class is the translation of the state name (compare “Name of definitions” on page 1542), and the class inherits from the run-time class “`tor::State`”.**

The state class contains the implementation of all Triggered transitions originating from that state.

The statemachine class will also get a member variable for each state, named “m_s_N”, where N is the translation of the state name. The prefix “`m_s_`” is short for member state.

Each state class is instantiated in the "`init`" function of the state machine class, and each obtained state instance is stored in the corresponding member variable for the state. A corresponding delete statement for freeing the allocated state is put in the destructor.

**Friend declarations for the state class are added to both the `statemachine` class and the owner (Dispatchable) class.**

This is done since the implementation of a state class may need to access private data of the `statemachine` class or even the owner (Dispatchable) class (for example when transferring signal data to variables defined in the owner class or within the `statemachine` implementation).

**Example 566:** ———————————————————————————

**UML**

```
active class C {
    statemachine initialize {
        start {
            {
            }
            nextstate Idle;
        }
        state Idle;
    }
}
```

**C++**

```
class C : public virtual tor::DispatchableClass {
public:
  friend class C_initialize;
  friend class Idle;
  ...
};

class C_initialize : public virtual tor::StateMachine,
public tor::TopRegion {
public:
  ...

  virtual ~C_initialize() {
    ...
    if (m_s_Idle)
      delete m_s_Idle;
  }

  virtual void init() {
    ...
     m_s_Idle = new Idle(this, this);
```

```
    m_s_Idle->init();
    ...
  }

  class Idle : public tor::State {
  public:
    Idle(tor::Region* region, C_Initialize* owner ) :
      tor::State(region),
      m_owner(owner) {}
    tor::Dispatchable::EventAction execute(tor::Event*
e);
  protected:
    C_Initialize* m_owner;
  };

protected:
...
  Idle* m_s_Idle;
public:
  friend class Idle;
};
```

For brevity, the example has left out most things not related to the translation of the state.

---

A state may contain or reference a state machine, in order to become a composite state. The translation of such a state is described in "State machine for defining a composite state" on page 1618.

## Start transition

**A start transition is translated to a function "`initialTransition`" in the `statemachine` class that is the translation of the `statemachine` implementation in which the start transition is contained.**

The implementation of "`initialTransition`" is the translation of the actions of the start transition. This translation follows exactly the same rules as for any other Action.

Compare with "Triggered transition" on page 1612 for an example of translating a start transition.

# Triggered transition

**A triggered transition is translated to a function in the `statemachine` class that is the translation of the `statemachine` implementation in which the triggered transition is contained. Also, it yields an if-statement in the 'execute' function of each state class, corresponding to a state in which the transition may be triggered.**

The name of the “transition function” is

1. if the transition has no guard and no asterisk trigger:

   `trans_<StateNames>_<SignalClassNames>`
2. otherwise:

   `trans_<StateNames>_< SignalClassNames >_<GUID>`

<`StateNames`> are the names of all states in which the transition may be triggered (separated by underscores), and <`SignalClassNames`> are the names of all classes corresponding to any Signal that may trigger the transition.

<GUID> is the GUID of the triggered transition.

The if-statement in the '`execute`' functions uses the '`isTypeOf`' function to test that the dynamic type of the signal receipt event matches the static event type specified in the trigger of the transition. If so the current state is left (by calling the '`leave`' function) and the transition function is called. Finally the event is deleted and the '`execute`' function returns '`Consumed`' to indicate that the event has been consumed.

**Example 567:** ———————————————————————————

**UML**

```
statemachine initialize {
  Charstring 'from';
  start {
    {
      {
        count = 0;
      }
    }
    destination.sig(strName);
    nextstate Idle;
  }
  state Idle;
  for state Idle;
    input sig('from') {
      {
```

```
            {
              Charstring name = strName;
              count = count + 1;
            }
          }
          ^ destination.sig(strName);
          nextstate Idle;
        }
    }
```

**C++**

```
    class C_initialize : public virtual tor::StateMachine,
                         public tor::TopRegion {
        protected:
        C* m_owner;

        class Idle : public tor::State {
            protected:
            C_initialize* m_owner;
            public:
            Idle(tor::Region* region, C_initialize* owner )
            : tor::State(region), m_owner(owner){

            }
            tor::Dispatchable::EventAction execute(
    tor::Event* e ) {
                if (sig::isTypeOf(e))
                    {
                    m_owner->from = tor::cast<sig*>(e)-
    >sender;
                        {
                            leave();
                            m_owner->trans_Idle_sig(e);
                            delete e;
                            return
    tor::Dispatchable::Consumed;
                        }
                    }
                return tor::Dispatchable::NoMatch;
            }
        };

        private:
        tor::Charstring from;
        protected:
        Idle* m_s_Idle;
        public:
        virtual void initialTransition( ) {
            {
                {
                    m_owner->count = 0;
                }
            }
            tor::sendTo(new sig(m_owner->strName), m_owner-
```

```
>destination);
      m_s_Idle->enter();
    }

    void trans_Idle_sig(tor::Event* theEvent) {
        {
            {
                tor::Charstring name = m_owner->strName;
                m_owner->count = m_owner->count + 1;
            }
        }
        m_s_Idle->enter();
    }
};
```

For brevity, the example has left out most things not related to the translation of the triggered transition.

The event that triggered a transition is available as a parameter of the generated “transition functions”. This parameter is called `theEvent` and can be accessed from the actions of the transition by using target code.

---

**Note**

*The actual arguments of a received signal are assigned to the variables referenced in the UML signal sending action.*

### Multiple triggers

**If a triggered transition has more than one trigger, the transition can be triggered by any of the events specified by these triggers.**

Thus, the if-statement condition for the transition in the '`execute`' function will be extended to check the event type against the event types of all triggers, and becomes true if any of these matches.

**Example 568:** ———————————————————————————

**UML**

```
  state Idle;
  for state Idle;
    input sig('from'), sig2(x) {
      {
...
      }
      nextstate Idle;
    }
}
```

**C++**

```
class Idle : public tor::State {
    protected:
    C_initialize* m_owner;
    public:
    Idle(tor::Region* region, C_initialize* owner )
    : tor::State(region), m_owner(owner){

    }
    tor::Dispatchable::EventAction execute( tor::Event*
e ) {
        if (sig::isTypeOf(e) || sig2::isTypeOf(e)) {
            if (sig::isTypeOf(e)) {
                m_owner->from = tor::cast<sig*>(e)-
>sender;
            }
            if (sig2::isTypeOf(e)) {
                m_owner->x = tor::cast<sig2*>(e)->
            }
            leave();
            m_owner->trans_Idle_sig(e);
            delete e;
            return tor::Dispatchable::Consumed;
        }
        return tor::Dispatchable::NoMatch;
    }
};
```

The code that transfers actual signal arguments to state machine variables in this case will be placed inside an if-statement checking that the event types match.

---

A special case of multiple triggers, is the case of an asterisk (‘*’) representing a list of all possible signals to receive:

**A triggered transition with an asterisk trigger ('*') is triggered by a received event of any kind. The if-statement condition for such a transition will thus just be a check that the event (e) is not NULL (the part of the condition that is derived from specified triggers).**

The if-statement for an asterisk trigger is placed as the last if-statement in the execute function.

**Example 569:** ———————————————————————————

**UML**

```
...
```

```
  for state Idle;
    input * {
...
    }
}
```

**C++**

```
class Idle : public tor::State {
...
    tor::Dispatchable::EventAction execute(tor::Event* e
) {
        if (e)
        {
...
        }
        return tor::Dispatchable::NoMatch;
    }
};
```

---

# Guard

**A guard on a triggered transition is translated to an additional expression in the if-statement for the transition in an 'execute' function. The expression must evaluate to true for the transition to be triggered.**

**Example 570:** ———————————————————————————————

**UML**

```
...
  for state Idle;
    input sig('from') [myGuard == true]{
...
    }
}
```

**C++**

```
tor::Dispatchable::EventAction execute( tor::Event* e )
{
    if (sig::isTypeOf(e) && (myGuard == true))
        {
...
        }
```

---

If a transition only has a guard and no triggers, the transition should be triggered as soon as the guard expression evaluates to true, without the reception of an event. Therefore the if-statement for the transition will in this case only check that the event variable ('e') is NULL, and that the guard expression is true.

**Example 571:**

**UML**

```
...
  for state Idle;
    [myGuard == (x > 14)]{
...
    }
}
```

**C++**

```
tor::Dispatchable::EventAction execute( tor::Event* e )
{
    if (!e && (x > 14))
        {
...
        }
```

# Label transition

**A label transition is translated to a function in the `statemachine` class that is the translation of the `statemachine` implementation in which the label transition is contained.**

The name of the “transition function” is “trans_<`LabelName`>”, where <`LabelName`> is the name of the label of the label transition.

**Example 572:**

**UML**

```
statemachine initialize {
  L:
  {
    count = 1;
  }
  ...
}
```

**C++**

```
class C_initialize : public virtual tor::StateMachine,
                     public tor::TopRegion {
  ...
  void trans_L() {
    count = 1;
  }
}
```

---

## State machine for defining a composite state

A state machine that is not defined as the "classifier behavior" of a class, can be used to define a sub-state machine of a composite state. Such a state machine can either be defined as a stand-alone state machine that can be referenced independently from different composite states, or it can be defined as an inline sub-state machine of one particular composite state.

**A "composite state" state machine is translated in the same way as a "classifier behavior" State Machine, with the following modifications:**

1. There is no Dispatchable class to transform in this case.
2. The name of the state machine class is C_S_SM, if the state machine is the inline state machine of a composite state, or SM if the state machine is a stand-alone state machine. S is the name of the composite state, and C is the name of the state machine class that is the translation of the state machine implementation that contains that state. SM is the name of the state machine.
3. The "`m_owner`" attribute is typed by the class that is the translation of the owning statemachine (in the case of an inline statemachine), or by "`tor::DispatchableClass`" (in the case of a stand-alone statemachine).
4. A member attribute "`m_ownerState`" is added. Its type is the class that is the translation of the owning state (in the case of an inline statemachine), or by "`tor::State`" (in the case of a stand-alone statemachine).
5. The implementation of the "`getDispatchableClass`" function is modified for the case of an inline state machine. It will return

   `"m_owner->getDispatchableClass()",`
   in order to recursively get to the owner Dispatchable class.

The state class of a composite state will contain an "`init`" function which instantiates the sub-state machine class, and the obtained state machine instance is stored in a member variable "`m_sm`" in the class of the composite state. A corresponding delete statement for freeing the allocated state machine is put in the destructor of the state class.

The "`init`" function of a state class is called in the "`init`" function of the state machine class for the state machine implementation that owns the state.

**Example 573: Composite state with inline state machine**

**UML**

```
statemachine initialize {
...
  state Idle : statemachine initialize {
  ...
  };
}
```

**C++**

```
class C_initialize_Idle_initialize : public virtual
tor::StateMachine,
                                     public
tor::TopRegion {

  class Idle : public tor::State {
      protected:
      C_initialize* m_owner;
      public:
      Idle(tor::Region* region, C_initialize* owner);
     tor::Dispatchable::EventAction execute(tor::Event*
e);

      virtual void init() {
         m_sm = new
C_initialize_Idle_initialize(m_owner, this);
         m_sm -> init();
      }

      C_initialize_Idle_initialize* m_sm;
      virtual tor::StateMachine* getStateMachine() {
        return m_sm;
      }
  };

...
};
```

**Example 574: Composite state with non-inline state machine**

**UML**

```
statemachine initialize {
...
  state Idle : SM {
  ...
  };
}
statemachine SM {
...
}
```

**C++**

```
class Idle : public tor::State {
    protected:
    C_initialize* m_owner;
    public:
    Idle(tor::Region* region, C_initialize* owner);
    tor::Dispatchable::EventAction execute(tor::Event*
e);

    virtual void init() {
       m_sm = new SM(m_owner->getDispatchableClass(),
this);
       m_sm -> init();
    }

    SM* m_sm;
    virtual tor::StateMachine* getStateMachine() {
      return s_sm;
    }
};

class SM : public virtual tor::StateMachine,
           public tor::TopRegion {
...
}
```

This shows that the only difference between a composite state that has an in-line state machine and a composite state that has a non-inline state machine is a small difference in the implementation of the "`init`" function.

# Connection point

**A connection point is translated to a member variable in the `statemachine` class that is the translation of the state machine in which the connection point is contained.**

The name of the member variable is the translation of the connection point name. The type of a connection point is `tor::EntryPoint` for an entry connection point, and `tor::ExitPoint` for an exit connection point.

The connection point member variables are initialized in the constructor of the state machine class.

**Example 575: Connection point**

**UML**

```
active class C : DispatchableClass {
    statemachine initialize {
...
        state Idle : statemachine initialize
        in Cin
        out Cout {
            start Cin {
...
            }
            start {
...
            }
            state WaitForSig;
            for state WaitForSig;
                input sig() {
                    return Cout;
                }
        };
        for state Idle;
            [Cout] {
                {
                    count = 14;
                }
                stop;
            }
    }
}
```

**C++**

```
class C_initialize : public virtual tor::StateMachine,
                     public tor::TopRegion {
    public:
    void trans_Idle_Cout() {
```

```
      m_owner->count = 14;
      finish();
    }
    virtual void exitPointTransitions( tor::ExitPoint*
ep) {
      if(ep == &m_s_Idle -> m_sm -> Cout) {
       if (m_current)
          m_current->leave();
        trans_Idle_Cout();
        return;
    }
    protected:
    Idle* m_s_Idle;
};
class C_initialize_Idle_initialize : public virtual
tor::StateMachine,
                                     public
tor::TopRegion {
    friend class WaitForSig;

    public:
    void trans_Cin() {
    ...
    }
   virtual void entryPointTransitions( tor::EntryPoint*
ep) {
      if (ep == &Cin) {
        trans_Cin();
        return;
      }
    }
    Idle* m_ownerState;
    public:
    tor::EntryPoint Cin;
    tor::ExitPoint Cout;
};
```

**Note**

*Details not related to the translation of the connection points have been omitted from the example.*

---

# Architecture

The architecture (or internal structure) of a class is composed of the attributes of the class, the ports of the class and the connectors that link them together. This section describes how these concepts are translated to C++ code.

## Attributes

**For each attribute that is typed by an active class and has a multiplicity > 1 a protected utility method in the containing class is generated that automatically will add an instance to the attribute, add the instance to the same dispatcher as the owner, initialize the instance, connect it to appropriate other instances based on the surrounding connectors, and finally start it.**

**Example 576:**

**UML**

```
active class Sys {
...
part Ping[*] p1;
...
part Pong[*] p2;
connector c1 from p1.pingP to p2.pingP;
...
}
```

**C++**

```
...
void Sys::p1_Insert( Ping * p ) {
  .p1.append( p );
  .addToCurrentDispatcher(p);
  .p->init();
  .for (int i = 1; i <= p2.length(); ++i) {
    c1.connect(&(p->pingP), &(p2[i]->pongP));
  .}
  .p->start();
}
...
```

It is not supported to change the collection type used for the attribute that contains the active instance. It must be a type that implements the iteration operators used by the Insert operation.

**The calls of the predefined template operation `tor::insert<Any p>(Any obj)` is translated to a call to the appropriate utility method as defined in previous rule.**

**Example 577:**

**UML**

```
   part Ping[*] p1;
...
   tor::insert<p1>( new Ping() );
...
```

**C++**

```
...
   this->p1_Insert( new Ping() );
...
```

**For each active class a protected utility method in the corresponding class is generated that automatically will disconnect it from all connectors it currently is connected to.**

**Example 578:**

**UML**

```
active class Ping {
...
   port p1 out with PongSignal;
...
}
```

**C++**

```
...
void Ping::disconnect() {
      p1.disconnect();
}
...
```

## Connectors

**Each connector is translated to an attribute of the containing class typed by `tor::Connector`. The name of the created attribute is the translation of the Connector name.**

**Example 579:**

**UML**

```
...
  connector c1 from part1.port1 with sig1 to part2.port2
with sig2;
...
    }
}
```

**C++**

```
...
  tor::Connector c1;
...
```

The connectors are used as the basis for connecting ports and support a '`connect`' method that can connect two ports on different objects.

**Note**

*Implicit connectors (connectors that are created based on matching unconnected ports) are not supported.*

Connectors can have constraints with respect to the transported signals/supported interfaces etc. associated with them. These constraints are checked statically by the tool but not enforced at run-time. In most cases the only consequence is that the execution is faster, but it also implies that 'connector splitting' based on different subsets of supported signals will not work. If more than one connector is connected to a port the signal will always follow the first connector that is connected.

In general it is good to avoid run-time ambiguities of where to deliver a sent signal. If connectors are not relied upon for signal transportation (i.e. signals are sent to explicitly specified receivers) no ambiguities can arise. If connectors are used, you may use a 'via <port>' clause to specify which outgoing port to use on the sender. This can reduce or eliminate the risk of ambiguities in case multiple outgoing ports exist. If a signal is output without specifying

an explicit receiver, nor an outgoing port, there must exist exactly one matching port and connector path to the receiver at run-time. Otherwise the signal will be lost (or delivered to an unexpected receiver).

# Ports

**Each port is translated to an attribute of the containing class typed by `tor::Port` for each communication direction supported by the port. The name of the created attribute is `<portName>_in` and <`portName`>, where <`portName`> is the translation of the Port name.**

**Example 580:**

**UML**

```
...
  port p1 in with sig1 out with sig2;
...
```

**C++**

```
...
  tor::Port p1_in;
  tor::Port p1;
...
```

**For each port that is a behavior port there is code generated in the `init` method for the corresponding class that initializes the "`targetBehavior`" attribute of the port to true.**

**Example 581:**

**UML**

```
active class Pong ... {
...
  bport p1 in with sig1 out with sig2;
...
}
```

**C++**

```
...
void Pong::init() {
    ...
    p1.targetBehavior = this;
```

```
    ...
}
...
```

## Initialization of static structure

When dynamically creating an instance of a class that contains static parts (as defined by their Multiplicity), instances of the parts will automatically be created and initialized. This is accomplished by generating code in the `init` method for the containing class.

**For each attribute typed by an active class with a static multiplicity there is code generated in the `init` method that initializes the static instances.**

**Example 582:**

**UML**

```
active class Top {
...
part Ping[1] ping;
part Pong[1] pong;
connector c1 from ping.port1 with sig1 to pong.port2
with sig2
...
}
```

**C++**

```
...
class Top ... {
...
Ping ping;
Pong pong;
...
}
...
void Top::init() {
    m_sm = new Top_initialize(this);
    theDispatchableClass::init();
    addToCurrentDispatcher(&ping);
    addToCurrentDispatcher(&pong);
    ping.init();
    pong.init();
    c1.connect(&(ping.pingP),&(pong.pongP));
}
...
```

If the UML class already has an 'init' operation another one will of course not be generated. The C++ Application Generator then assumes that the existing 'init' operation performs the initialization of the Dispatchable class in some custom way.

**Note**

*The 'init' function also is used for the purpose of State Machine initialization.*

**For each attribute typed by an active class with a defined initial cardinality there is code generated in the init method that initializes the initial instances.**

**Example 583:**

**UML**

```
active class Top {
...
part Ping[*]/2 ping;
part Pong[*]/3 pong;
connector c1 from ping.port1 with sig1 to pong.port2
with sig2
...
}
```

**C++**

```
...
class Top ... {
...
tor::String<Ping> ping;
tor::String<Pong> pong;
...
}
...
void Top::init() {
    m_sm = new Top_initialize(this);
    theDispatchableClass::init();
    tor::addToDispatcher<Ping>(this, ping, 2);
    tor::addToDispatcher<Pong>(this, pong, 2);
    tor::init(ping, 2);
    tor::init(pong, 2);
    for (int i2 = 1; i2 <= 2; i2++) {
      for (int i3 = 1; i3 <= 3; i3++) {
        c1.connect( &(ping[i2]->pingP), &(pong[i3]-
>pongP) );
      }
    }
```

```
...
```

`tor::addToDispatcher, tor::init` and `tor::start` are utility functions of the TOR run-time library. They are used instead of for-loops to gain readability.

**Note**

*Static initialization (with initial cardinality > 0 or a constant multiplicity > 1) is only supported when the created class has a constructor without parameters.*

## Disconnecting an instance

When deleting an active instance with ports it will automatically be removed from the architecture, for example it will be disconnected from the connectors that it is attached to. This is handled by the run-time framework.

# Package TTDCppPredefined

The `TTDCppPredefined` package contains UML representations of C++ language constructs and predefined types. It is used by both the C/C++ Import and the C++ Application Generator, but may of course be used by the user also when none of these tools are used.

## Predefined types

The `TTDCppPredefined` package contains the following UML types representing predefined C++ types:

| UML Type | Predefined UML Type | C/C++ Fundamental Type |
|---|---|---|
| `int` | `Integer` | `signed int`,<br>`int` |
| `unsigned int` | `Integer` | `unsigned int`,<br>`unsigned` |
| `long int` | `Integer` | `signed long int`,<br>`signed long`,<br>`long int`,<br>`long` |
| `unsigned long int` | `Integer` | `unsigned long int`,<br>`unsigned long` |
| `short int` | `Integer` | `signed short int`,<br>`signed short`,<br>`short int`,<br>`short` |
| `unsigned short int` | `Integer` | `unsigned short int`,<br>`unsigned short` |
| `long long int` | `Integer` | `signed long long int`,<br>`signed long long`,<br>`long long int`,<br>`long long` |
| `unsigned long long int` | `Integer` | `unsigned long long int`,<br>`unsigned long long` |
| `char` | `Character` | `char` |
| `signed char` | `Character` | `signed char` |
| `unsigned char` | `Character` | `unsigned char` |

| UML Type | Predefined UML Type | C/C++ Fundamental Type |
|---|---|---|
| `wchar_t` | `Character` | `wchar_t` |
| `float` | `Real` | `float` |
| `double` | `Real` | `double`<br>`long double` |
| `bool` | `Boolean` | `bool` |

## Stereotypes

The TTDCppPredefined package contains the following UML stereotypes representing C++ constructs that cannot directly be represented in plain UML:

### 'globalNamespace' extends Package

Specifies that no explicit namespace should be generated for the package. Instead the package represents the implicit global namespace of C++.

### 'struct' extends Class

Specifies that the class should be translated into a struct instead of a class.

### 'inheritanceVisibility' extends Generalization

Contains attributes for specifying that a generalization should be translated into inheritance that is non-public (private or protected)

### 'virtualInheritance' extends Generalization

Contains attributes for specifying that a generalization should be translated into virtual inheritance

### 'inline' extends Operation

Specifies that a function is declared as inline.

### 'bitfield' extends Attribute

Contains an integral attribute for specifying the number of bits to use for a bitfield.

### 'CppReference' extends StructuralFeature

Specifies that an attribute or parameter should be a C++ reference (&). This stereotype is mainly used for specifying that the return parameter of an operation should be mapped to a C++ function which returns a reference.

### 'auto' extends StructuralFeature

Specifies that an attribute or parameter should have the `auto` storage specifier in C++.

### 'register' extends StructuralFeature

Specifies that an attribute or parameter should have the `register` storage specifier in C++.

### 'mutable' extends Attribute

Specifies that an attribute should be declared as mutable.

### 'volatile' extends StructuralFeature, Operation

Represents the C++ `volatile` specifier.

### 'explicit' extends Operation

Specifies that a constructor should be marked as `explicit` in C++.

### 'export' extends Signature

Specifies that a template definition should be marked as exported in C++ using the `export` keyword.

### 'friend' extends Dependency

This stereotype is used in the UML representation of a C++ friend declaration. See Friend dependency for more information.

### '__declspec' extends Definition

Represents the `__declspec` keyword which are supported by some C/C++ compilers as an extension to the C/C++ language.

### 'manifest implementation' inherits manifest

This stereotype can be used instead of the ordinary 'manifest' stereotype to specify that a file artifact representing a C++ implementation file manifests all implementation aspects of the supplier definition. So instead of having to specify for each operation body of a class that they should be put in the same C++ implementation file, it is possible to specify that by just one dependency, stereotyped by this stereotype, from the file artifact to the class.

# Translation Options

Options to the C++ Application Generator are represented as Tagged values for attributes of the <<C++ Application Generator>> stereotype contained in the C++ Application Generator profile package. The options are summarized in the sections below. For each option the corresponding stereotype attribute is specified.

## Name mangling options

These options are related to name mangling which is performed by the code generator when a UML name cannot be used as is in the generated C++ code.

### Name prefix

```
nameManglingOptions = NameManglingOptions (. namePrefix .)
(string)
```

Specifies which prefix to give to definitions to make their name legal according to C++ naming rules. Default is "`Name_`".

## Enable non-ASCII compilation

```
enableUnicode (boolean)
```

If set to true, all character and Charstring values will be enclosed in the _T macro, which makes it possible to compile the generated code in both ASCII and non-ASCII (e.g. Unicode) configurations.

## Default model-to-file mapping options

These options are related to the default Model-to-File Mapping used by the code generator.

### Use default model-to-file mapping

```
generateDefaultFileMapping (boolean)
```

If set to true, the C++ Application Generator will generate a UML specification for the Model-to-File Mapping it will use by default.

### Use precompiled header

```
fileMappingOptions = FileMappingOptions (.
precompiledHeader .) (string)
```

The specification of a precompiled header file to be used for all implementation files that are generated in the default model-to-file mapping.

## Code formatting options

These options are related to how the generated code will be formatted.

### Indentation

```
codeFormattingOptions = CodeFormattingOptions (. indentSize
.) (natural)
```

The number of spaces to use for each indentation level.

## Code organization options

These options are related to how the generated code will be organized.

### Sorting of bodies

```
sortingOptions = CodeOrganizationOptions (.
sortOperationBodies .) (SortOperationBodiesKind)
```

This option controls how to organize operation bodies in the generated code. They can either be sorted alphabetically, or according to the order of the corresponding operations in the model. In the latter case, synthesized bodies will be placed after other non-synthesized operation bodies.

Note that this option does not affect inline operation bodies.

### Grouping of members

```
sortingOptions = CodeOrganizationOptions (. sortMembers .)
(SortMembersKind)
```

This option controls how to group member definitions. They can be grouped either by their order in the model, or by their visibility. When grouping according to order in the model, synthesized members will be placed after other non-synthesized members. When grouping according to visibility public members will be generated before protected members which in turn will be generated before private members.

### Group transition if-statements according to trigger definition scope

```
sortingOptions = CodeOrganizationOptions (.
sortTransitionIfStatements .) (Boolean)
```

If this option is enabled, transition if-statements will be grouped according to the scope where the transition trigger is defined. Scopes more local to the transition will be placed before more remote scopes.

## Enable COM agents

`enableCOMAgents (boolean)`

When this option is turned on agents implemented as COM objects will be enabled during code generation. Since there is a performance overhead associated with using COM, you should not turn this option on unless you are using COM agents for customizing the code generator.

## Support roundtrip

`supportRoundtrip (boolean)`

If set to true, the generated files will contain a few annotations (comments and macros) which makes it possible to "roundtrip" changes in the files back to the UML model. This option could be turned off if you do not intend to make changes in the generated files. Benefits with doing so include improved code generator performance (existing files do not need to be analyzed and can simply be overwritten) and less annotations in the generated code.

## Time unit

`timeUnit (TimeUnit)`

Specifies the unit of time to use in the generated application. Choose between the time units: seconds, milliseconds, microseconds and nanoseconds.To use another time unit or change time unit dynamically at run-time, see "setTimeUnit" on page 1676.

## Link with TOR

`torLibrary (TORLibraryLinkKind)`

Specifies how the generated code should be linked with the TOR library. The default is to link with TOR as a static library, but there is also the option of linking with TOR as a dynamic (shared) library. For more information on this topic, see Building TOR.

If the generated code does not depend on the TOR library, this setting is ignored.

# Instrumentation Options

These options are related to instrumentation of the generated code.

### Enable instrumentation

```
instrumentationOptions = InstrumentationOptions (. enable
.) (boolean)
```

When this option is turned on the C++ Application Generator will generate instrumented code. This typically includes two extra files `torInstrumentation.h` and `torInstrumentation.cpp` and also some minor additions to other generated files. The purpose of the instrumentation is to make the generated application able to communicate what it is doing by means of sending events (known as “meta events” to distinguish them from the application-level events). These meta events can be written to a log file or sent to the host Tau IDE in order to produce a sequence diagram trace showing what is happening in the application. Instrumentation must also be enabled in order to perform UML level debugging.

### Synchronous instrumentation events

```
instrumentationOptions = InstrumentationOptions (.
syncMetaEvents .) (boolean)
```

When this option is turned on the processing of meta events (i.e. events telling what the application is doing) is synchronous. This can be useful for example if the generated application is debugged in an external debugger while producing a sequence diagram in a host Tau IDE. The diagram will then be updated incrementally while stepping though the debugged application.

# Debug options

These options are related to UML level debugging of a model that has been translated to C++ by the C++ Application Generator.

### Executable for debug session

```
debugSettings = DebugOptions (. executable .) (string)
```

When starting a debug session from Tau using the Launch command on a build artifact, the application that was generated from the build artifact will by default be launched, and attached to for debugging. This option allows

you to specify another executable to launch instead. This can for example be useful if a dynamic (shared) library was generated from the build artifact, and another application is used as host for the library.

### Command line arguments

```
debugSettings = DebugOptions (. commandLineArgs .) (string)
```

Allows you to specify command-line arguments for the executable that is launched when starting a debug session from Tau using the Launch command.

### Host name

```
debugSettings = DebugOptions (. host .) (string)
```

Specifies the name of the TCP/IP host for a debug session that is started by attaching to an already running executable.

### TCP/IP port

```
debugSettings = debugOptions (. port.) (natural)
```

Specifies the port number of the TCP/IP host for a debug session that is started by attaching to an already running executable.

## Include protection options

These options control the format of the include protection that are generated for each header file. See Include protection for more information.

### Include Protection First String

```
includeProtSettings = IncludeProtectionSettings (.
includeProtectionBegin .) (string)
```

Controls the format of the beginning of the include protection (generated in the beginning of the header files).

### Include Protection Last String

```
includeProtSettings = IncludeProtectionSettings (.
includeProtectionEnd .) (string)
```

Controls the format of the end of the include protection (generated in the end of the header files).

## Automatic model update

`autoUpdate (boolean)`

If set to true, model updates will be performed automatically when Tau detects that a generated C++ source file has been modified (for example saved in a text editor). Use of this option requires that the Support roundtrip option also is enabled.

## Automatic code generation

`autoGenerate (boolean)`

If set to true, C++ code generation will be performed automatically when a modified UML model is saved.

## Automatically add operation bodies for operations

`autoGenerateOperationBodies (boolean)`

This option controls if the C++ Application Generator should automatically add empty operation bodies for operations which have no body defined in the model.

# Translation Customization

The output of the C++ Application Generator can be customized using agents. This gives the user a very precise control of the generated code.

The generated code can be customized in two different ways:

1. Adding arbitrary text at certain locations in the generated files. This makes it possible to generate additional code, comments or preprocessor directives in conjunction with the ordinary code that gets generated.
2. Replacing the generation of certain constructs of the C++ language by an arbitrary text. This makes it possible to represent non-C++ constructs, such as use of macro libraries, as UML entities, and to define how they shall be translated to C++.

This chapter describes these two ways of customing the C++ Application Generator. Both methods are based on the definition of customization agents (see Chapter 75, Agents for more information about agents).

## Adding Text During Code Generation

This kind of customization is based on tool events and agents triggered during code generation by these tool events. The customization that is possible is to add arbitrary text

- at the beginning or end of a generated file (see tool event Print C++ Source File)
- just before or just after the generation of a C++ definition (see tool event Print C++ Definition)

The example **umlCppAgentCustomization** contains an example of how to write a customization agent using the C++ API in order to respond to the Print C++ Definition tool event.

## Replacing Text During Code Generation

This kind of customization is modeled using user-defined stereotypes. These user-defined stereotypes are specializations of a set of built-in stereotypes that indicate which C++ construct (grammar rule) that should be customized,

e.g. Operation Heading, Operation Definition, Typedef and so on. The built-in stereotypes that are to be specialized by the user all have definitions on the following form:

```
stereotype customCppGen<GRAMMAR RULE> extends
TTDMetamodel::<METACLASS> [0 .. 1] {
    abstract void Unparse(out Charstring result);
}
```

<GRAMMAR RULE> is the name of the C++ grammar rule that can be customized through that stereotype, and <METACLASS> is the UML meta class that corresponds to that grammar rule.

Since the "Unparse" operation is defined to be abstract in the base stereotype, the user-defined stereotype must implement this operation. This is done by adding an "Unparse" operation with the same signature to the user-defined stereotype, and to apply the <<agent>> stereotype on that operation in order to define an implementation to execute. That agent implementation is responsible for assigning a string value to the output parameter "result". That string will then be printed by the C++ Application Generator instead of the text that normally is generated for that particular grammar rule.

**Note**

*We recommend that translation customization is not used in connection with round-trip engineering. If you anyway intend to roundtrip files that have been generated with user-defined customization agents active, the code that is added by these agents should be enclosed within the <GENERATED> tagged comments so that they will be ignored during roundtrip.*

**Example 584**

To illustrate these customization possibilities, let's see an example from a special application of C++: System-C. System-C is a C++ based hardware modeling language intended both for performance analysis and hardware synthesis. Essentially it is a library of C++ classes and utilities that provide a mechanism for describing hardware in C++. A central concept in System C is a "Module". This is a C++ class but in source code it is defined using a macro SC_MODULE. A small System-C module (defining a nand gate) might look like this:

```
SC_MODULE(nand){
sc_in<bool> A, B;
sc_out<bool> O;
}
```

Even though this a standard C++ class it is defined using a macro. Since this macro usage replaces the usual C++ class definition the standard C++ code generation can not generate this code. However using the customization possibilities this can be arranged.

What we want to achieve is that an end user simply defines a class in the UML model and stereotypes it with <<Module>> to indicate that it is a System-C module. When he runs the C++ code generator this should generate the SC_MODULE macro as in the example above instead of the usual class header.

To accomplish this we create a UML profile for System-C with a stereotype definition <<Module>>. To make this stereotype affect the C++ code generation we make it inherit from the built-in stereotype customCppGenClassHeading. Furthermore we add one operation called “Unparse” that is defined to be an agent that will override the standard C++ code generation. The implementation of the agent can be done in various ways but in this case we choose a C++ implementation provided in a separate DLL. In textual syntax the definition of the <<Module>> stereotype looks like this:

```
stereotype Module : customCppGenClassHeading {
<<agent(.implKind = CPP,
implementation="some/path/myLib#Module.dll".)>>
void Unparse(out Charstring result);
}
```

When the System-C profile is loaded the C++ code generator will call the Unparse agent instead of emitting code for a class heading whenever it finds a class that is tagged with the stereotype <<Module>>.

The dynamic library Module.dll that is referenced from the agent definition contains the implementation in C++ of a function that assigns a text string value for the “result” output parameter. This text string is then used instead of the normal class heading. In this case the agent can be defined as follows in the C++ source code:

```
AGENT_IMPL( Module )
{
    tstring strReturn = _T("SC_MODULE( ");
    tstring strName;
    pContext->GetValue( _T("Name"), strName );
    strReturn += strName;
    strReturn += _T( " )" );

    u2::AgentParameter* apReturn = agentParameters.front();
    apReturn->Set(strReturn);
}
```

Note the implicit parameter “pContext” that identifies the element that triggers the agent execution, in this case the <<Module>> class. The text string that will be part of the generated code is returned using the “agentParameters” out parameter.

### Customization Points

This section defines the contexts where the output of the C++ code generator can be customized by replacing the text that normally would be emitted for that particular context. The definition of what parts of the C++ code that is replaced is done in terms of the supported C++ grammar. See Chapter 51, C++ Textual Syntax.

### Namespace Heading

Name of stereotype: customCppGenNameSpaceHeading

Extends: Package

Grammar rule:

<namespace definition> ::=

‘namespace’ [ <identifier> ] <braced declarations>

Replaced part: ‘namespace’ [ <identifier> ]

### Namespace

Name of stereotype: customCppGenNameSpace

Extends: Package

Grammar rule:

<namespace definition> ::=

‘namespace’ [ <identifier> ] <braced declarations>

Replaced part: Entire rule

### Class Heading

Name of stereotype: customCppGenClassHeading

Extends: Class

Grammar rule:

<class definition> ::=

<class key> [ <identifier> ] [ <base clause> ] <class body>

Replaced part: <class key> [ <identifier> ] [ <base clause> ]

**Class**

Name of stereotype: customCppGenClass

Extends: Class

Grammar rule:

<class definition> ::=

<class key> [ <identifier> ] [ <base clause> ] <class body>

Replaced part: Entire rule

**Interface Heading**

Name of stereotype: customCppGenInterfaceHeading

Extends: Interface

Grammar rule:

<interface definition> ::=

'UML_INTERFACE' <identifier> [ <base clause> ] <interface body>

Replaced part: 'UML_INTERFACE' <identifier> [ <base clause> ]

**Interface**

Name of stereotype: customCppGenInterface

Extends: Interface

Grammar rule:

<interface definition> ::=

'UML_INTERFACE' <identifier> [ <base clause> ] <interface body>

Replaced part: Entire rule

**TypeDef**

Name of stereotype: customCppGenTypedef

Extends: Syntype

Grammar rule:

<typedef declaration> ::=

‘typedef’ <decl specifier seq> <declarator> ‘;’

Replaced part: Entire rule

**Attribute**

Name of stereotype: customCppGenAttribute

Extends: Attribute

Grammar rule:

<member declaration> ::=

[ <decl specifier seq> ] [ <member declarator> ] ‘;’

| <function definition> [ ‘;’ ]

| <using declaration>

| <template declaration>

| <friend declaration>

| <typedef declaration>

Replaced part: [ <decl specifier seq> ] [ <member declarator> ] ‘;’

**Enumeration Heading**

Name of stereotype: customCppGenEnumerationHeading

Extends: Datatype

Grammar rule:

<enum definition> ::=

‘enum’ [ <identifier> ] <enumerator body>

Replaced part: ‘enum’ [ <identifier> ]

**Enumeration**

Name of stereotype: customCppGenEnumeration

Extends: Datatype

Grammar rule:

<enum definition> ::=

‘enum’ [ <identifier> ] <enumerator body>

Replaced part: Entire rule

**Union Heading**

Name of stereotype: customCppGenUnionHeading

Extends: Choice

Grammar rule:

<class definition> ::=

<class key> [ <identifier> ] [ <base clause> ] <class body>

Replaced part: <class key> [ <identifier> ] [ <base clause> ]

**Union**

Name of stereotype: customCppGenUnion

Extends: Choice

Grammar rule:

<class definition> ::=

<class key> [ <identifier> ] [ <base clause> ] <class body>

Replaced part: Entire rule

**Operation Definition Heading**

Name of stereotype: customCppGenOperationHeading

Extends: Operation

Grammar rule:

<function definition> ::=

[ <simple decl specifier seq> ] [ <type specifier> ] <declarator> <function body>

| '#if' <expression> [ <simple decl specifier seq> ] [ <type specifier> ]
<declarator> <function body> '#endif'

Replaced part: [ <simple decl specifier seq> ] [ <type specifier> ] <declarator>

**Operation Definition**

Name of stereotype: customCppGenOperation

Extends: Operation

Grammar rule:

<function definition> ::=

[ <simple decl specifier seq> ] [ <type specifier> ] <declarator> <function body>

| '#if' <expression> [ <simple decl specifier seq> ] [ <type specifier> ]
<declarator> <function body> '#endif'

Replaced part: [ <simple decl specifier seq> ] [ <type specifier> ] <declarator> <function body>

**Operation Declaration Heading**

Name of stereotype: customCppGenOperationDeclarationHeading

Extends: Operation

Grammar rule:

<function definition> ::=

[ <simple decl specifier seq> ] [ <type specifier> ] <declarator> <function body>

| '#if' <expression> [ <simple decl specifier seq> ] [ <type specifier> ]
<declarator> <function body> '#endif'

Replaced part: [ <simple decl specifier seq> ] [ <type specifier> ] <declarator>

**Operation Declaration**

Name of stereotype: customCppGenOperationDeclaration

Extends: Operation

Grammar rule:

<function definition> ::=

[ <simple decl specifier seq> ] [ <type specifier> ] <declarator> <function body>

| '#if' <expression> [ <simple decl specifier seq> ] [ <type specifier> ]

<declarator> <function body> '#endif'

Replaced part: [ <simple decl specifier seq> ] [ <type specifier> ] <declarator> <function body>

**Operation Body**

Name of stereotype: customCppGenOperationBody

Extends: Operation

Grammar rule:

<function body> ::=

[ <ctor initializer> ] <compound statement>

Replaced part: Entire rule

**Generalization**

Name of stereotype: customCppGenGeneralization

Extends: Generalization

Grammar rule:

<base specifier> ::=

[ <access to base> ] <identifier>

Replaced part: Entire rule

# Miscellaneous

Here is described some aspects of the UML to C++ translation which are not directly related to any UML construct, but rather have to do with the nature of the C++ language.

## Order of declarations and forward declarations

The declarations that are generated into a C++ source file are ordered in the following way:

1. Definitions (including inline operation bodies)
2. Operation Bodies (non-inline)

If a definition depends on another definition within the same file, a forward declaration of the supplier definition is generated if required by the ordering between the reference and the definition. The forward declaration is placed as close as possible to the first referring entity. However, the scope rules of C++ must be followed, meaning that a forward declaration must be placed in the same scope as the corresponding definition. Furthermore, in the case of a member definition the access visibility (public, private etc.) must be the same for the forward declaration and the corresponding definition.

**Example 585:**

**UML**

```
class C {
  public D x;
  class D {
  }
  E y;
}
class E {
}
```

**C++**

```
class E; // placed in global scope
class C {
public:
  class D; // public and placed in C
  D* x;
  class D {
  };
  E* y;
};
```

```
class E {
};
```

**Note**

*When a definition A in scope S has an ordering dependency to another definition B in the same scope S, then A will be generated before B. This means that definitions within one scope will be sorted if required by their ordering dependencies. An ordering dependency is a reference which requires the full definition of the target (a forward declaration is not sufficient). Ordering dependencies are transitive, that is to say that if A has a dependency to B which has a dependency to C, then A also has a dependency to C.*

Forward declarations are also generated in header files instead of `#include`:s whenever possible. By doing so the risk for circular include dependencies between generated files is minimized.

## Main function

Every C++ program must have a function called ‘main’ in the global scope, in order to define the entry point of the executable application. When generating a C++ application from a UML model there are three alternatives for how to get a ‘main’ function.

1. Specify an operation ‘main’ in the UML model and make sure it is placed in a «`'global namespace'`>>` package. Add a «`'manifest implementation'`>>` dependency in order to manifest the implementation of the operation in an implementation file. The implementation of the ‘main’ operation can be fully specified in UML, or contain fragments of target code if necessary.
2. Specify an operation ‘main’ in the UML model as above, but do not implement it in UML. Instead write the implementation by hand in a separate file which then is compiled and linked with the generated application.
3. Do not specify ‘main’ at all in the UML model, but define it completely manually.

**Note**

*When generating a library from the UML model it is of course not necessary to have a ‘main’ operation.*

### Generating a default main operation in the UML model

Tau provides a command for adding an operation ‘main’ to the UML model which will be translated to a ‘main’ function by C++ Application Generator. To invoke this command, do the following:

1. Select a package in the Model View. The package should be a <<global namespace>> package.
2. In the context menu select the command **Utilities - Generate Main Function**.

The implementation of the default ‘main’ function is generated according to the following rule:

**One instance is created for each active class that is manifested by the Build Artifact. Each instance is added to one single Dispatcher, and is then initialized and started. Finally the ‘run’ function is called on the Dispatcher.**

This means that the default ‘main’ function makes a fully synchronous (single-threaded) application. You can of course modify this default implementation in any way to become appropriate for your program.

**Example 586: Generation of a default ‘main’ function**

**UML**

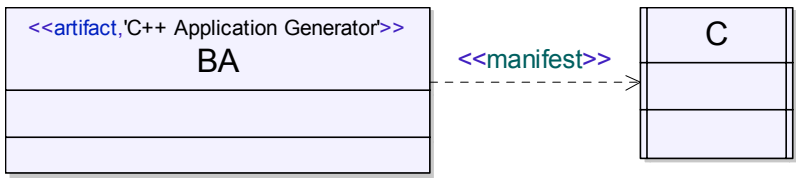



*Figure 255: Active class manifested by build artifact*

**C++**

```
int main(int argc, char** argv) {
  tor::Dispatcher* dispatcher = new tor::Dispatcher(new
tor::EventQueue);
  C* v1 = new C;
  dispatcher -> add(v1);
  v1 -> init();
  v1 -> start();
```

```
  dispatcher -> run();
  return 0;
}
```

Note that the model must contain a C++ build artifact which manifests the package in which the main operation is generated. If no C++ build artifact can be found an empty main operation will be generated (a warning will then be printed in the Messages tab).

# 53

# *Environment of C++ Applications*

This section describes techniques for how to interface an application (or part of an application) generated with the C++ Application Generator with the environment. Which technique to choose for a particular model depends on a number of things, such as the characteristics of the environment, the thread deployment used in the model, and personal taste of the designer. Therefore, the information presented here should not be seen as a complete listing of available alternatives, but more as guidelines for some common typical cases.

# Introduction

A traditional definition of what constitutes the environment of an application usually includes all parts of the application that in some sense are considered external to the application's main area of responsibility. The word "external" here is deliberately vague since it often is difficult to draw a sharp borderline between the application and its environment. There are many criteria for drawing such a borderline, for example:

- Hardware components vs. software components
- User interface components vs. other components
- Components developed in one technology vs. components developed in another technology
- Legacy components vs. newly developed components

In the context of this section the term environment will be used to mean such parts of an application that are not generated by the C++ Application Generator. Examples include handwritten software components, third-party libraries, or hardware components.

# Modeling the Environment

It is often useful to include a representation of the environment in the UML model itself. Such a representation may consist of one or many UML definitions (typically classes, attributes, actors, subjects etc.). Often this can be done already during system analysis or early design.

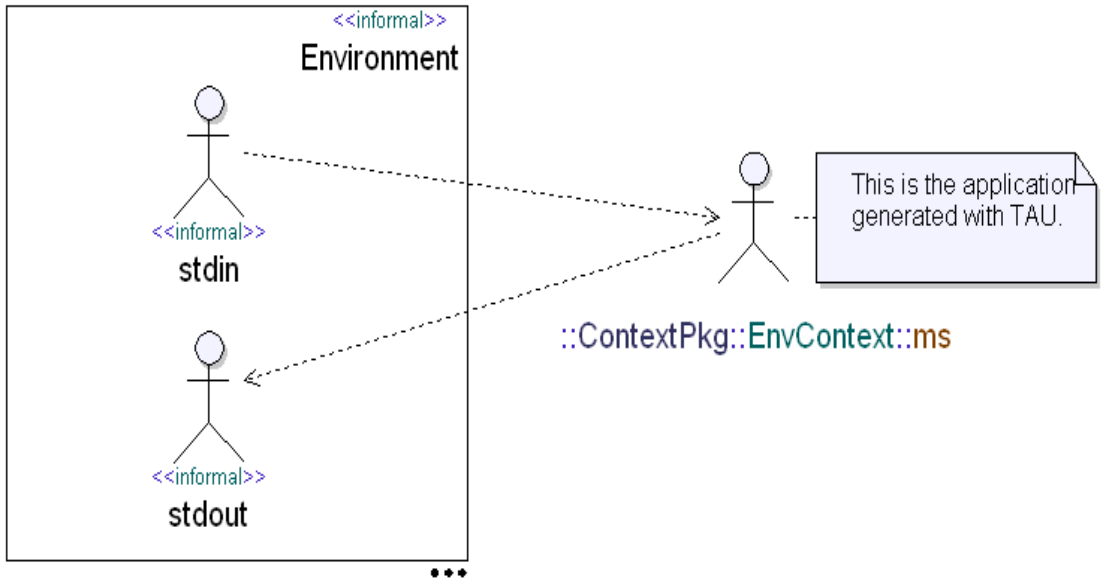



*Figure 256: Modeling the environment of an application reading from* `stdin` *and writing to* `stdout`.

# Interfacing with the Environment

When going into more detailed design the interfaces to components in the environment must be formalized. There are at least three ways of doing this:

- If an environment component is a C or C++ module, the C/C++ Import may be used to automatically create a detailed UML representation of its interfaces. Thereby it can be used directly from the UML model that is generated with the C++ Application Generator.
- If an environment component is a hardware component it is often a good idea to add an interface adaptor for the component in the model. Such an interface adaptor is often modeled as a class-typed attribute, and is responsible for interacting with the hardware component. It also serves as an abstraction for the hardware component for the rest of the UML model.

- If an environment component consists of external code (for example a graphical user interface) it may realize the `tor::EventReceiver` interface which allows signals to be sent to it from the UML model. Of course, other custom interfaces may also be used for the purpose of plain function calls (in both directions). These interfaces should be included in the UML model, and will be generated to a C++ header by the C++ Application Generator. This header is then included in the external code. If the external module needs to send signals back to the UML model it need to be linked with the TOR library.

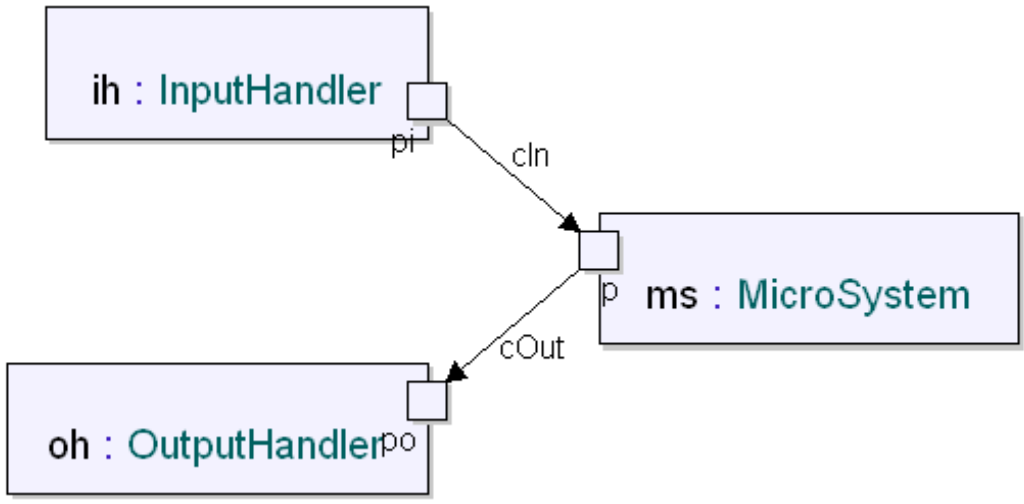



*Figure 257: Representing environment components with interface adaptors.*

In general the following mechanisms are useful when interfacing with components in the environment:

1. The ability to access external C or C++ code from the model. This is made easy using the C/C++ Import tool, but also other ways exists, such as using inline target code ([[...]]). If the external code is developed in parallel with the model, it is also useful to generate the headers of external classes from Tau, and then manually create the implementation files.
2. The ability to call operations through interfaces that are realized by an environment component. The interfaces could either be externally defined (should then be imported to UML) or defined within the UML model (should then be generated to C++).

3. The ability to send signals to an external C++ class (using the `tor::EventReceiver` interface). The external class may either process the signal immediately (synchronous execution) or at a later time (asynchronous execution).
4. The ability to send signals from external code to the model. This is done using the `tor::sendTo` utilities that are available in the `torUtilities.h` header of the C++ Run-time Framework.
5. The ability to call operations and access data that are defined in UML from external code. This is done by simply including the relevant header files that have been generated from the model.

## Multi-threaded Applications

If the model is deployed as a multi-threaded program it is important to understand that operation calls or signals that are made from entities in the model may be performed by different threads. If the external component that is interfaced with is not thread-safe by itself, an interface adaptor could be made for the component in the UML model, see Figure 257 on page 1656. This adaptor should typically be an instance of an active class that either is dispatched by its own thread or in one of the other threads in the model. In either case it will act as a "protector" against threading problems in the environment component, by "sequencing" all communication from the model to the external component.

An interface adaptor responsible for reading data from some resource often becomes blocking (to avoid a busy-wait for data to read). Such an interface adapter should typically be given its own thread of execution to avoid blocking other parts of the model while waiting for data to read from the external component.

# 54

# *C++ Run-time Framework*

This chapter describes the C++ run-time framework, Tau Object Run-time (TOR). The purpose and run-time semantics of all classes in the framework is described.

**See also**

Chapter 52, C++ Application Generator Reference for details on how TOR is used by the code generated with the C++ Application Generator.

# Introduction

This chapter is a reference guide for Tau Object Run-time (TOR), an object-oriented UML run-time framework implemented in C++. TOR implements the run-time semantics of UML, both for structural and behavioral aspects.

TOR is implemented as a set of classes each providing a well specified service. Some classes are used in the framework itself, while others are used by the code generated with the C++ Application Generator to transform a model to executable C++ code.

Some of the classes are used when modeling; these "TOR Classes" are available as a TOR UML Model.

The framework is delivered as a set of header and source files. The files are listed in the List of Files section.

## TOR namespace

All declarations of TOR are made in a namespace called `tor`. This facilitates the use of short and descriptive class names, like State, while avoiding conflicts with user defined classes. In addition it prevents having too many definitions in the global scope.

## TOR UML Model

Parts of TOR are available as a model library, called 'tor', that is loaded automatically when the C++ Application Generator is activated. All types, classes, operations and functions that can be used in the user model is included in this model.

The classes and operations of this model are described in the TOR Classes section.

**Note**

*The entities in the TOR 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.*

## Building TOR

TOR is usually built automatically when building an application generated with the C++ Application Generator for the first time. If the generated application does not depend on TOR definitions in a way that require it to be linked with TOR (for example, it may only use template definitions defined in header files) TOR will of course not be built.

The C++ Application Generator stereotype applied on a build artifact provides an option for controlling whether TOR should be built as a static library or as a dynamic (shared) library. The default is to build it as a static library. You should change this to build TOR as a dynamic (shared) library instead, if your application consists of multiple binaries that all depend on TOR. A typical situation is when your application consists of a number of dynamic (shared) libraries, or an executable together with a number of dynamic (shared) libraries. It is important to use dynamic linking in these cases since TOR is designed to only exist in one copy in an entire application. Having multiple copies of TOR in an application (which will be the case if more than one binary link statically with it) can lead to unexpected run-time problems related mainly to signals that are lost when sent between the different binary modules of the application.

When building TOR as a dynamic (shared) library the macro `TOR_DLL` will be defined when compiling the generated code and the TOR library itself. Any user-written code that is included in the application and which depends on TOR must also be compiled with that macro defined.

For more information about how to customize the way TOR is built see Building.

## Initializing and Finalizing TOR

Before the TOR library can be used it must be initialized. Initialization takes place automatically when the library is loaded, either as part of starting an executable that links with TOR statically, or when loading TOR built as a dynamic (shared) library.

When TOR shall not be used by the application anymore it should be finalized to release resources (for example threads and memory) that were taken during its initialization. Finalization of the library takes place automatically when the library is unloaded, either as part of exiting an executable that links with TOR statically, or when unloading TOR built as a dynamic (shared) library.

Sometimes the automatic initialization and finalization of TOR is not enough. For example, if an application only uses TOR in parts of its execution it could save resources (memory and threads) by explicitly finalizing TOR when it no longer needs it. Later it may initialize TOR explicitly when it again needs the library.

Explicit initialization and finalization of TOR is performed using the functions initializeLibrary and finalizeLibrary.

**Note**

*On some operating systems (for example Windows) a normal exit of a program (for example by returning from the main function) leads to an implicit termination of all active threads* ***before*** *dynamically linked libraries are unloaded. Thus, if TOR is dynamically linked with an application running under such an operating system, it must be explicitly finalized by a call to* finalizeLibrary *before exiting the application. Otherwise the automatic finalization that takes place when TOR is unloaded will cause the program to hang since the threads that are used by TOR were prematurely terminated by the OS.*

It is good practise to be explicit with the initialization and finalization of TOR although not always needed. For example, the call to initializeLibrary could be placed first in the main function, or in the constructor of some global object. In the same way the call to finalizeLibrary could be placed last in the main function, or in the destructor of some global object.

**Important!**

*If TOR is explicitly finalized by a call to* finalizeLibrary *it is important that TOR is not used after that. One situation that is especially tricky is when the application has deployed the UML model onto multiple threads (using ThreadedDispatcher). All such threads must be finalized before TOR is finalized; otherwise there is a risk that these threads will access TOR after it has been finalized, which usually will lead to a crash. Hence, remember to delete all instances of ThreadedDispatcher before calling* finalizeLibrary.

# TOR Classes

This section contains an alphabetical list of the classes defined in TOR. The purpose and run-time semantics of each class is described. All classes are declared in the TOR namespace. For OS related classes, see Operating System Abstraction Layer. For metadata related classes, see Meta-Data Representation.

- CompletedEvent
- Connector
- Dispatchable
- DispatchableClass
- Dispatcher
- DispatcherData
- EntryPoint
- Event
- EventExecutor
- EventQueue
- EventReceiver
- ExitPoint
- InstanceManager
- InternalEvent
- Port
- Region
- RunInitialTransition
- State
- StateMachine
- ThreadedDispatcher
- ThreadSafeEventQueue
- TimerEvent
- TimerObject
- TimerQueue
- TopRegion

## CompletedEvent

An internal Event generated by the framework when a State is finished. It is passed to the state and processed instantly. It is used to trigger any transitions without a triggering event, also known as trigger free transitions.

There exists a null event which is used for transitions with guards only. In comparison, the CompletedEvent triggers transitions without guards, that is transitions with no events and no guards.

## Connector

A connector represents a connection between two Ports. It is used for sending Events between the classes that own the ports. These classes are typically subclasses of DispatchableClass.

A connector can connect two ports. When connected, an event sent to one of the ports is passed on to the other port. To connect two ports use the connect operation:

```
void connect(Port* from, Port* to);
```

Connecting ports can also be managed using operations on the port itself.

## Dispatchable

An abstract class representing an entity with the ability to both receive and execute events. The ability to receive events is represented by a realization of the EventReceiver interface, and the ability to execute events is represented by a realization of the `EventExecutor` interface.

A dispatchable is associated with a Dispatcher and an EventQueue. When the dispatcher processes an Event from a queue, it asks the dispatchable to which the event is addressed to process the event. As a result, the dispatchable returns a Dispatchable::EventAction indicating how the event is handled.

### Dispatchable::EventAction

An enumeration used to indicate how an event is handled by the receiver. The literals and their meaning are listed in the table below.

| Literal | Description |
|---|---|
| `NoMatch` | The event is not handled by the receiver. |
| `Defer` | The event is saved by the receiver. |
| `Consumed` | The event is consumed by the receiver. |

## DispatchableClass

A dispatchable class represents a UML active class. It inherits Dispatchable to get the capability of receiving an executing events that are dispatched to it.

Examples of dispatchable classes include classes with a State machine and/or internal structures, compare with "Architecture Modeling" on page 299 in Chapter 8, *UML Language Guide*.

Events sent to an instance of a dispatchable class are passed on to the State-Machine of the instance.

If a class is made active, an inheritance to `tor::DispatchableClass` will be added automatically. The active property thus holds the information of the dispatchable class.

### Instantiation of dispatchable classes

When an instance of a dispatchable class is created, it does not automatically start the execution of its behavior. It must first be initialized, after which it can be started.

During initialization, any internal structure and the state machine is prepared for execution. The initialization is normally done automatically by the code generated with the C++ Application Generator. To manually initialize a dispatchable class, call the `init` operation:

```
virtual void init();
```

Starting an instance starts its state machine by posting a request (RunInitial-Transition) for executing the initial transition. Once started, it is ready to receive events. Starting is often performed automatically by the code generated with the C++ Application Generator. For example, the generated main function will start the instances of manifested classes, and starting a container by default starts its contained elements too. However, to ensure maximum flex-

ibility all this can be customized. The start operation can be overloaded to provide a custom way of starting the instance, and starting can be done manually at any time by calling the `start` operation:

```
virtual void start();
```

Starting an already started instance has no effect.

**Note**

*Beware that "start" is a reserved word in UML textual syntax. It is therefore necessary to write it with single quotation marks ('start') when referring to the function above.*

## Dispatcher

A dispatcher is associated with an EventQueue and is responsible for processing Events placed in the queue. Events can be retrieved from the queue and processed one by one, or continuously as long as the queue is not empty.

A dispatcher is also associated with a TimerQueue where TimerObjects corresponding to currently active timers are located.

The code associating a dispatcher with a Dispatchable and then starting the dispatcher is normally added by the code generated with the C++ Application Generator, but it can also be done manually as described below.

To add a dispatchable to the dispatcher, call the `add` operation:

```
void add(Dispatchable*);
```

To remove a dispatchable from the dispatcher, call the `remove` operation:

```
bool remove(Dispatchable*);
```

To start processing events from the event queue of the dispatcher, call the `run` operation. It retrieves events one by one from the event queue until the queue is empty, then it returns.

```
void run();
```

To process events off the queue one by one, call the step operation. It retrieves and processes the next event in the queue.

```
void step();
```

## DispatcherData

A container class containing the data needed by a ThreadedDispatcher. The data is encapsulated in a DispatcherData in order to be safely accessible both by the thread that is launched by the ThreadedDispatcher and the calling thread.

## EntryPoint

An entry point is a named pseudo state used to enter a Region of a composite State or a StateMachine. It provides a means to enter the composite state or sub-state machine without revealing anything about its internals, for example the actual target state.

When an entry point is reached, i.e. a transition with an entry point as its target has been executed, the region owning the entry point is entered, and then the outgoing transition of the entry point is executed.

An entry point can have any number of incoming transitions, but only one outgoing transition.

## Event

The Event class is used to represent types and occurrences of all kind of events in a system, for example; signals, asynchronous operation calls and timer time-outs. Subclasses of Event specifies event types and instances of these classes represent event occurrences.

Every event has a receiver that is represented by an object id. The InstanceManager is responsible for mapping such an object id to a pointer to a Dispatchable that is the real receiver of an event instance. The receiver is set internally by the framework.

## EventExecutor

This is an abstract class capturing the ability to execute events. The class contains one pure virtual function that must be implemented by concrete implementations of EventExecutor:

```
virtual EventAction execute(Event* event) = 0;
```

Implementations of this function should return an appropriate literal of the Dispatchable::EventAction enumeration to indicate how the event was handled by the EventExecutor.

Note that before an event can be executed, it must be received. Thus it is normally so that a class that implements the EventExecutor interface also implements the EventReceiver interface.

**Note**

*It is important to make a distinction between receiving and executing an event. An event is **received** when it is delivered by the framework to the receiver. An event is **executed** when behavior that is associated with the reception of the event is executed (typically a transition). Event reception must precede event execution, and there is typically some time interval between reception and execution.*

## EventQueue

An event queue is a queue of Events. Events placed in the queue are processed by the Dispatcher associated with the queue.

The framework automatically handles everything else related to event queues, i.e. insertion and removal of events.

## EventReceiver

This is an abstract class capturing the ability to receive events. The class contains one pure virtual function that must be implemented by concrete implementations of EventReceiver:

```
virtual bool receive(Event* e) = 0;
```

Implementations of this function should return true if the event was received, and false otherwise.

EventReceiver is also available in the TOR UML profile as an interface. This makes it possible to declare your own event receivers. Events can be sent to such event receivers by means of the sendTo utility function. One common use for this is when there are passive classes that need to receive signals. By letting the classes inherit the EventReceiver interface, and implementing the `receive` function, this becomes possible.

For information about the difference between receiving and executing an event see the note in EventExecutor.

## ExitPoint

An exit point is a named pseudo state used to leave a Region of a composite State or a StateMachine. It provides a means to leave the composite state or sub-state machine without knowing anything about the context in which it is instantiated, for example the target state of the outgoing transition.

When an exit point is reached, i.e. a transition with an exit point as its target has been executed, the exited region (owning the exit point) is left, and then the outgoing transition of the exit point is executed.

An exit point can have any number of incoming transitions, but only one outgoing transition.

## InstanceManager

This class is responsible for keeping a map between object ids and EventExecutors and EventReceivers. The map is kept up-to-date when event receivers and event executors are created and deleted. The main use for the instance manager is to have a symbolic representation (an object id) of an instance, rather than a direct pointer to the instance. This indirection is crucial in a multi-threaded system where instances are created and deleted independently from multiple threads. It is also necessary in a system that is distributed over multiple address spaces.

## InternalEvent

This class is a common base class for all events that are sent internally by the framework, i.e. all events that do not correspond directly to user-defined events in the model.

## Port

A port represents a connection point through which a DispatchableClass can send and receive Events.

A port is associated with a dispatchable class owning the port. This association is normally set up automatically by the generated C++ code during initialization of the dispatchable class.

### Connecting ports

To connect a port with another port using a Connector, call the `attach` operation on one of the Ports:

```
void attach(Port * port, Connector * connector);
```

### Sending and receiving events

When an event is sent to a port, it can be handled in two different ways:

1. If the port is connected to any other ports through a Connector, the event is sent to the first connected port found by the framework.
2. Otherwise, if the associated dispatchable class has a classifier behavior, i.e. StateMachine, the events is passed on to the state machine.

A boolean value is returned to indicate if the event is handled by the receiver or not. To send an event through a port, call the `send` operation:

```
bool send(Event * event, bool deleteEvent = true);
```

## Region

Represents an orthogonal region of a State or a StateMachine. A region owns a set of states. A region keeps track of its current and previous states. The current state is the currently active state of the region. The previous state is the state that was active the last time the region was left.

A region can be entered and left as the result of a transition.

### Entering a region

A region can be entered in a number of different ways. The first time a region is entered, its initial transition is executed. Subsequently, when the region is entered, history information can be used to re-enter the previous state. A region can also be entered through an EntryPoint.

The current and previous states are updated whenever a state in the region is entered as a result of a transition.

### Leaving a region

A region is left when a transition with a target state in a different region is triggered, or when a transition with an ExitPoint as its target is triggered.

The current and previous states are updated whenever a state in the region is left.

### Finishing a region

A region is finished when a final state of the region is reached. Final states are not explicitly defined in the framework.

When a region is finished, trigger free transitions of the enclosing state(s) are evaluated.

## RunInitialTransition

An internal event that is sent to a DispatchableClass when it is started. When this event is executed, the initial transition of the class will be executed.

## State

Represents a state in a StateMachine. A state is owned by a Region. A state can own a number of regions or a state machine, in which case it is referred to as a composite state.

### Transitions and event handling

A state has a set of incoming and outgoing transitions. Transitions are not represented by separate classes in the framework, instead they are represented by operations in the enclosing state machine.

When a state receives an event, it checks if there are any outgoing transitions triggered by the event. If there is a match and the guard expression is true, the transition is executed. Transitions are not ordered, the first matching one found by the framework will be executed.

If no match is found, the event is passed on to any parent of the state, i.e. another state or a state machine.

### Entering a state

A state is entered when a transition with the state as a target has been executed. When a state is entered any entry actions of the state are executed. Then if the state is a composite state, any owned regions or state machine is also entered.

The current and previous state of the owning region is updated when a state is entered.

### Leaving a state

A state is left when any of its outgoing transitions is triggered, or when any outgoing transition of a parent to the state is triggered. When a state is left any exit actions of the state are executed.

The current and previous state of the owning region is updated when a state is left.

## StateMachine

Represents a state machine and its implementation.

**Note**

*A StateMachine has exactly one TopRegion.*

A state machine is associated with a DispatchableClass, as the classifier behavior of the class. An Event sent to the dispatchable class is passed on to the state machine and processed there.

A state machine can be owned by a State, in which case it is called a sub-state machine. Events sent to the state are passed on to the sub-state machine for processing.

Before a state machine is ready to process events it has to be initialized and started. This is done internally by the framework when the corresponding actions are preformed on the associated dispatchable class.

## ThreadedDispatcher

This class, historically also known as DispatcherThread, is a threaded version of a Dispatcher. ThreadedDispatcher realizes a thread which will dispatch events to all Dispatchables that are added to it. This allows for flexible thread deployment of a model. Some examples:

- One thread per active class instance
  Each active class is then associated with exactly one ThreadedDispatcher. This can for example be modeled by letting each active class contain one ThreadedDispatcher as a part. In the constructor of the class, the newly created instance is added to the ThreadedDispatcher, and it is launched. The lifetime of the launched thread is tied to the lifetime of the ThreadedDispatcher, so when an instance of the active class is deleted, the thread will be ended.
- One thread per active class
  This can for example be modeled by adding a static ThreadedDispatcher attribute to the class. In the constructor of the class, the newly created instance is added to the ThreadedDispatcher.
- One thread for an arbitrary set of instances
  This can for example be modeled by adding a package-scoped ThreadedDispatcher attribute, and then add the instances it shall dispatch to it.

A ThreadedDispatcher owns a DispatcherData which comprises all data that needs to be accessed by both the launched thread and the caller thread. For example, the dispatcher data contains a Dispatcher and a ThreadSafeEventQueue. The encapsulation of the data in a DispatcherData instance ensures that the data can be accessed in a thread-safe manner.

The dispatcher and the event queue are automatically instantiated and associated. The dispatcher will process events off the event queue in a thread-safe manner.

To start a ThreadedDispatcher call

```
void launch();
```

To stop a ThreadedDispatcher (will end the launched thread) simply delete it. The thread may also be ended explicitly by a call to

```
bool endThread();
```

This function will end the ThreadedDispatcher as soon as possible, but without interrupting the behavior triggered by the currently executed event. The function is synchronous, i.e. it will wait until the behavior triggered by the current event has run to completion. There is also an overloaded version of this function that allows for specifying a timeout value, to avoid waiting too long for the thread to end. If the thread has not been ended after the specified time has elapsed, the function will return false.

## ThreadSafeEventQueue

A thread-safe subclass of EventQueue used by ThreadedDispatcher. All operations performed on the queue is protected by a Mutex or a Semaphore.

## TimerEvent

A timer event is a special kind of Event, representing a timer timeout event. It is associated with a TimerObject, on which timer actions such as `set` and `reset` can be performed.

## TimerObject

A timer object represents the declaration of a timer in a Dispatchable. It provides functions for setting and resetting a timer and for querying a timer whether it is currently active or not. The implementation of these functions uses an associated TimerEvent and TimerQueue to realize the timer semantics.

A timer object also holds the timeout time value when the timer is active.

## TimerQueue

A timer queue is a priority queue of TimerObjects corresponding to timers that are currently active. The timer objects are ordered with regards to their timeout times.

Every Dispatcher has a timer queue in which timer objects corresponding to active timers of managed Dispatchables are administered.

## TopRegion

A top region is a Region owned by a StateMachine. The owning state machine can be retrieved from a top region.

When a top region is finished the owning state machine is also finished if all its top regions are finished.

TopRegion is a subclass of Region.

# Utilities

There are a number of utility functions that are frequently used internally by the framework and by the generated C++ code. They can also be used by the user in the TOR UML library. All utilities are declared in the TOR namespace.

## sendTo

A function used to send an Event to a receiver. There are three overloaded versions of this function. They allow you to specify the receiver as an EventReceiver, as an object id, or as a a Port.

```
bool sendTo(Event* e, EventReceiver* c);
bool sendTo(Event *event, InstanceManager::ObjectId
receiverId);
bool sendTo(Event* e, Port* pp);
```

The event is sent to the receiver for processing. A boolean value is returned to indicate if the event was received by the receiver or if it was lost.

In either case, the responsibility of the event is passed on to the receiver and the event must not be accessed or deleted after passing it to the `sendTo` function.

The version with an EventReceiver or object id as the receiver should be used when the receiver is known. The Port version is used when sending events in internal structures when the receiver is not known to the sender. Only the port through which the event is sent needs to be known. Naturally, there is a small overhead associated with sending an event through a port rather than sending it to the receiver directly, so only use the Port version of `sendTo` when you must rely on the connector structure for conveying the event.

## cast

A template function used for run-time type-checking of Events. Performs a dynamic cast on the event to see if it matches the template type.

```
template <class T> T cast(Event* e) {
    return dynamic_cast<T>(e);
}
```

There are two versions of this function, one working on const declared event pointers and one on non-const pointers.

## setTimeUnit

A function used for setting the unit of time to be used by TOR.

```
void setTimeUnit(double seconds);
```

The time unit is specified in seconds. For example, to specify a time unit of 1 millisecond, call `tor::setTimeUnit(0.001)`.

The C++ Application Generator has an option for setting the Time unit to seconds, milliseconds, microseconds or nanoseconds. If you want to use a different time unit (for example minutes), or if you want to dynamically change the time unit at run-time, it is possible to call `setTimeUnit`.

**Important!**

*At present the time unit is shared throughout the entire application. Do not change time unit from different threads and when timers already have been activated, to avoid unexpected results.*

## initializeModel

This is only available as an operation in the TOR UML library. The reason is that the implementation of this operation is generated by the C++ Application Generator. Although it is not always mandatory from a functional point of view to call `initializeModel` in a UML model, it is a good idea to always do this anyway (for example in the beginning of the `main` operation, or in the constructor of a global object).

## initializeLibrary

Performs an explicit initialization of the TOR library. Initialization is automatically performed (see Initializing and Finalizing TOR for more information), so this function is only needed if TOR has been explicitly finalized using finalizeLibrary, and should be initialized again.

Calling this function has no effect if TOR already is initialized.

### finalizeLibrary

Performs an explicit finalization of the TOR library. Finalization is automatically performed when terminating an application (see Initializing and Finalizing TOR for more information). Sometimes it is however necessary, or desirable, to finalize TOR explicitly at an earlier stage, and then this function should be used.

Calling this function has no effect if TOR already is finalized.

# Predefined Types

A number of data types is defined in TOR to allow mapping from the Predefined types in UML to C++ data types (Predefined). All types are declared in the TOR namespace.

## Simple types

The table below describes the mapping of simple UML data types to C++ primitive types.

| UML Type | C++ Type |
|---|---|
| Boolean | `bool` |
| Character | `char (wchar_t in Unicode)` |
| Integer | `int` |
| Natural | `unsigned int` |
| Real | `double` |
| OperationReference | `void*` |

## Operators

### equal

```
Boolean equal(Integer i1, Integer i2)
Boolean equal(Natural n1, Natural n2)
Boolean equal(Boolean b1, Boolean b2)
```

**implies**

```
Boolean implies(Boolean b1, Boolean b2)
```

**mod**

```
Integer mod(Integer i1, Integer i2)
```

**power**

```
Integer power(Integer base, Integer exponent)
```

**is**

```
template <class T, class ARG> Boolean is(ARG arg)
```

**as**

```
template <class T, class ARG> T as(ARG arg)
```

### Any class

The Any type is mapped to an empty class called Any.

```
class Any {};
```

### Charstring class

The Charstring type is mapped to a class with the same name based on the `std::string` or `std::wstring` (in Unicode) templates.

## Containers

The general containers defined in the Predefined package in U2 Predefined are implemented in TOR. All containers are declared in the TOR namespace.

**Note**

*Only the String container is supported.*

### String

The String container in TOR is a C++ implementation of the U2 container `Predefined::String`.

String is defined as a template class based on `std::vector`.

**See also**

"Collections and impact of multiplicity" on page 1547 in Chapter 52, *C++ Application Generator Reference*

# Operating System Abstraction Layer

TOR uses a separate abstraction layer to interface with the underlying operating system. This section describes the classes defined in this layer.

All classes in the OS layer are defined in a namespace called `os`, which is a namespace contained in the `TOR namespace` `tor`. The fully qualified name is therefore `tor::os`.

Some of the classes in the OS layer are available in the TOR UML Model. For example, it is often necessary to make use of the thread synchronization primitives of TOR, such as mutex or semaphore.

### Mutex

A mutex is used to lock resources accessed by more than one Thread. A mutex can be locked and unlocked. A status is returned when performing any of these operations to indicate if it is successful or not. Only the thread that locked a mutex can unlock it.

There is a convenient utility class called `auto_lock`, which can be used to synchronize a block of code in a compound statement. When constructed it locks the mutex, and when destructed it unlocks it. It is particularly useful when synchronizing a function with multiple exit paths (including functions that throw exceptions), since it otherwise is easy to forget to unlock the mutex in all ways the function may be returned from.

### RWLock

A read/write lock used to access data in a thread-safe way. The lock can be configured to lock/unlock for read and lock/unlock for write independently. A typical way of using a read/write lock is to lock writing but allow many readers.

### Semaphore

A semaphore used to access and wait for resources accessed by more than one Thread. A semaphore can be retrieved and released. There are two ways of getting a semaphore, either wait until the semaphore is released so you can get it, or wait for a specified amount of time before moving on.

### Gate

A gate is a synchronization primitive that allows one Thread to control the execution of other threads. When a thread attempts to enter the gate, it will only succeed if the gate it unlocked. If it is locked it has to wait until another thread unlocks it. Only one thread may pass the gate simultaneously.

### Thread

An OS thread. Represents a thread of execution at the OS level. Threads are normally created and launched automatically by the framework. See ThreadedDispatcher for usages.

### Time

OS representation of time (both absolute time and relative time durations). The class contains various functions for getting the current time, adding and subtracting time values etc.

### Process

Representation of an OS process. The class contains functions for launching and killing processes and for retrieving their process id (PID).

# Meta-Data Representation

TOR defines a number of classes that are used to describe meta-data about an application generated by the C++ Application Generator. The meta-data includes both information about the structure of the application (a “tree” of available definitions, their names and so forth), and information about what the application is doing (which operations are called, which signals are sent and so forth). The latter is known as “meta-events”. This section describes these meta-data classes.

All classes in the meta-data layer are defined in a namespace called `meta`, which is a namespace contained in the `TOR namespace` `tor`. The fully qualified name is therefore `tor::meta`.

Meta-data is only available in instrumented applications. The C++ Application Generator has an option to Enable instrumentation. The effect of turning this option on is that some instrumentation is added to the generated source files. In addition the files `torInstrumentation.h` and `torInstrumentation.cpp` will be generated. These files contains definitions that use the classes in the meta-data layer of TOR. The TOR library is compiled with the macro TOR_USE_INSTRUMENTATION set when building an instrumented application. This macro enables all definitions of the meta namespace.

### Application

An application is an object that represents the entire application. There is always exactly one instance of this class in an application. Application inherits from Definition, and its name and GUID are taken from the Build Artifact that was used when generating the application.

### Call

This Event is sent when an operation is called. It is sent from the context of the caller.

### Called

This Event is sent when an operation is called. It is sent from the context of the called operation.

### Coder

Interface that must be realized by all coder classes. A coder encodes application data to a CoderBuffer, and decodes a CoderBuffer to application data.

### CoderBuffer

Interface to an abstract buffer that can be used by a Coder class in order to store the result of encoding data from the application.

### Create

This Event is sent when a new instance of a type is created. It is sent from the context of the creator.

### Definition

Represents a general definition in the application. For each definition its name and GUID is stored, in order to establish a relationship with the original UML definition from which the C++ definition was generated. A meta definition also has a link to an owning meta definition, representing the definition in which the C++ definition is defined. This results in a tree of meta definitions that describes the structure of the application.

### Delete

This Event is sent when an instance of a type is deleted. It is sent from the context of the deleter.

### Event

This class is a common base class for all metaevents. Event is a subclass of InternalEvent. It defines a pure virtual `encode` operation which is implemented by all subclasses that represent concrete metaevents. Communication between the application and an EventMedia takes place by sending appropriate Event instances that represents what is currently happening in the application.

### EventManager

Every instrumented application has one (and only one) instance of this class. It is an active class and executes in a thread of its own. It maintains a list of EventMedias and has functions to register and unregister these. The event manager can be either active or inactive. When active it responds to requests for "processing" meta-events (i.e. descriptions about what is currently happening in the application). The processing includes encoding the meta-event to a string and emitting that string to all currently registered EventMedias. The event manager also is responsible for holding a representation of the current call stacks for each thread in the application.

Being an active class, the event manager has a statemachine. This statemachine represents the state the event manager is in. A special state is entered when the application reaches a breakpoint or is broken into by the Tau host debugger. In that state the event manager can handle all the various debug commands, such as Step and Go.

### EventMedia

Interface to a media onto which encoded metaevents can be emitted. The term "media" is used here in a very general sense; an event media can be any object that takes an encoded metaevent and does something with it.

### EventReceived

This Event is sent when an event of the application (i.e. not an InternalEvent) is received.

### EventSent

This Event is sent when an event of the application (i.e. not an InternalEvent) is sent.

### Exit

This Event is sent when the application is about to exit. It is used internally as a means for synchronizing with the EventMedias so that all metaevents that are pending will be processed before terminating the application.

### LogFile

This EventMedia writes each encoded metaevent to a logfile.

### Operation

Represents an operation in the application. There are three kinds of operations; constructors, destructors and ordinary operations. Operation inherits from Definition.

### Return

This Event is sent when an operation is returned from. It is sent from the context of the called operation.

### Returned

This Event is sent when an operation is returned from. It is sent from the context of the operation to which the return is made.

### StackContext

Representation of a stack frame. The stack frame belongs to a call stack, and there is one call stack for each thread in the application.

### StringCoderBuffer

This class realizes the CoderBuffer interface. It simply stores the encoding result as a string member. This is the standard class that is used to store encoded data in a TOR application.

### StructuralFeature

A structural feature represents an attribute or an operation parameter. It inherits from Definition.

### TauHostDebuggerProxy

A proxy class representing the Tau host debugger in the TOR application. The EventManager uses this proxy for communicating with the Tau host debugger over a TCP/IP connection.

### TauHostTracer

This EventMedia sends each encoded metaevent to a running Tau application. The Tau application will produce a trace in the form of a sequence diagram, showing what the application is doing.

### U2P_Coder

This class realizes the Coder interface. It is a standard coder that uses a U2P (UML textual syntax) for representing the result of encoding.

# List of Files

The following sections describes the files of TOR delivered as a part of the Tau installation.

### Source and header files

The complete TOR source code is included in the Tau installation and can be found in the following folder in the installation directory:

```
addins\CppGen\Etc\TOR\CPP
```

These files are used when building C++ applications.

The table below lists all the files and the TOR declarations they contain.

| File | TOR declaration |
|---|---|
| tor.h | Common include file for including all files related to communication and state machines. Mainly intended for code generated by the C++ Application Generator. |
| torAnnotations.h | A few macros used for the purpose of updating the model with code changes (a.k.a. round-tripping). |
| torChoice.h | Representation of a UML choice. |
| torCoders.h<br>torCoders.cpp | Coder, U2P_Coder, CoderBuffer, StringCoderBuffer |
| torCommunication.h<br>torCommunication.cpp | Various classes used for TCP/IP communication. Can be used when building a distributed application that communicates over TCP/IP. |
| torCompletedEvent.h | CompletedEvent |
| torConnector.h<br>torConnector.cpp | Connector |
| torContainers.h | String |

| File | TOR declaration |
|---|---|
| torDispatchable.h<br>torDispatchable.cpp | Dispatchable, EventExecutor, EventReceiver |
| torDispatchableClass.h<br>torDispatchableClass.cpp | DispatchableClass |
| torDispatcher.h<br>torDispatcher.cpp | Dispatcher |
| torDispatcherThread.h<br>torDispatcherThread.cpp | ThreadedDispatcher, DispatcherData |
| torEntryPoint.h | EntryPoint |
| torEvent.h<br>torEvent.cpp | Event |
| torEventManager.h<br>torEventManager.cpp | EventManager |
| torEventManagerImpl.h<br>torEventManagerImpl.cpp | Generated statemachine implementation of the event manager. |
| torEventMedia.h<br>torEventMedia.cpp | EventMedia, LogFile, TauHostTracer |
| torEventQueue.h<br>torEventQueue.cpp | EventQueue |
| torExceptions.h | Definition of exception types. |
| torExitPoint.h | ExitPoint |
| torMeta.h | Common include file for including all files related to meta-data. |
| torMetaData.h<br>torMetaData.cpp | Definition, StructuralFeature, Operation, Application |
| torMetaEvents.h<br>torMetaEvents.cpp | Event, Call, Called, Create, Delete, EventReceived, EventSent, Exit, Return, Returned<br>In addition there is one meta event for each debug command. |

| File | TOR declaration |
|---|---|
| torMetaStackContext.h<br>torMetaStackContext.cpp | StackContext |
| torOperators.h | equal, implies, mod, power and other functions representing UML operators. |
| torOs.h<br>torOs.cpp | Common include file for all OS abstraction layer files. |
| torOs_X.h<br>torOs_X.cpp | Implementations of Operating System Abstraction Layer. X represents operating system name.<br>There exist two files torOs_any.h and torOs_any.cpp that can be used as a template when doing an implementation of the OS abstraction layer for a new OS. |
| torOSRep.h<br>torOSRep.cpp | Switches between which actual OS files to use depending on how the TGTOS macro is set. |
| torPlatform.h | Platform specific build settings for the TOR library. This file is included by all implementation files of the TOR library. |
| torPort.h<br>torPort.cpp | Port |
| torProcess.h<br>torProcess.cpp | Process |
| torRegion.h<br>torRegion.cpp | Region |
| torState.h<br>torState.cpp | State |
| torStateMachine.h<br>torStateMachine.cpp | StateMachine |
| torSync.h<br>torSync.cpp | Mutex, RWLock, Semaphore, Gate |

| File | TOR declaration |
|---|---|
| torThread.h<br>torThread.cpp | Thread |
| torThreadSafeEventQueue.h<br>torThreadSafeEventQueue.cpp | ThreadSafeEventQueue |
| torTime.h<br>torTime.cpp | Time |
| torTimer.h<br>torTimer.cpp | TimerEvent, TimerObject, TimerQueue |
| torTopRegion.h<br>torTopRegion.cpp | TopRegion |
| torTypes.h | Simple types, Any class, Charstring class |
| torUtilities.h<br>torUtilities.cpp | sendTo, cast, setTimeUnit, initializeModel |

# TOR Integration guide

This guide describes how to adapt the C++ run-time library to a new operating system.

Included in the installation are a few examples of integrations with operating systems. These should be used as a starting point.

## OS Primitives

This section describes the “low-level” operating system primitives used by the C++ run-time library. Each of these primitives is abstracted by the library in order for the generated code to be independent of the underlying operating system.

The primitives are logically located in the namespace “`tor::os`”. In most cases each primitive is abstracted by a class that then uses an underlying representation that implements the primitive in terms of the underlying operating system.

These “representations” are all located in the files named “`torOs_xxx.h`” and “`tor_Os_xxx.cpp`” (where “`xxx`” is the name of the operating system). The two files “`torOs_user.h`” and “`torOs_user.cpp`” are intended for user defined implementations of these representations. I.e. a “good start” for an integration to a new operating system is to copy one of the existing “torOs_xxx.cpp” and “torOs_xxx.h” to “torOs_user.cpp” and “torOs_user.h”, respectively and use that as a basis.

The file “torOsRep.h” includes one of the files named “torOs_xxx.h” depending on the value of the preprocessor macro “TGTOS”. The macro can have one of the following values (defined in “torOsRep.h”):

| name | value | description |
|---|---|---|
| `OS_USER` | `0x0100` | This is intended for user-defined integrations. |
| `OS_WIN32` | `0x0200` | This selects 32 bit versions of Microsoft Windows operating system. |
| `OS_LINUX` | `0x0300` | This selects the generic Linux integration. |
| `OS_SOLARIS` | `0x0400` | This selects the Solaris integration. |

If the macro “TGTOS” is not explicitly defined, a preprocessor construction in the file “torOsRep.h” selects one of the above values depending on the host operating system (i.e. on Linux the value OS_LINUX is selected).

### Time

Time is abstracted in the class “`tor::os::Time`” and is implemented in the class “`tor::os::TimeRep`”. The time class is used for representing both “absolute times” (in terms of the operating system) and “relative times” (durations). In the example integrations it has been chosen to coincide with the OS time representation that is used for timed-waiting on a semaphore and timed suspension of a thread. This prevents unnecessary conversion between a general representation and that of the operating system.

The class “`TimeRep`” must provide the following interface (the actual attribute that holds the time value has been omitted):

```
class TimeRep {
public:
    TimeRep();
    TimeRep(long s, long ns);
    TimeRep(const TimeRep &a, const TimeRep &b);
    TimeRep(const TimeRep &a);          //  = a
    ~TimeRep();

    void norm();
    void zero();
    void setNow();

    void set(const TimeRep &x);
    void set(long s, long ns);
    void set(double d);

    void add(const TimeRep &x);
    void add(long s, long ns);

    void sub(const TimeRep &x);
    void sub(long s, long ns);
    void sub(const TimeRep &a, const TimeRep &b);

    int cmp(const TimeRep &x) const;

    double to_double() const;
    long to_ms() const;

    std::ostream &print(std::ostream &os) const; //
optional
```

```
    static double setUnit(double u);

};
```

The default constructor "`TimeRep()`" must initialize the value to a logical zero so that:

```
TimeRep a, b; b.setNow(); a.add(b); if (a.cmp(b) == 0) {
OK; }
```

The "zero()" operation must yield the same value.

The constructor "`TimeRep(long s, long ns)`" initializes the representation using the given second and nanosecond (10-9s) value. The operation "`set(long s, long ns)`" must set the value in the same way.

The constructor "`TimeRep(const TimeRep &a, const TimeRep &b)`" initializes the representation with the logical difference "`a - b`". The operation "`sub(const TimeRep &a, const TimeRep &b)`" sets the same value.

The operation "norm()" is a helper function not normally used outside of the operations in the representation. It "normalizes" the value after the arithmetic operations.

The operation "`setNow()`" assigns the current time.

The operation "`int cmp(const TimeRep &x) const`" returns "-1" if the object itself is "less than" the object x, "0" if the two objects are "equal" and "+1" if the object itself is "greater than" the object x.

The conversions from and to "double" all use an internal unit multiplier which is set using the operation "`setUnit()`".

The actual representation of time is implemented as a "`struct timespec`" in POSIX solutions and as a pair "`long m_s, m_ns;`" under Win32.

### Mutex

The abstraction of a mutual exclusion lock is defined by the class "`tor::os::Mutex`" and is implemented in the class "`tor::os::MutexRep`". The uses of "`Mutex`" requires only the most basic from a mutual exclusion lock; only unconditional lock and unlock is used.

The class "`MutexRep`" must following interface (the actual attribute that holds the mutual exclusion lock has been omitted):

```
class MutexRep  {
public:
```

```
  MutexRep();
  ~MutexRep();
  Status init(bool initialOwner);
  Status destroy();
  Status lock();
  Status unlock();
};
```

**Note**

*The initialization and destruction of the mutual exclusion lock has been separated from the constructor and destructor, respectively (this is also reflected in the "Mutex" class). This allows for global instances of mutual exclusion locks that can be re-initialized. If this is not necessary, the user may choose to implement the initialization and destruction in the constructor and destructor ("*`init()`*" and "*`destroy()`*" should then both be empty).*

The "lock()" and "unlock()" operations are unconditional. They must return "`StatusOK`" or "`StatusFAIL`" depending on the success of the operation.

In the case of POSIX the attribute that holds the mutual exclusion lock is defined as "`pthread_mutex_t m_mutex`" and under Win32 "`HANDLE m_handle`". Under Win32 the handle is initialized using "`CreateMutex()`".

### Semaphore

The abstraction of a semaphore is defined by the class "`tor::os::Semaphore`" and is implemented in the class "`tor::os::SemaphoreRep`". The uses of "`Semaphore`" require; unconditional lock and unlock as well as timed-lock (where the calling thread blocks on the semaphore using a time-out).

The class "`SemaphoreRep`" must following interface (the actual attribute that holds the semaphore has been omitted):

```
class SemaphoreRep {
public:
  SemaphoreRep();
  ~SemaphoreRep();

  Status init(unsigned int initialCount, unsigned int
maxCount);
  Status destroy();

  Status put(unsigned int count);

  Status get();
  Status get(Time timeout);
};
```

Similarly to the mutual exclusion lock, the initialization and destruction of the mutual exclusion lock has been separated from the constructor and destructor, respectively. Note that the timeout in the "`get(Time timeout)`" operation is an absolute time.

The POSIX implementation uses three attributes to implement the semaphore:

```
pthread_mutex_t m_mutex;
pthread_cond_t m_cond;
unsigned m_count;
```

The Win32 solution uses a "HANDLE" initialized using "`CreateSemaphore()`".

### Thread

Threads are implemented using three operations in the namespace "`tor::os::Thread`":

```
bool createThread(ThreadFunc func, void *arg);
void suspend(Time timeout);
unsigned long id();
```

The "`createThread()`" operation creates a new thread with the "`ThreadFunc`" as thread function with a single argument. In the POSIX case, the thread is created in a "detached state".

The "suspend()" operation uses an absolute time. In the POSIX case this is implemented using "`nanosleep()`" and in Win32 using "`Sleep()`".

The "id()" operation returns an id number that is unique for each thread.

### Process

Processes are not used as such in "normal" integration and can be left out. In the case they are used the representation must provide the following:

```
class ProcessRep {
public:
    typedef Charstring::inherited string;
    ProcessRep();
    ~ProcessRep();
    bool launch(const string &cmd, const
std::vector<string> &argv, const string &dir);
    bool terminate();
    PID getPID() const { return m_pid; }
}
```

The constructor and destructor are empty. The “launch()” operation launches an external process. The “terminate()” operation kills the external process and returns when the process has terminated.

## Building

This section briefly describes the settings needed to use a user defined tool chain when building an application. The user should read the section on the Makefile Generator in connection with this section.

The starting point of this section is a model and an accompanying build artifact that is used for building the system. We also assume that the changes described in the previous section has been made and put in the two files “`torOs_user.cpp`” and “`torOs_user.h`”. The compiler, linker, make and other tools that are necessary for the build process must be available in the users path.

The first thing to do is to attach the stereotype named `<<MakefileGenerator Settings>>` to the build artifact this will allow us to change the behavior of the Makefile generator. The stereotype is defined in the addin “`MakefileGen`” and is activated from the Add-Ins tab. Apply the stereotype “`MakefileGen::Makefile Generator Settings`” to the build artifact (Note: there is a similar stereotype named “Makefile Generator”, without “Settings”, it should not be used in this case).

Change the settings of the stereotype so that “Target Kind” corresponds to the built-in settings that are closest to the users target system. If we now generate the code (by right-clicking on the build artifact and then selecting “Build (C++ Application Generator)->Generate” a Makefile will be generated (together with the C++ code). Most likely, the make variables used for the compiler etc. needs to be changed. This is done by adding entries in the “User Code” section of the “Makefile Generator Settings” stereotype on the build artifact. Each line in this section will override the built-in values if there is a corresponding make variable.

The preprocessor macro “TGTOS” must be defined to “OS_USER”; this is done by copying the current line that sets the preprocessor defines in the Makefile generated in the previous step. In the case that “`Target Kind = Linux - g++`” the line entered into the “User Code” section will look like:

```
DEFINES=$(TAUDEFINES) -D_REENTRANT -DTGTOS=OS_USER
```

and if “`Target Kind = Win32 - cl`” it looks like:

```
DEFINES=$(TAUDEFINES) /D "WIN32" /D "NDEBUG"
$(SUBSYSDEF) /DTGTOS=OS_USER
```

In both cases this is one line.

Similarly, if the C++ compiler, compiler flags, linker, make program etc. needs to be changed, they must be entered in the “User Code” section.

The full list of make variables and settings for the Makefile generator is found in the section “Makefile Generator” on page 854.

# 55

# Debugging a C++ Application

The UML Debugger allows you to debug the behavior of your UML model and verify that the implementation is correct.

When you are building an instrumented application for the UML Debugger, you are performing similar instructions as you are when building an application with the C++ Application Generator. This section lists the basic build functionality and it covers the basic usage of the UML Debugger.

**See also**

Chapter 11, Verifying an Application.

# Overview of the UML Debugger

The UML Debugger allows you to debug the behavior of your UML model and to debug that the implementation is correct. Using the Application Builder, you generate instrumented C++ code from your model and you link it with a predefined run-time library which is customized for simulation purposes. You can also use the UML Debugger if you are building an instrumented C++ library. To simulate the model means that you run the executable program using various commands and breakpoints. You can run the simulation automatically or can manually step through transitions, etc.

You can control the UML Debugger from the user interface or you can switch to a Visual Studio debug session. If your application communicates with the environment, this behavior can be followed with a target C++ debugger (Visual Studio), you use the UML Debugger for execution control and the Visual Studio debugger for inspecting/setting values, looking at threads and call stacks etc.

The execution of the session can be traced graphically in state chart diagrams, class diagrams and text diagrams.

# Generating an Instrumented Application

**Note**

*You must have a C/C++ compiler installed to generate an executable* UML Debugger *application.*

The UML Debugger requires an instrumented application to be generated. An application is instrumented by

1. adding additional instrumentation code when generating the application.
2. using an instrumented version of the TOR library which is linked with the application.

The instrumentation does not change the behavior of the generated application, but allows the application to be controlled during execution by the UML Debugger.

### Build settings

For each executable that is to be generated there must be a Build Artifact. To generate an executable that can be debugged this artifact must have settings for instrumentation enabled. This is set in the Properties dialog for the C++ Application Generator stereotype (stereotype properties are switched with the Filter menu field).

The following settings should be considered carefully and set to match your needs.

- Link with TOR
- Instrumentation Options
- Debug options

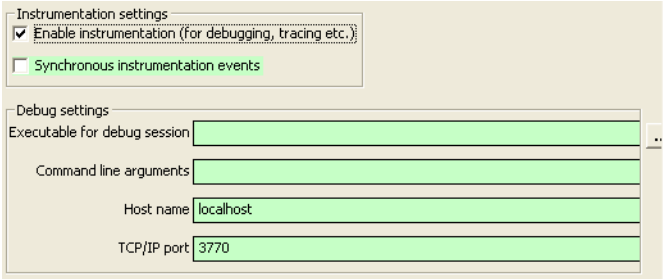



*Figure 258: Options for instrumentation and debugging*

The instrumented application is generated by building the build artifact. For more information on how to do this see Building Using Build Artifact.

Note that if a build configuration is used for building multiple build artifacts at the same time (see Using Configurations for Build), at most one of those build artifacts should be a UML Debugger build artifact. The reason for this is that only one instance of the UML Debugger is allowed to execute at the same time.

### See also

Chapter 27, Building Applications Reference

# Running the UML Debugger

You can start the UML Debugger after a UML Debugger build is completed without errors.

# Start the UML Debugger

You can either start the UML Debugger by attaching it to a running application or by doing Launch, which will build and start a UML Debugger from a build artifact. Applicable commands on the Verify toolbar become active.

### Attach to a running application

When attaching to an already running executable, it is required that this executable was built with the same build artifact used in the model.

1. Start the application, for example using Visual Studio, or directly from the command prompt.
2. Attach the UML Debugger to the running application by selecting the command ***Attach to running executable*** in the ***Verify*** menu. The build artifact should be selected for this command to be available.

As part of the attach procedure Tau performs some non-exhaustive checks that the attached-to process is instrumented as an UML Debugger.

If Tau doesn't detect a running debuggable application within 5 seconds it will ask if you want to continue waiting, or if you want to abort the attach procedure.

Sometimes an application is difficult to attach to because it may terminate before you have had the time to attach to it from Tau. In such situations you may set an environment variable `TTD_TOR_WAIT_FOR_ATTACH` before starting the application. If this environment variable is set the application will wait before executing the first statement, which gives you time to attach to it from Tau.

There is also another environment variable which can be set: `TTD_TOR_HOST_DBG`. This variable can be set to a string on the form "hostname#portnumber" to specify the TCP/IP settings to use for the debug session. You only need to set this variable if you want to change the default TCP/IP host or port being used.

**Launching after Building Using Build Artifact**

This method is useful if you already have created a build artifact for a UML Debugger.

1. In the Model View, right-click the build artifact manifesting the UML Debugger to launch.
2. On the shortcut menu, select **Build (**C++ Application Generator**)**, followed by the menu choice **Launch**.

A UML Debugger is built and, if the build is successful, the newly built UML Debugger is launched.

**Launching after Using Configurations for Build**

This method is handy in case you want to build multiple build artifacts, of which one is a UML Debugger, and launch the newly built UML Debugger. One special application is when your configuration contains exactly one UML Debugger build artifact.

- In the project toolbar, click the button **Execute Configuration**. This orders a build of all build artifacts contained in the active configuration and launches the newly built UML Debugger

## Exit the UML Debugger

To exit the UML Debugger, you can proceed with either of following:

- On the **Verify** menu, select **Stop.**
- Terminate the debugged application (for example by closing its console window).

# Tracing the Execution

When you are running your UML Debugger, you can easily obtain trace information of the execution. This allows you to track each transition and event in your application. You can select between different tracing methods.

- UML model tracking
- Sequence diagram tracing via the Visual Studio Integration.

## UML model tracking

This tracking method allows you to follow each action in your UML model. When the tracking starts, the diagram with the selected action opens. When the execution continues, the next action in the diagram is highlighted and so on.

A green triangle is inserted beside the symbol or the statement within a symbol that is about to be executed.

## Sequence diagram tracing

Sequence diagram tracing is enabled via the Visual Studio Integration. The general workflow to do a trace is the following:

- Open your workspace with the project containing the application to trace.
- Ensure that the code is generated with the Enable instrumentation option activated.
- Create and open a Visual Studio project using the Visual Studio integration.
- Build and run a debug session in Visual Studio.
- Set a breakpoint in the very beginning of the code. When the code halts on the breakpoint you activate the trace through the Tau Trace command (from the Tau toolbar). Select **Sequence diagram visualization**.

The trace should now start in the form of a sequence diagram in Tau.

**See also**

Tracing execution of Tau generated applications for a way to obtain a sequence diagram trace without using the Visual Studio integration.

# Executing the Application

There are several different commands available to control the execution of the application. Which commands that are available depends on the mode of the UML Debugger. There are two modes:

1. ***Run mode***
   This mode means that the application is currently running. Commands are available for setting and removing breakpoints, and for breaking the execution (to enter Break mode). See Commands available in Run mode and Breakpoint commands.
2. ***Break mode***
   This mode means that the application is currently not running, but is suspended waiting for a debug command. Commands are available for setting and removing breakpoints, and for running the application (to enter Run mode). See Commands available in Break mode and Breakpoint commands.

## Commands available in Break mode

When the UML Debugger is in Break mode execution can be resumed by using one of the commands for controlling the execution (available in the **Verify** menu and toolbar). In addition Breakpoint commands can also be used.

### Go

Resume execution of the application. It will run until a breakpoint is hit, or the Break Execution command is invoked, or the application terminates.

### Step Over

Let the application execute the next action. If the action involves a call to an operation, do not step into its implementation (step over it instead).

### Step Into

Let the application execute the next action. If the action involves a call to an operation, step into its implementation and break the execution on the first action of that implementation.

### Step Out

Let the application execute until the currently executed operation implementation is returned from.

### Run execution up to this point

This command is available in the context menu when clicking on an action in a diagram. The application will execute until it reaches the selected action.

## Commands available in Run mode

When the UML Debugger is in Run mode you can break its execution by using one of the commands for breaking the execution (available in the **Verify** menu and toolbar). In addition Breakpoint commands can also be used.

### Break Execution

The application will break its execution (and enter Break mode) as soon as possible.

## Breakpoint commands

Breakpoints can be set to allow you to stop the execution at positions that are of interest.

### Insert breakpoints

To insert a breakpoint:

1. Open the diagram that shows the action on which you want to insert the breakpoint.
2. Right-click on the symbol or in the text and select **Insert/Remove Breakpoint** from the shortcut menu. A red dot is added to the symbol frame or next to an action within a symbol frame to indicate that the breakpoint has been inserted.

You can also insert a breakpoint via commands in the **Verify** menu and from the **Verify** toolbar.

### Precise positioning of breakpoints

Below there are some examples of how to set breakpoints to achieve the exact position you desire.

- Statement: To set a breakpoint to a specific statement, when an action symbol contains several statements, position the cursor on the statement. This also applies to setting breakpoints in text diagrams.
- Nextstate: To insert a breakpoint on a nextstate action, insert it on the connector line going to the state symbol.
- Action-symbols and lines (Output, Decision, Stop, Return, Flow-line to nextstate, History, Task, Junction-symbol representing a join): the breakpoint is set on the corresponding action.
- Transition symbols and lines (Start, DecisionAnswer, Input, Junction-symbol representing a label transition, Transition line, Connect transition (i.e. guard symbol after composite state)): the breakpoint is set on the compound action of the transition (i.e. the first action that will be executed when the transition is taken)
- State (selected in text or in browser): the breakpoint is set on all nextstate actions referring to the state (i.e. it will hit whenever the state is entered)
- State-symbol: the breakpoint is set on all nextstate actions referring to any of the states that are included in the state symbol. It is not possible to set breakpoints on an asterisk state or a state with excluded states.
- Operation-body (selected in text or in browser): the breakpoint is set on the compound action of the body (i.e. the first action that will be executed when the body is executed).
- Operation (OperationSymbol, OperationLabel in Class-symbol, or selected in text): the breakpoint is set on the compound action of its body. If the operation has no body (abstract operation or interface operation), the breakpoint will not be set in the debugged application.
- StateMachineImplementation (selected in the Model view): the breakpoint is set on the first action of all start transitions

### Remove breakpoints

To remove a breakpoint:

1. Open the diagram that shows the action having a breakpoint to be removed.
2. Right-click on the symbol or in the text and select **Insert/Remove Breakpoint** from the shortcut menu.

### List breakpoints

You can see all breakpoints currently being set under the Model Verifier Session package in the Model View. By double-clicking these you navigate to the corresponding action.

### Compiled breakpoints

Sometimes it is useful to insert a breakpoint as persistent within the model itself. Such a breakpoint is represented in the model as a call to the operation `tor::debugBreak` which is available in the `tor` library package. Whenever that call is executed the application will enter Break mode.

The insertion of a compiled breakpoint modifies the model, and thus requires the application to be recompiled.

## Step into source

When you execute a stepwise execution of your application the selection can affect the outcome of a **Transfer control to target debugger** command.

When **Transfer control to target debugger** is pressed the code is checked for a current execution point, which exists if you are in a debug session that is stopped. If a source code reference for the current execution point exists this will be used (which is what would be expected). There are however entities that do not have any explicit source code reference, which results in the command button not being enabled. For such a situation it is possible to make a selection in the code that is about to be executed and then this current selection will be used as a reference for the **Transfer control to target debugger** command.

# Error Handling

Violations of the dynamic rules of UML cause dynamic errors during the execution of a simulation. Examples include (but are not limited to) null reference access, out-of-range container manipulation, memory access errors, not catching thrown exceptions etc.

If a dynamic error occurs the debug session is terminated.

# 56

# *Visual Studio Integration for C++*

The integration between Tau and Visual Studio facilitates building and debugging a generated C++ application. For more information refer to the documentation of the Visual Studio Integration.

# *UML and Requirements*

The chapters listed under UML and Requirements describe how to model requirements in UML and how to work with requirements defined in DOORS.

# 58

# *Modeling Requirements*

This chapter describes how to model requirements in UML.

# Getting started

To start modeling requirements in Tau:

- Start Tau and create a new project, or use an existing project
- Activate the Requirements add-in, see Activating the Requirements add-in
- Optionally, switch to the Requirement View
- Create a Requirement Diagram and start modeling.

**See also**

"Working together with DOORS" on page 1725

"Importing requirements" on page 1726

"Working with links" on page 2409

# Requirements add-in

The Requirements add-in adds support for modeling requirements in Tau.

The main features of the add-in are:

- It implements the Requirements profile
- A requirement centric model view, the Requirement View.
- A requirement centric property view Requirement Property View
- A number of Requirement Reports

To activate the add-in, see Activating the Requirements add-in.

## Activating the Requirements add-in

The requirements modeling feature is available as an add-in, which must be manually activated for every project you want to use it in. To activate the add-in:

1. In the **Tools** menu, select **Customize...**.
2. Click the Add-Ins tab and check the `Requirements` add-in.
3. Click **OK**.

## Requirement View

The Requirement View provides a requirements centric view over a model, focused on showing requirements and their relations.

To activate the Requirement View:

1. In the **View** menu, select **Reconfigure Model View...**
2. Select **Requirement View** in the dialog and click **OK**.

You can switch between the Standard View and the Requirement View at any time.

**See also**

"Model View" on page 19 in Chapter 4, *Introduction to Tau 4.2*

"Default Model View" on page 2462 in Chapter 91, *Dialog Help*

## Requirement Property View

The Requirement Property View provides a requirements centric view of the properties of an element.

To activate the Requirement Property View:

1. Select an element and open the Properties Editor.
1. Click **Options...**
2. Change the property view to `Requirement Property View`
3. Click **OK**.

You can switch between the Standard Property View and the Requirement Property View at any time.

**See also**

"Properties Editor Options" on page 90 in Chapter 6, *Working with Models*

## Requirement Reports

A number of reports are available for requirements and requirement relations.

The following reports lists requirements:

- Requirements Report
- Non-satisfied Requirements Report
- Non-verified Requirements Report

And the following report lists requirement relations:

- Requirement Relations report

Reports listing requirements are available as Basic reports or Detailed reports.

### Basic reports

These reports are always available and lists all requirements meeting the criteria in the selected element(s).

For each Requirement the following information is displayed:

- Name
- Id
- Text
- Heading

### Detailed reports

When using the Requirement add-in together with DOORS Formal modules, detailed reports are available in addition to the Basic reports.

The detailed reports are only available for packages representing formal modules or single requirements in formal modules. The reason is that the set of attributes is potentially different for each module.

In addition to the three columns of the basic reports, the detailed versions has one column for each attribute column visible in the corresponding DOORS formal module as specified by the Attributes

**Note**

*DXL Layout columns are not represented in Tau and therefore not present in the reports.*

### Requirements Report

Simply lists all requirements.

### Non-satisfied Requirements Report

Lists all non-satisfied requirements in the selected element(s). A requirement is considered not satisfied when it has no incoming Satisfy relations.

### Non-verified Requirements Report

Lists all non-verified requirements in the selected element(s). A requirement is considered not verified when it has no incoming Verify relations.

### Requirement Relations report

Lists all requirement relations in the selected element(s). All Requirement relations are considered.

For each relation the following information is displayed:

- The kind(s) of the relation
- The name of the source element
- The kind of the source element
- The name of the target element
- The kind of the target element

# Requirements profile

The Requirements profile extends UML with concepts for modeling requirements. The Requirements profile is standardized in the OMG as a part of SysML.

## Basics

The requirements profile defines a `«requirement»` stereotype representing a requirement, and a set of stereotypes for specifying relationships between requirements and other UML model elements. Together these concepts adds support for requirements specification and traceability.

It also introduces a new diagram type, the Requirements diagram for editing requirements and their relations.

All the concepts useful for requirements specification are listed and described in the following sections. Most concepts are specified in the Requirements profile, but some in standard UML and some in the UML Testing Profile. They are listed here for completeness.

## Requirement

A `«requirement»` is a stereotyped Class representing a requirement. A requirement has the following properties:

- Text - the text of the requirement
- Heading - the heading text of the requirement
- Id - the id of the requirement

Requirements can be nested to any depth to represent hierarchies of requirements.

Requirements easily can be created from the **New** menu in the **Model View** both when using **Standard view** and **Requirements view**. Additionally requirements can be created in Requirements diagrams.

### Visualizing requirements in diagrams

A requirement can be visualized in a Requirement Diagram. However, since a requirement is a stereotyped class, it is also possible to visualize it in other diagrams where classes can be visualized. This means that it is possible to visualize a requirement in most kinds of diagrams.

In most diagrams the easiest way to visualize a requirement is to drag it from the Model View and drop it on the diagram. However, in some diagrams dropping a class means something else than to visualize it. For example, in an Activity Diagram an object node typed by the dropped class will be created, and in a Sequence Diagram a lifeline typed by the dropped class will be created. To be able to visualize requirements also in these kinds of diagrams, the Requirement profile contains a Diagram Generator called "Visualize Requirement". To use it, drag a requirement from the Model View to a diagram using the right mouse button. Select **Visualize in Diagram - Visualize Requirement** in the context menu that appears to visualize the dropped requirement in the diagram.

## Requirement relations

The profile specifies a set of requirement relations, all are stereotyped dependencies. Each relation is described in its own section below. All requirement relations are Dependency links. For more information on how work with links, see Managing links.

### Trace relation

A «trace» relationship is a standard UML concept used to track any kind of changes between model elements. It's therefore also applicable to requirement traceability.

«trace» is not defined in the Requirements profile, but it is listed here since it is commonly used together with requirements.

### Copy relation

A «copy» relationship between two requirements specifies that the text of the client requirement is a read-only copy of text of the supplier requirement.

The copy relationship is a kind of master/slave relationship intended to be used to specify requirements re-use.

**Note**

*The text of the client is not automatically kept synchronized with the text of the supplier. This has to be done manually.*

### DeriveReqt relation

The «deriveReqt» relation is used to specify that a requirement (the client) has been derived from another requirement (the supplier). For example a system requirement in a SRD is typically derived from a user requirement in a URD.

### Refine relation

The «refine» relationship is a standard UML concept used to specify that a model element (the client) refines another model element (the supplier).

«refine» is not defined in the Requirements profile, but it is listed here since it is commonly used together with requirements.

### Satisfy relation

The «satisfy» relation is used to specify that a model element (the client) satisfies, or implements, a requirement (the supplier).

### Verify relation

The «verify» relation is used to specify that a test case (the client) verifies, or tests, a requirement (the supplier).

### Test Case

A test case is an operation specifying a test behavior for testing one or more model elements. Test cases are defined in the UML Testing Profile.

**Note**

*Test cases are only available when the UML testing profile support is active, see Activating the Testing Profile Support.*

## Requirement Diagram

A diagram for viewing and editing requirements and their relationships.

A requirement diagram can contain requirements, requirement relations and other related elements.

Only requirements, packages and requirement relations can be created using the toolbar. To show other kinds of element, drag them from the Model View to the diagram.

**Note**

*Not all elements are shown in the Requirements View, so in order to drag them from the Model View to the diagram it might be necessary to switch view to for example the Standard View.*

**Note**

*To create a requirements diagram you must use the Requirements view. See Requirement View for details on how to switch to this view.*

# 59

# *Working together with DOORS*

This chapter describes how to work with requirements using both Tau and DOORS. The main features are:

- Visualizing DOORS requirements in UML models
- Visualizing UML models in DOORS
- Establishing links between requirements and UML model elements.

DOORS requirements are visualized in UML according to the SysML requirements profiles. This profile is described in more detail in "Modeling Requirements" on page 1715.

UML models are represented in DOORS by surrogate modules. A surrogate module is a DOORS formal module that corresponds to a UML package. The DOORS surrogate module will contain one DOORS object for each UML model element that is exported to DOORS. In addition the diagrams in the UML model can be exported to DOORS and shown as images in the surrogate module.

# Importing requirements

Requirements are imported from DOORS using the DOORS Import Wizard.

To start the import wizard:

1. Select the `Model` node, or any other model element, in the Model View
2. Select **File->Import...**
3. Select **Import from DOORS**
4. Click **OK**

The DOORS Import Wizard is started.

## DOORS Import Wizard

The first thing the import wizard does is connecting to DOORS to present a tree-view of the DOORS database. If DOORS is not started it will be started, and you will be prompted to login.

Once connected, a tree-view representing the database is displayed, all projects, folders and formal modules are displayed. Each node in the tree-view has a check-box in front of it indicating if it will be imported or not.

1. Select the formal module(s) you would like to import.
2. For each module, optionally change the Import Settings
3. Click **Import >**

The import process is started. A progress bar indicates the progress and messages are listed in a list.

4. Click **Finish** to close the wizard

For information on the result of the import, see Import result. For a list of supported modifications to imported requirements see Modifying imported requirements.

### Import Settings

For each formal module, the following import settings can be specified:

- Baseline

  Specifies the baseline to use during import. By default the latest version is used.

- View

  Specifies the view to use during import. The “Standard View” is the default.

- File

  Specifies the name of the .u2 file in which the imported formal module is stored. The default file name is the name of the formal module.

The baseline and view (and filters applied in the view) both affects the result of the import, see Import result for details.

## Import result

The following section lists the elements created during an import of a formal module. The supported modifications to imported requirements are described in Modifying imported requirements.

### Formal module

A formal module is imported using the representation described in UML representation of DOORS elements. The formal module package is inserted at the top level of the model, and placed in the file specified by the Import Settings.

### Objects

Each object is imported using the representation described in UML representation of DOORS elements.

Only objects that are visible in the View (specified by the Import Settings) used during import are imported.

Object attribute values are imported into an instance of the «requirementAttribute» stereotype on the object. For details on the definition of the stereotype, see Attributes below.

### Attributes

Attributes are imported using the representation described in UML representation of DOORS elements. The set of imported attributes is based on the columns of the View (specified by the Import Settings) used during import.

The «requirementAttributes» stereotype is created and inserted into the formal module package.

**Note 1**

*DXL Layout attributes are not imported in Tau.*

**Note 2**

*The «requirementAttributes» stereotype shall not be changed manually.*

### Links

Links are always imported as Trace relations. If desired the relation type can be changed manually in Tau at any time.

The target of a link is specified using a symbolic reference. If an element matching the symbolic reference is loaded, the link will be bound to that element, otherwise the symbolic link is kept. When importing new modules symbolic links are resolved if possible.

## Modifying imported requirements

Imported requirements can be modified in Tau and the changes can be pushed back to DOORS, see Committing changes from Tau to DOORS.

In addition to changes within a formal module, the formal module package itself must not be changed or moved in the model. Moving a formal module to a different scope will make links unbound.

# UML representation of DOORS elements

This sections describes the UML representation of DOORS elements. It is compliant with the Requirements profile, but adds some extensions

## Formal module

A DOORS formal module is represented as a package with the `«formaModule»` stereotype applied. The properties of a formal module are listed in the table below.

| Property | Description |
|---|---|
| Id | The id of the corresponding formal module. |
| Baseline | The baseline used when importing the module. An empty value indicates that the current version is used. |
| View | The view used when importing the module. An empty value indicates that the standard view is used. |
| Link Module | The full path to the link module used by the integration when committing links to DOORS. If this value is empty, the default link module is used. |
| Last Synchronized | The date and time of the latest synchronization. |

**Note**

*The values of the formal module properties are set automatically during import and synchronization and should not be changed manually.*

*The only value that can be changed is the Link Module.*

## Object

A DOORS object is represented using the standard Requirement representation.

The mapping between requirement properties and DOORS object attributes is described in the table below.

| Property | DOORS Object Attribute |
|---|---|
| Name | Object Identifier |
| Id | Absolute Number |
| Text | Object Text |
| Heading | Object Heading |

**Note**

*The id of a requirement is set automatically during import and synchronization and should not be changed manually.*

### Tables

Tables are represented in a structured way. The table itself and each row is represented by a place-holder element, and the cells are the actual requirement objects.

```
Table
        Row1
                Cell1
                Cell2
                Cell3
        Row2
                Cell4
                Cell5
                Cell6
```

A table element is represented as a class stereotyped «table», and a row is a class stereotyped «row».

## Attributes

DOORS attributes are represented in UML using a stereotype called «requirementAttributes». Each Formal module contains a stereotype with this name representing the attributes available for objects in that module. This stereotype is created automatically during import of a formal module.

Instances of the «requirementAttributes» stereotype are applied to all objects within a formal module. These instances contain the actual attribute values.

**Note 1**

*Changes made to a view in DOORS have to be saved before they can be imported into Tau. For example if an attribute has been added. Always save the view before importing.*

**Note 2**

*The* `«requirementAttributes»` *stereotype is created automatically during import of a formal module and should not be changed manually.*

## Link

A DOORS link is represented using the standard Requirement relations representation, i.e. a stereotyped dependency, or Dependency link.

# Exporting a UML model to DOORS

Any UML package can be exported to DOORS using the command **Export to DOORS**. The command is available in the context menu for the package in the **Model View**.

When a Tau model is exported, a DOORS formal module is created. However, as the original objects of this module are not DOORS objects, the module is called “surrogate”.

To export of a UML model sub-hierarchy to DOORS, perform the following steps:

1. Select the item you want to be the root of your DOORS formal module in the **Model View**
2. Right-click on this item, select **Export to DOORS** from the shortcut menu.
3. If DOORS is not already running, it will be started now. After providing your login information, you can proceed with the export.
4. The **Export Module** dialog allows setting the parameters for the surrogate DOORS module.
   - Choose the **location** in the DOORS database.
   - Optionally fill in **Object identifiers** data: start number and prefix; or use the default values.
   - Press OK.
5. Choose the view which should be used for the export from the **Reconfigure View** dialog.

Tau now creates a formal module in DOORS, containing objects corresponding to the exported objects in Tau. You will also be prompted to save your Tau project, to ensure that the information is consistent between the two tools.

# Unexport of a DOORS module

It is possible to remove the association of a package with the corresponding DOORS surrogate module using the command **Unexport from DOORS.** This command will remove the relationship between a model in Tau and the corresponding formal module in DOORS. After this, it will no longer be possible to **Commit to DOORS** for the module. However, the corresponding formal module will still be present in DOORS.

The command is activated with the following steps:

1. Select the root object of the exported model in the **Model View**.
2. Right-click on the item, and select **Unexport from DOORS** from the shortcut menu.

# Locating an element in DOORS

To navigate from a UML model element to the corresponding DOORS element:

1. Right-click the element in the Model View or in a diagram
2. Select ***Locate in DOORS*** from the context menu

This operation will start DOORS if it is not running and try to locate the object. The view and baseline used during import is used to locate the object. If the view is no longer available or the object isn't visible in the view, the standard view is used.

This command is available both for UML elements that have been exported to DOORS, for packages representing imported formal modules and for UML <<requirements>> classes representing imported requirements.

The command is also availble in the DOORS tool bar and in the Doors menu.

# Committing changes from Tau to DOORS

It is possible to update both surrogate modules exported to DOORS and requirement modules that have been imported from DOORs.

To update a formal module in DOORS with changes made in Tau:

1. Select the formal module in the Model View
2. Right-click and select **Commit to DOORS**.

The command is also available in the Doors menu and in the DOORS tool bar.

**Note**

*Elements in the surrogate modules in DOORS should not be edited.Changes should always be made in Tau and propagated to DOORS. If the module is edited in DOORS the changes will be overwritten next time a commit is done from Tau.*

**Note**

*When committing for the first time in a new Tau version, it is important that the instructions given in Migrating from earlier Tau versions is followed. Failing to do so will result in loss of data.*

**Note 2**

*A formal module imported with a specific baseline can by definition not be updated.*

## Disable synchronization with DOORS

It is possible to specify that parts of the UML should not be transferred to DOORS. This is accomplished by the command **Enable synchronization with DOORS** that is availabe in the context menu for UML elements that have been exported to DOORS. This is a check-box menu entry that be default is selected. By de-selecting this menu choice for a UML element, the next time the UML model is committed to DOORS, the UML element will not be shown in the DOORS surrogate module. The operation can be reverted by selecting the **Enable synchronization with DOORS** command again, which will show the UML element in DOORS after the next commit.

## Show diagram image in DOORS

The **Show diagram image in DOORS** is available in the context menu for diagrams in the Model View. The command is a check-box menu entry that by default is de-selected. By selecting this menu choice for a diagram, the next time the UML model is committed to DOORS, the diagram image will be shown in the DOORS module. The diagram can be removed from the DOORS module by de-selecting the **Show diagram in DOORS** menu entry and then committing to DOORS.

## Update/Commit on Open/Save

When this option is enabled, each time the Tau workspace is loaded, all changes made in DOORS are automatically propagated to the UML model. Furthermore, when the UML model is saved, all changes to an exported model will be committed to the corresponding DOORS surrogate module. In both cases, if DOORS is not already running it will be started.

The option is enabled using the command **Update&Commit on Open&Save** that is available in the context menu for UML model elements that correspond to DOORS formal modules, both requirement modules and surrogate modules.

The **Update&Commit on Open&Save** is a check-box menu entry that be default is de-selected.

## Supported changes

The following table lists the set of supported changes to UML elements that will be committed to DOORS.

**Note**

*Be careful when editing formal modules. Not all of the non-supported changes are prohibited by the user interface. This is intentional to support less restricted requirement modeling when the DOORS integration isn't used.*

| Element | Supported changes | Restrictions/Comments |
|---|---|---|
| Requirement | Create, delete, move<br>Change of name and/or attribute values | The `Id` property should not be changed.<br>Requirements should not be moved between formal modules.<br>Requirements inside tables can't be created, deleted or moved.<br>Requirements in tables can't have child requirements. |
| Requirement relation (link) | Create, delete, move, change of target | The link module used by the integration to create links is specified as a property of the Formal module. |
| Requirement diagram | Any change is supported. | - |
| Formal Module | - | The attribute definition (i.e. the `«requirementAttributes»` stereotype) must not be changed.<br><br>Moving or renaming an imported formal module will cause problems when updating the module from DOORS as described in "Updating from DOORS to Tau" on page 1739. |
| Table | - | The only supported changes to tables is changing the properties of the contained requirements. |
| Row | - | The only supported changes to rows is changing the properties of the contained requirements. |
| Other UML elements | Any change is supported. | - |

Changes not explicitly mentioned in the table are not supported. This includes for example:

- Creation or deletion of table rows
- Addition of requirements to table rows

# Updating from DOORS to Tau

There are two ways to update formal modules in Tau with changes made in DOORS:

- Update with changes since last synchronization
- Full update

To update a formal module in Tau with changes made in DOORS:

1. Select the formal module in the Model View.
2. Right-click and select **Update from DOORS** or select **Full Update from DOORS** depending on the type of update that is needed.

Note that the behavior when updating Tau models with changes made in DOORS is different for surrogate modules, i.e. formal modules that have been created by exporting a UML package to a surrogate module in DOORS, and requirement modules. For surrogate modules any changes done to the DOORS objects are ignored, only changes to links to/from the objects are updated in the UML model. For requirements modules all changes, both to the requirements objects and the links to/from the objects are propagated to the UML model.

## Update with changes since last synchronization

The formal module will be updated with changes made to the formal module in DOORS since the last synchronization. This is an update optimized for performance, but the set of Supported changes is slightly limited.

**Note 1**

*A formal module imported with a specific baseline can by definition not be updated.*

**Note 2**

*Objects that have been deleted in DOORS are always deleted in Tau even if they are visible in the view in DOORS. Deleted objects shall be synchronized to Tau before they are purged, otherwise they will not be deleted from Tau.*

### Supported changes

The following changes in the DOORS module are supported:

- Creation and deletion of objects
- Move of objects
- Change of object attribute values
  - Only attributes present during the import are updated
- Creation and deletion of links

No other changes are currently supported and can cause unexpected results during the update procedure. This includes (for example):

- Change of view definition
- Change and application of filters
- Change of attribute definitions

If such changes are needed, the module has to be updated using a Full update, or reimported, see Importing requirements.

# Full update

The formal module is fully updated with all changes made in DOORS. After a full update, the module in Tau is fully synchronized with the DOORS version.

### When to use the full update

A full update is needed when any of the following changes have been made in DOORS:

- The view definition has been changed
  - For example by changing or adding attributes
- A filter has been changed or applied
- Objects have been deleted and purged without updating the formal module in Tau.

# Changing the view or baseline

To change the view and/or baseline of an imported formal module:

1. Select the formal module in the Model View
2. Right-click and select **Change View/Baseline...**
3. In the dialog, select the view and baseline you want to use and click **OK**.

When changing the view and/or the baseline, a Full update of the module is performed.

# Creating Links

Links can be created in several different ways as described in Managing links. However, the DOORS integration adds another mechanism to create links: Drag'n'drop between requirements in DOORS and UML elements in Tau.

To create a link from a DOORS element to a UML element, drag a requirement from DOORS and drop it on an element in the Model View in Tau.The drop from DOORS will work if the target model element has DOORS representation, i.e. it is imported from DOORS or it is exported to DOORS. Note that the link will automatically be created in DOORS only. To see the link in Tau the UML model must be updated from DOORS.

To create a link from a UML element to a DOORS requirement drag the UML element from the Model View to DOORS and drop it on a requirement. The result will be a <<trace>> dependency in the UML model.In this case it is not necessary for the UML element to have a DOORS representation. Note that no link is created in DOORS until the changes in the UML model has been manually committed to DOORS.

# Exporting requirements to DOORS

The most commonly used scenario when creating DOORS formal modules is to create them in DOORS and then import them into Tau. However, in some situations it can be useful to start the definition of formal module in Tau instead. The mechanism to achieve this is described in this section. A package containing requirements can be exported to DOORS as a formal module. To export a package:

1. Select the item you want to be the root of your DOORS formal module in the **Model View**
2. Right-click on this item, select **Export to DOORS** from the shortcut menu.
3. If DOORS is not already running, it will be started now. After providing your login information, you can proceed with the export.
4. The **Export Module** dialog allows setting the parameters for the surrogate DOORS module.
5. Choose the **location** in the DOORS database.
6. Optionally fill in **Object identifiers** data: start number and prefix; or use the default values.
7. Select the **Export as** choice as **Requirements package**
8. Press OK.

A new formal module is created in DOORS. The name of the formal module is the same as the name of the exported package. The name cannot be changed.

All requirements in the package are exported as DOORS objects. All other elements in the package are ignored. The name of the exported requirement is ignored, and after the export it is replaced with the object identifier of the corresponding DOORS object.

**Note 2**

*A package can only be exported once. If the Id property of the formal module is set, export is disabled.*

### Supported changes

Once a module has been exported, the same restrictions for changes as for imported modules apply. For a complete list, see Supported changes.

### Adding requirement attributes

Requirement attributes can't be specified or changed manually in Tau, so they will therefore not be exported. To add requirement attributes and values to an exported module:

- Add the desired attributes and values in DOORS and save the view
- Switch to the new view in Tau, see Changing the view or baseline

This will update the module with the correct attribute definitions and values. From now on they can be used as in any imported module.

## See also

Common Workflows to Manage Traceability

# DOORS Menus

This section summarizes all commands made available inside DOORS by the DOORS integration. The commands are available in the following menus>



## Available commands in database explorer menus

**Create UML Model...**

This command will start up a wizard with the objective of creating a new project in Tau.

You will be prompted for the project name and location. Once this has been gathered, Tau will be started and a new project and UML package will be created.

Tau will then export this information to a DOORS surrogate module.

**Open in Tau**

This option is only enabled if there is a currently selected module in the Database Explorer.

For modules containing surrogate information, a Tau client will be started with that particular workspace loaded.

It is not possible to open modules containing requirement information (i.e. not surrogate modules) in Tau.

**Start Tau**

This option will start a Tau client with no particular workspace loaded.

**Help**

This option opens the help file containing information about the integration.

**About Integration...**

This option will display a dialog with information about the integration, e.g. copyright and version number.

## Shortcut menu for database explorer

In the database explorer, the following commands are available in the shortcut menu (under the Tau submenu):

**Create UML Model...**

**Open in Tau**

**Start Tau**

## Surrogate module menus

The commands described below are found in the **Tau** menu.

**Update from Tau**

This option is only enabled for modules containing surrogate information. It will make a call to Tau in order to begin an export operation. This will in turn cause the Tau client to be started if it is not already running.

**Commit Links to Tau**

This command will propagate outgoing links from the current module to the corresponding Tau model. It will make a call to Tau to begin an import operation. This will cause the Tau client to be started if it is not already running.

**Locate**

This option is only enabled for modules containing surrogate information.

This will cause the corresponding Tau item for the selected surrogate entry to be selected. The Tau client will be started if it is not already running.

**Filter (Sub-Menu) (UML only)**

The Filter function provides an important aspect of the integration allowing traceability between requirements and design.

**Identify Design Elements Not Justified by Requirements**

This option is only enabled for modules containing surrogate information.

It will query all out-links in the object hierarchy and filter on out-links that do not have a corresponding entry in another (assumed to be a requirements) module.

Note that if the description for the Tau surrogate module is changed, this function will not work as expected.

**Identify Design Elements by UML Kind (UML only)**

This option will present the user with a list of all UML kinds present in the surrogate module. A filter will then be applied so that only objects with these values will be displayed.

**Open in Tau**

**Import Links from Rational Rose**

This command lets you Preserve DOORS links in an model imported using the XMI import.

**Help**

**About Integration...**

## Shortcut menu for surrogate modules

In a formal module, when a requirement is selected, the following commands will be available in the shortcut menu (under the Tau submenu):

**Locate**

## Requirements module menus

The commands described below are found in the **Tau** menu.

**Commit to Tau**

If the requirements module already has been exported to Tau, the corresponding Tau project will be used. Otherwise, you will be prompted for the name and path to a Tau project which this module should be imported to.

A call to Tau will be made for it to begin an import operation. This will in turn cause the Tau client to be started if it is not already running.

**Update Links from Tau**

This command will propagate the outgoing links from the Tau model that corresponds to the current DOORS module into DOORS. It will make a call to Tau to begin an export operation. This will cause the Tau client to be started if it is not already running.

**Open Linked Surrogate Item in Tau**

This option allows you to open linked surrogate entries in Tau.

If in-links exist from multiple surrogate entries then a dialog will be displayed allowing a specific selection to be made.

**Create UML (Sub-Menu) (UML only)**

This allows creation of specific UML elements in Tau, these will then cause an update in the corresponding surrogate module prior to creating an out-link to the original object (assumed to be a requirement).

The link will be created in DOORS; to update Tau with the new link, use Commit Links to Tau from the Tau surrogate module.

This operation currently only allows you to create use cases.

**Use Case (UML only)**

This command will launch a wizard dialog. This dialog will guide you through the following steps:

- **Surrogate module**, the module corresponding to the Tau workspace that the use case will be placed in.
- **Context**, contains a listing of the entities in the surrogate module, allowing you to specify the context of the use case.
- **Details**, contains a text field for the use case name and tabs for **Description** and **Constraints**. The name field is mandatory. Description and Constraints fields can be reordered with “+” and “-” options.

Providing that all fields have been entered correctly, the data will be sent to Tau and the Use Case will be created in the selected workspace as well as in the surrogate module.

**Link Requirement to Selected Item in Tau**

This will create an in-link to the selected object from the corresponding entry in the surrogate module for the currently selected object in the Tau workspace.

**Filter (Sub-Menu) (UML only)**

**Identify Requirements Not Addressed by Design Elements**

This will query all in-links in the object hierarchy and filter on in-links that do not have a corresponding entry in any surrogate module.

When filtering all ancestors of accepted objects will be shown. This will preserve the heading hierarchy and provide help to place each requirement.

Note that if the description for the Tau surrogate module is changed, this function will not work as expected.

**Start Tau**

**Help**

**About Integration...**

## Shortcut menu for requirements modules

In a requirements module, when a requirement is selected, the following commands will be available in the shortcut menu (under the Tau submenu):

**Open Linked Surrogate Item in Tau**

**Create UML (Sub-Menu) (UML only)**

**Link Requirement to Selected Item in Tau**

# DOORS toolbar

It is possible to perform some of the previously described operations using the buttons of the DOORS toolbar in Tau:

- **Start DOORS**: This starts DOORS if it is not already started. After providing the login information, Tau will be connected to DOORS.
- **Import a DOORS formal module**: This will launch the DOORS Import Formal Module wizard.
- **Locate an element in DOORS**: This starts DOORS and selects a given object into DOORS.
- **Export to DOORS**: This will export the selected element (and contained elements) to DOORS.
- **Update from DOORS:** This will update the currently selected elements with changes in DOORS.
- **Commit to DOORS:** This command will commit all changes done to the selected exported/imported element to DOORS.

# Common Workflows to Manage Traceability

This section describes how to work with both Tau and DOORS to establish and maintain traceability between UML models and requirements.

Three main workflows can be identified:

- Traceability in Tau
- Traceability in DOORS
- Traceability in Tau and DOORS

The last workflow is the most advanced, but also the most common one.

## Traceability in Tau

This workflow is primarily intended for analysts, architects, designers and developers that have a specified set of DOORS requirements to work with, but they don't necessarily need to look at the requirements in DOORS. Maybe they don't even have regular access to the DOORS database.

The important thing for this kind of user is that it is possible to establish and maintain traceability from model to requirements. The traceability information is not propagated to DOORS in any way.

The following steps are needed to set up this workflow:

- Import the requirements into Tau as described in Importing requirements

Once this is done, Dependency links are created between the model elements and the imported requirements to establish traceability.

Requirement Reports and diagram generators (see Generate Diagram) are used for traceability analysis. The **Generate dependency view ...** diagram generators are useful for creating graphical traceability diagrams.

If the requirements change in DOORS, they are updated as described in Updating from DOORS to Tau.

If there's a need to change requirements or adding links between requirements in Tau, it can be done and the changes committed to DOORS as described in Committing changes from Tau to DOORS.

This workflow does *not* support:

- Off-the-shelf advanced traceability analysis

  While Tau has the advantage of graphical traceability analysis it lacks some of the advanced traceability features found in DOORS.
- DOORS representation of links from/to/between model elements.

  Since the model is not present in DOORS there's no way to import these links into DOORS as standard links. (It is possible to use external links in DOORS to create links from requirements to model elements though. The integration never creates such links.)
- DOORS representation of UML models

## Traceability in DOORS

This workflow is primarily intended for requirement engineers, analysts and architects that have a specified set of DOORS requirements and need to verify that they have been properly addressed by the UML model(s) in Tau. They don't necessarily need to work with the model in Tau. Maybe they don't even have regular access to the Tau models.

The important thing for this kind of user is to be able to verify that all requirements have been properly addressed by the UML model by using DOORS links and traceability analysis.

The following steps are needed to set up this workflow:

- Export the Tau model to DOORS as described in Exporting a UML model to DOORS

Once this is done, DOORS links are created between the requirements and the surrogate model elements to establish traceability. Note that links between imported model elements will be present in DOORS.

The traceability tools in DOORS are used for traceability analysis.

If the model(s) change in Tau, they are committed to DOORS again using the **Commit to DOORS...** command as described in Committing changes from Tau to DOORS.

This workflow does *not* support:

- Graphical visualization of requirements and traceability analysis in UML diagrams

  Since the requirements and the links are not present in Tau, the diagram generators can't be used.

- Changes to the UML model in DOORS

  The UML model can't be changed in DOORS, a combination of Tau and DOORS is needed for this

## Traceability in Tau and DOORS

This workflow is a combination of the two previously described workflows, Traceability in Tau and Traceability in DOORS. It is primarily intended for architects and designers that need to work with both requirements and UML models and regularly use both Tau and DOORS.

This type of user needs to perform traceability analysis both in Tau and in DOORS, and needs to be able to synchronize the information in both directions.

The following steps are needed to set up this workflow:

- Import the requirements into Tau as described in Importing requirements
- Export the Tau model to DOORS as described in Exporting a UML model to DOORS

Once this is done, changes can be made in either Tau or DOORS and then be synchronized to the other tool.

**Note**

*Changes should only be made in one tool before synchronizing. If changes have been made in both tools, they can be discarded during synchronization.*

### Working in Tau

Dependency links are created between the model elements and the imported requirements to establish traceability.

Requirement Reports and diagram generators (see Generate Diagram) are used for traceability analysis. The **Generate dependency view ...** diagram generators are useful for creating graphical traceability diagrams.

If the requirements change in DOORS, they are updated as described in Updating from DOORS to Tau.

If there's a need to change requirements or adding links between requirements in Tau, it can be done and the changes committed to DOORS as described in Committing changes from Tau to DOORS.

### Working in DOORS

DOORS links are created between the requirements and the surrogate model elements to establish traceability. Note that links between imported model elements will be present in DOORS.

The traceability tools in DOORS are used for traceability analysis.

If the model(s) change in Tau, they are committed to DOORS again using the **Commit to DOORS...** command as described in Committing changes from Tau to DOORS.

### Keeping consistency between Tau and DOORS

To ensure that the requirements and the models are kept up to date and consistent, a simple procedure is used.

**Important!**

*Never change the same piece of information in both tools without synchronizing between the changes.*

*As an example, if a link is deleted in DOORS, but not in Tau, it will be reinserted into DOORS when the Tau model is committed.*

To synchronize changes made in DOORS into Tau:

- In the Model View tab, execute **Update from DOORS** for all imported formal modules and exported UML packages.

To synchronize changes made in Tau into DOORS:

- In the Model View tab, execute **Commit to DOORS** for all imported formal modules and exported UML packages.

# Migrating from earlier Tau versions

In order to preserve information in Tau and DOORS when migrating imported requirements between Tau versions, it is vital that the instructions below are followed.

**Important!**

*If these instructions are not followed, information can be lost in Tau and/or DOORS. Since the integration automatically saves the data in both tools, there is no easy way to recover lost data, other than reverting to previously stored/baselined versions.*

## UML Requirements

This section applies when using the UML representation of requirements, potentially together with exported UML models (see Exporting a UML model to DOORS).

In the old Tau version:

- Execute **Commit to DOORS...** for all exported UML models (surrogate modules)

In the new Tau version:

- Execute **Full Update from DOORS** for all formal module packages.
- Execute **Update Links from DOORS...** for all exported UML models.
- Select **File->Save All**.

## Requirements in .dim files

This section applies when using the old representation of requirements (.dim files) potentially together with exported UML models (see Exporting a UML model to DOORS).

In the old Tau version:

- Execute **Commit to DOORS...** for all exported UML models (surrogate modules)
- Execute **Commit Links to DOORS...** for all imported formal modules

In the new Tau version:

- Remove the .dim files from the Tau project (and optionally delete the files from the file system)
- Reimport the formal modules using the new integration, see Importing requirements
- Execute **Update Links from DOORS...** for all surrogate modules
- Select **File->Save All**.

**Note**

*Mixing the two different requirement representations, i.e. .dim files and UML models is not supported. Doing so can result in loss of data.*

## Surrogate Modules Exported using Tau 3.1.1 Previous Versions

This section applies when using surrogate modules that were exported using Tau 3.1.1 and previous versions. In Tau 4.0 the scheme used for exporting modules has been simplified and the intermediate representation shown in the DOORS tab has been removed.

The consequence is that all exported modules must be re-exported to DOORS.

In the old Tau version:

- Execute **Update Links from DOORS...** for all exported UML models (surrogate modules) to make sure all links are available in the UML model.

In DOORS

- Remove (or rename) the surrogate module.

In the new Tau version:

- Remove the .dim files from the Tau project (and optionally delete the files from the file system)
- Re-export the formal modules using the new integration, see Exporting a UML model to DOORS
- Execute **Update Links from DOORS...** for all surrogate modules
- Select **File->Save All**.

## Commit warning dialog

As a reminder of the migration procedure described above, a dialog is displayed when committing requirements or links to DOORS.

The dialog is a reminder of the importance to run an update from DOORS before doing a commit.

This dialog is displayed every time a module is committed in both the old and the new integration. To disable the warning, check the **Don't show this message again** check-box. Note that this will disable the dialog for all projects on the computer no matter if they have been updated or not.

To enable the dialog again if it has been disabled:

- Double-click the file `EnableCommitWarningDialog.reg` found in the folder `addins/Requirements/etc` of the Tau installation.
- Answer **Yes** to the question in the following dialog, and click **OK** in the confirmation dialog that follows.

# *Testing UML Models*

This section describes the use of the Test Profile in Tau.

# 61

# *UML Testing Profile*

This section describes how to use the UML Testing Profile support to test UML models.

The UML Testing Profile is an extension to UML adding concepts for specifying tests.

The testing profile support facilitates testing of a wide variety of systems, but it is primarily intended for black-box testing of active classes. The key features include support for test specification, execution and logging.

For a quick start read the following sections:

- Activating the Testing Profile Support
- Creating a test model
- Building and Running test applications

# Activating the Testing Profile Support

The UML Testing Profile support is implemented as an add-in, and it must be manually activated for every project you want to use it with. To activate the add-in:

1. Select **Customize** from the **Tools** menu.
2. Click the Add-Ins tab and check the `TestingProfile` add-in.
3. Click **OK**.

# UML Testing Profile

The UML Testing Profile is a UML profile standardized by the OMG extending UML 2 with test modeling capabilities. It extends UML with test concepts like test cases, test components and verdict.

The most important concepts of the testing profile is briefly described in the following sections. For more details on the testing profile read the profile specification that can be downloaded from OMG.

The profile is implemented as a library called `TTDTestingProfile` loaded when the testing profile support is activated. It contains all definitions from the profile specification.

## Definitions

### Arbiter

Responsible for test arbitration, i.e. maintaining the verdict of a test case. Stores the current verdict of a test case and provides a way for the test components to set and get the verdict. There is one arbiter in each Test context.

The Arbiter interface specifies two operations:

```
getVerdict() : Verdict
```

Returns the current test case verdict.

```
setVerdict(v : Verdict)
```

Sets the verdict of the current test case.

### Scheduler

Controls test execution in a Test context. Manages Test components and execution of test cases within the test context. There is one scheduler in each test context.

### Test case

A stereotyped operation specifying test behavior. Specifies a set of stimuli sent to the SUT and the expected response. The Verdict is produced based on the correspondence between the actual response and the expected response.

### Test context

A class acting as a grouping mechanism for test cases. Typically all tests for a given class are contained in the same test context. The test context owns a set of test cases, the Test components participating in the test cases and an Arbiter and a Scheduler.

The test configuration, i.e. the initial configuration of test components, the arbiter and the scheduler are also specified by the test context.

### Test component

A class executing in Test case behavior. Communicates with the SUT according to the test case behavior and reports the verdict to the Arbiter.

### Test objective

The objective of a Test case; an informal description of what the test is supposed to test. A test objective is specified using a dependency from a Test context or Test case to the element they are supposed to test.

### SUT

The System Under Test, SUT, is an instance of the system or class the tests are designed to test. Each Test context contains one SUT.

### Verdict

A verdict is the result of a Test case, and can have four different values, listed in the following table:

| Verdict value | Meaning |
|---|---|
| pass | Test execution is successful and the SUT responded as specified by the Test case. |
| inconclusive | Test execution can't tell whether the SUT performs well or not. |
| fail | Test execution fails because the SUT does not behave as specified by the test case. |
| error | There's an error in the test system, e.g. a test case hangs in an infinite loop. |

# Creating a test model

To create a test model to test a specific class, the following steps has to be performed:

1. Create a Test context with the desired class as an SUT. The easiest way to do this is to use the Create test context dialog, but it is also possible to create a test context manually.
2. Create a Test case in the test context. Test cases can be created in a number of different ways, see Creating a test case, and there is also an option in the Create test context dialog to automatically add an empty test case.
3. Now you should specify the test case behavior, see Specifying test case behavior, for the new test case. Sequence diagrams has to be used for test cases in the test context, but State machine diagrams can be used to specify test cases in the Test component.
4. Repeat steps 2-3 and add as many test cases you need to specify the tests.

More than one test context, each with a different set of test cases can be created.

When the test model has been completed, a test application can be created and executed.

**See also**

"Building and Running test applications" on page 1780

# Creating a test context

To create a test context, select an active class, right-click and select **Create Test Context...**

The Create test context dialog is opened, initialized with the active class as the type of the SUT.

## Create test context dialog

This dialog is used to create a Test context. The different values entered into the dialog are listed in the table below.

| Value | Description |
|---|---|
| Owner | The owner of the test context. |
| Name | The name of the test context. |
| SUT Name | The name of the SUT part of the test context. |
| SUT Type | The type of the SUT part of the test context. |
| Test Component Name | The name of the test component part of the test context. |
| Test Component Type name | The type name of the test component of the test context. |
| Generate Test Configuration | Indicates whether a test configuration diagram should be generated or not. |
| Generate Test Case | Indicates whether an empty Test case should be generated or not. |
| Create Build Artifact | Indicates if a build artifact manifesting the test context should be created or not. |
| Save in new file | Indicates if the test context should be saved in a new file or not. |

After filling out all values and clicking **OK**, a test context is created using the information of the dialog.

A test context named *'Name'* is created in *'Owner'*, and placed in the same file as the owner. If *'Model'* is specified as the owner, the test context will be created as a root element in the model.

A Test objective dependency is added from the test context to the SUT Type.

A number of elements are created inside the test context, each described in more detail below:

- An SUT part
- A Test component part
- A number of Connectors
- A number of Dependencies
- A Test configuration diagram
- A Initial test case

In addition, if **Create Build Artifact** is checked, a build artifact manifesting the test context is automatically created.

If **Save in new file** is checked, the test context (and the build artifact if created) is saved in a new file as specified by a save file dialog.

### SUT part

The SUT part is a composite attribute with the «SUT» stereotype with name and type as specified in the dialog.

### Test component part

The test component part gets its name from the dialog, and a new active class is created, inside the test context, and set as the test component type. The «`testComponent`» stereotype is applied to the class.

The test component class has one port for each port in the SUT type. The Realized interface and the Required interface of the ports are exchanged to facilitate communication between the SUT and the test component, i.e. realized interfaces on an SUT port becomes required interfaces on the test component port, and vice versa.

### Connectors

For each pair of ports of the SUT part and the test component part, a connector is generated and connected to the ports.

### Dependencies

To make sure that all definitions visible in the SUT type also are visible in the test context, import and/or access dependencies are automatically created from the test context to all needed definitions.

### Test configuration diagram

Optionally, a Composite structure diagram describing the test configuration, can be created in the test context. The diagram contains two part symbols, one for the SUT and one for the test component. Each port of the SUT and test component is represented and the ports are connected with connectors.

### Initial test case

Optionally, a initial Test case can be created in the test context. The test case is an operation with the «testCase» stereotype applied. The test case contains one sequence diagram with two lifelines, one for the SUT and one for the test component part.

# Creating a test case

A test case is an operation with the «testCase» stereotype applied. Test cases can only be owned by Test contexts or Test components.

How to create a new empty test case is described in the following sections:

- Adding an empty test case to a test context
- Adding an empty test case to a test component

How to create a test case by reusing an existing sequence diagram is described in the section Creating a test case from an existing diagram.

## Adding an empty test case to a test context

To add a Test case to a test context, right-click the test context in the model view and select **Create Test Case**.

A new test case is added to the test context. The test case contains a sequence diagram with one lifeline for each part in the test context.

The test case behavior can now be specified manually by adding messages between the lifelines and/or references to other test cases and operations in the test context. For details, see "Sequence diagrams" on page 1772.

## Adding an empty test case to a test component

To add a Test case to a test component, right-click the test component in the model view and select **Create Test Case**.

A new test case is added to the test component. The test case contains an empty state machine diagram.

The test case behavior can now be specified manually with State machine diagrams.

## Creating a test case from an existing diagram

To create a Test case from an existing sequence diagram, for example a trace diagram from the Model Verifier, right-click the diagram in the Model View and select **Add to Test Context**.

This opens a sub menu with a list of all test contexts in the model. Click the test context the new test case should be created in. The Add to Test Context appears. The test case is created when clicking **OK** in the dialog.

Since the only supported elements in a test case sequence diagram are messages, timing constraints and reference symbols, only these elements will be copied to the test case diagram.

### Add to Test Context

The **Add to Test Context** dialog contains four list boxes that contain lifelines, and buttons for moving lifelines between these list boxes. The purpose of the dialog is to group the lifelines into different categories to determine how they should be interpreted in the test case.

When the dialog is opened, the **Lifelines** list box to the left contains all the lifelines of the selected diagram. (Potentially one lifeline is preselected as an SUT lifeline.) Lifelines left in this list when closing the dialog are not copied into the new test case.

The **SUT** list box to the right contains the lifeline that should represent the SUT in the new test case. If any of the lifelines of the original diagram has a type that matches the type of the SUT in the selected test context, this lifeline will be present in the SUT list box instead of the Lifelines list box. There must be exactly one SUT lifeline in order to click **OK** and close the dialog.

The **Test Component** list box to the right contains lifelines that represents the test component in the test contexts. When creating the new diagram, the lifelines in this list box will be *merged into a single lifeline* representing the test component.

The **Test case attribute** list box to the right contains lifelines that each represent an attribute in the test case. Each of these lifelines will be copied into the Test case diagram. For each attribute lifeline that has a bound type, an attribute of the same type is created in the test case. If present, the lifeline is bound to the attribute.

Messages between lifelines in the **Test Component** list and messages between lifelines in the **Test case attribute** list are not copied to the new diagram.

When the lifelines have been moved to the desired list box, click **OK** to close the dialog. A new test case is created in the selected test context. It contains a sequence diagram derived from the original diagram, depending on the classification of the lifelines, as previously described.

The new diagram is automatically created, and the order of the messages and references follows the original diagram. The diagram is opened for editing after creation.

# Specifying test case behavior

The behavior of a Test case can be specified in two different diagram types, sequence diagrams and state machine diagrams. Sequence diagrams are used for test cases owned by the Test context, and state machine diagrams for test cases owned by the Test component. A single test case can only be specified by one diagram type, i.e. if you specify a test case as a sequence diagram in the test context you cannot specify the same test case as a state machine in the test component.

Test cases specify the **expected response** from the SUT when sending stimuli to it. Therefore a test case is always considered to `pass` if the SUT behaves as specified by the test case. The Verdict pass is implicit and does not have to be specified in the test case. You only have to set the verdict in a test case if you want a different verdict than `pass`.

Specifying test cases using sequence diagrams is the most common way, and they are used for a wide variety of tests. State machine diagrams are used for lower level test cases, when the expressive power of sequence diagrams is not powerful enough. Typically you should only use a state machine diagram if you can not express the desired behavior in a sequence diagram.

Test cases specified as sequence diagrams in the test context are transformed into state machines in the test component when creating the Intermediate test model, but test cases specified as state machine diagrams in the test component are left untouched.

## Sequence diagrams

A test case specified by a sequence diagram should contain at least two lifelines: one for the SUT and one for the Test component. The test case behavior is specified as a set of interactions between these lifelines.

The following symbols can be used in a test case:

- Lifelines
- Message lines
- Action symbols (only on the test component lifeline)
- Reference symbols
- Time specification lines

Example 587 on page 1773 illustrates how to create a simple test case.

**Example 587: A simple test case**

The test component sends the signal `Ping` with an integer parameter with the value `3` to the SUT. As a response the SUT sends the signal `Pong` with the same parameter value to the test component.

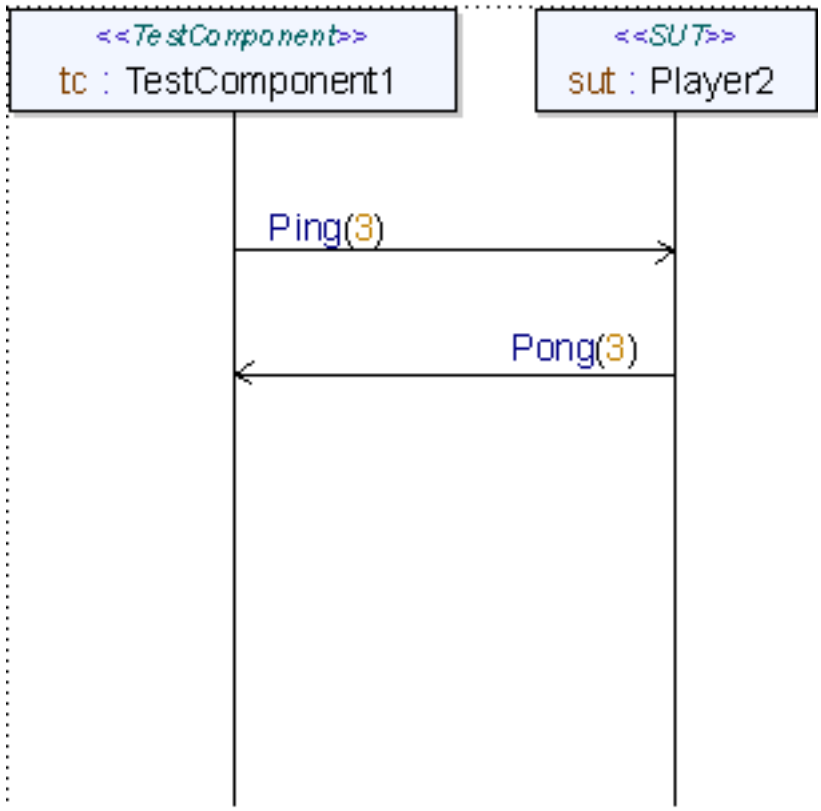



*Figure 259: Simple test case example*

The `Pong` signal with the parameter value 3 is the expected response from the SUT. In order for the test case to pass, this is the only acceptable response. If any other signal is received, or if the parameter value is different from 3, the test case will fail.

You don't have to set the verdict explicitly in this case. It is assumed that the test case will pass if the expected response is received, and fail otherwise.

### Using action symbols

An action symbol can be used to execute low level actions in a test case. This is typically needed to test the value of a parameter or to set a verdict different from `pass`.

- To set the verdict explicitly, call the `setVerdict` operation and supply the desired verdict value.
- To test a condition, typically a parameter or variable value, use the `assertTrue` operation.

Action symbols can only be used on the test component lifeline since it is not possible to make any assumptions on the internal behavior of the SUT in a test case.

**Example 588: Testing a condition**

Figure 260 on page 1774 illustrates how to use the `assertTrue` operation of ITTDArbiter to test the value of a parameter of a signal sent from the SUT to the test component.

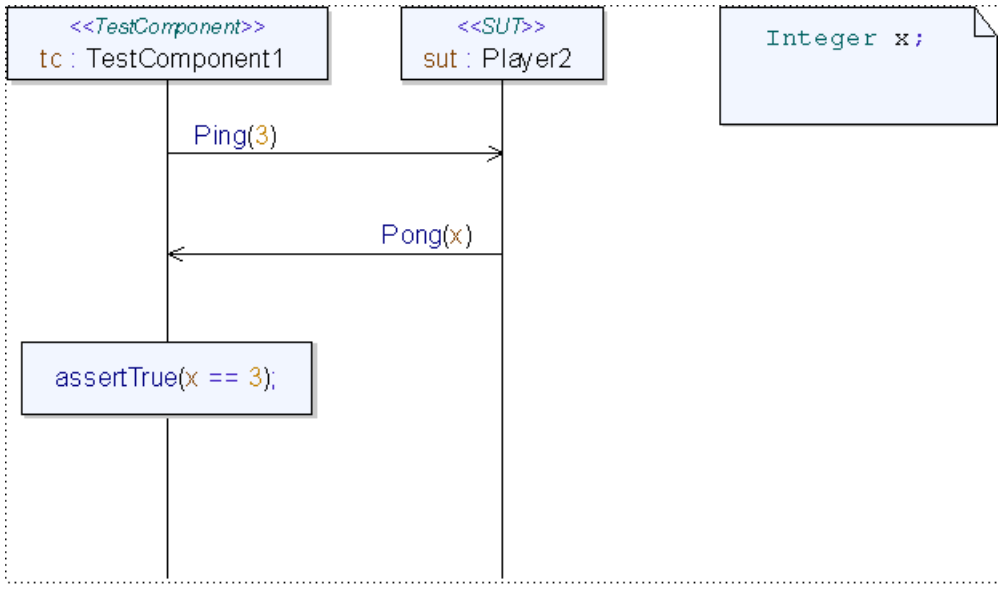



*Figure 260: Using the `assertTrue` operation to test a condition*

If the parameter value is `3`, the verdict will be set to `pass`, otherwise it will be set to `fail`.

### Referencing other test cases or operations

To reference a test case or operation from a test case, use a reference symbol in the sequence diagram and enter the name of the test case you want to refer to.

Use test cases if you also want to run them separately, and operations if you only want to run them as a part of other test cases.

**Note**

*The referenced test case / operation must be owned by the same test context and have the same set of lifelines.*

### Specifying timing constraints

Timing constraints are used to express constraints on the expected time between sending and receiving a message.

Timing constraints can only be used on the test component lifeline since it is not possible to make any assumptions on the SUT in a test case.

To specify a timing constraint, use a time specification line with both ends attached to the test component and enter the desired constraint in the text label. Three different kinds of constraints are supported:

- Within a given time value: < x, <= x

  If the time between sending and receiving the message is less than x seconds, the tests passes, otherwise it fails.
- After a given time value: > x, >= x

  If the time between sending and receiving the message is larger than x seconds, the tests passes, otherwise it fails.
- Within a closed range of time values: `{x..y}`

  If the time between sending and receiving the message is within the range, the test passes, otherwise it fails.

Constraints with a fixed absolute value, i.e. `{== x}` or `{x}`, are not supported, since they are not meaningful.

**Example 589: Timing constraint**

This example illustrates how to use a timing constraint in a test case. The test component sends signal `Ping` to the SUT and expects the signal `Pong` back within `3` seconds.

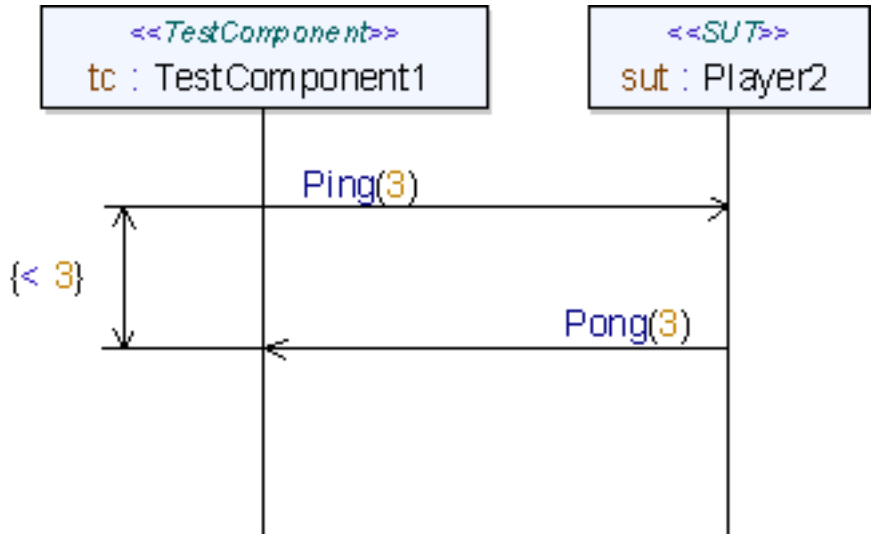



*Figure 261: Timing constraint example*

If signal `Pong` is received after more than 3 seconds, the test case will fail. If any other signal is received, or the parameter value is different from 3 it will fail no matter when the response is received.

---

### Additional lifelines

In addition to the SUT and test component lifelines, additional lifelines representing **attributes in the test case** may be used. This is only meaningful if the attribute is a reference to an instance of an active class. The type case is when the SUT passes a reference to an instance as a signal parameter and the test component wants to interact directly with that instance in addition to the SUT.

**Note**

*Currently there can only be three kind of lifelines in a sequence diagram. The should be exactly one representing the SUT, one representing the test component and any number of lifelines representing attributes in the test case itself.*

## State machine diagrams

There are no requirements or restrictions on a test case specified using a state machine diagram. The test case is assumed to pass unless the verdict is explicitly set to a different value.

To set the verdict explicitly, use an action box and call the `setVerdict` operation. To test a condition, use the `assertTrue` operation.

**Example 590: Test case specified using a state machine**

This example shows how the test case in Figure 259 on page 1773 could be specified using a state machine. The owner of the test case in this case is the test component rather than the test context.

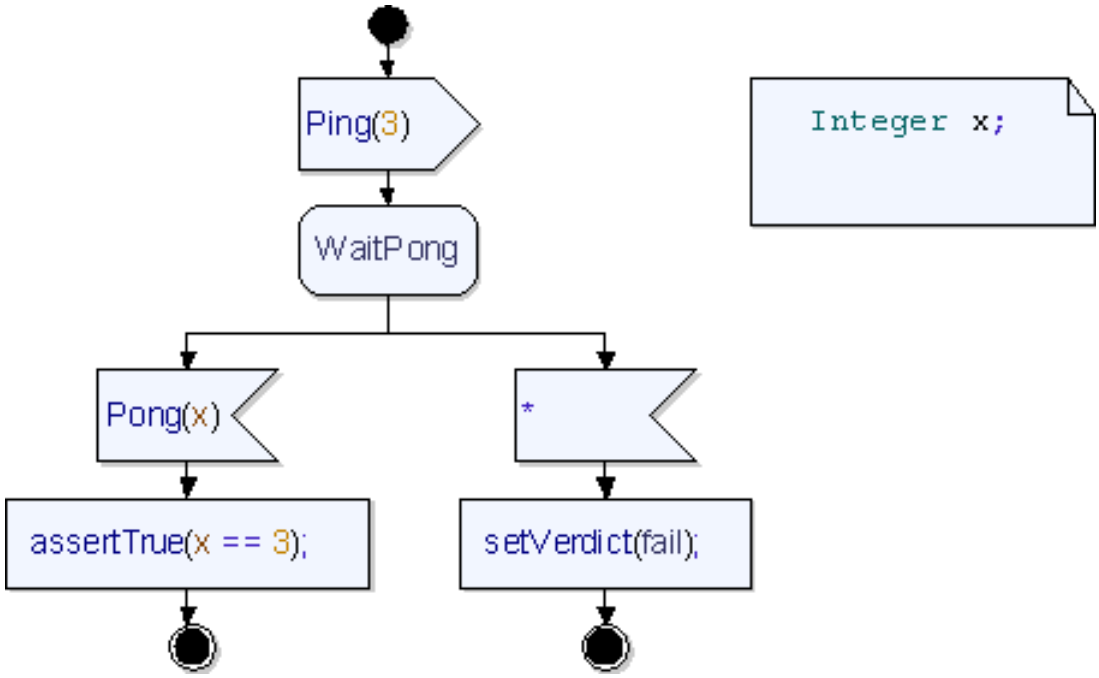



*Figure 262: Test case specified using a state machine*

# Test Framework

The testing profile implementation uses a test framework to implement the semantics specified by testing profile and to control test execution and logging. The framework is defined in a library called `TTDTestFramework`.

`TTDTestFramework` contains implementations of the interfaces from the testing profile and some additional elements. The key concepts of the framework are described below.

## Interfaces

### ITTDArbiter

An interface specializing the Arbiter interface. Two new operations are added:

```
assertTrue(b: Boolean)
```

A convenience operation for testing a condition and setting the verdict accordingly. The verdict is set to `pass` if the boolean expression `b` is true, and to `fail` if `b` is false.

```
getFinalVerdict() : Verdict
```

Returns the final verdict once a test case has been finished. Called by the Scheduler when finishing a test case.

### ITTDScheduler

An interface similar to the Scheduler interface, but adjusted to fit the framework. Defines two signals:

```
finishTestCase(ITTDTestComponent)
```

Used by test components to notify the scheduler when they have finished execution of a test case.

```
reportNewTestComponent(ITTDTestComponent)
```

Used by test components to notify the scheduler of their existence.

### ITTDTestComponent

A test component interface with one signal only:

`startTestCase(Charstring)`

Tells the test component to start execution of a test case. Used by the scheduler.

## Implementations

### TTDArbiter

An implementation of the ITTDArbiter interface. Stores the overall verdict of a test case in an attribute.

### TTDScheduler

An implementation of the ITTDScheduler interface. Drives test execution by retrieving a test case and executing it, keeping track of involved test components.

# Building and Running test applications

This section describes how to build and run a test application from a test context.

## Building a test application

To build a test application from a test context you need a Build Artifact manifesting the test context. Normally the build artifact is automatically created by the Create test context dialog, but it is also possible to create one manually:

1. Right-click the Test context and select **Test Generation -> New Artifact**.

   A build artifact is created and the Test generation stereotype is applied.
2. If desired, change the default values of the Test generation stereotype.
3. Apply a code generator stereotype to determine what kind of test application you would like to build. Currently only Model Verifier is supported.

To build the build artifact, use the Build command (Build Shortcut Menu).

If the build is successful the result is a test application that can be fed to the Test Driver for execution.

For information on how to execute the test application, see "Running a test application" on page 1785. The build step can often be skipped, since a build is performed automatically when running the test application if needed.

### Test application build process

During the build process, the original test model is translated into an Intermediate test model from which code is generated and then the test application is built. The process is outlined in Figure 263 on page 1781.

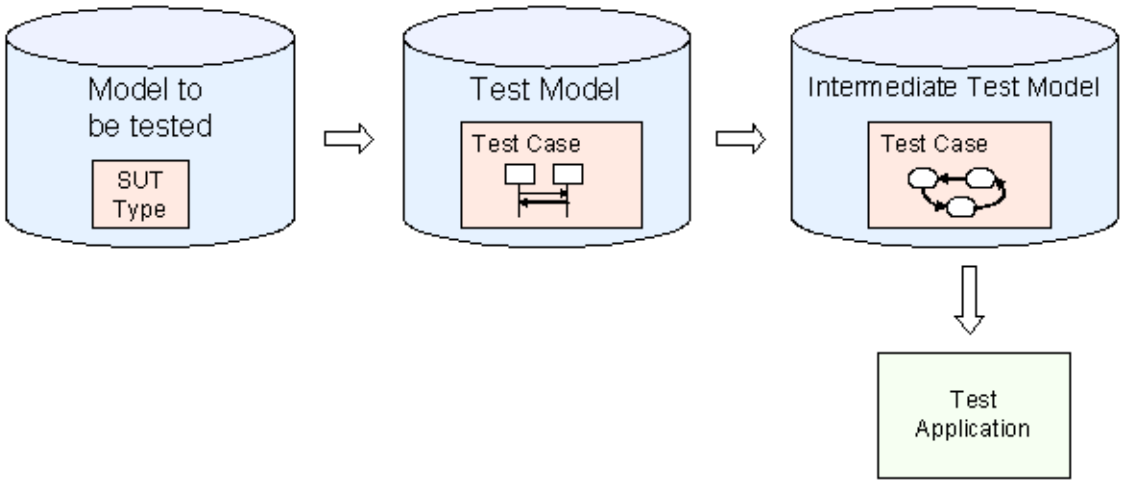



*Figure 263: Build process details*

The intermediate model is inserted into the project during build. For each build a new intermediate model is generated and the old one is thrown away. The intermediate model can however be saved and built separately. The intermediate model should not normally be changed, and if you make any changes to it, it is your responsibility to make sure the changes are compatible with the test framework and that the model builds.

## Intermediate test model

A test model is translated into an intermediate test model during the build process. A number of transformations take place to turn the test model into an executable test model.

For example, the behavior of each Test case specified as a sequence diagram, as described in Sequence diagrams, needs to be translated into something the chosen code generator can generate code from.

The transformations also ensure that the test model interacts properly with the Test Framework.

### Transformations

The table below lists all transformations performed during translation from a test model to an intermediate model.

| Test Model | Intermediate model |
|---|---|
| Test Case | Operation in the test component with the same name. |
| Test Case behavior (Sequence diagram) | State machine in the corresponding operation in the test context. (No diagram) |
| Test case attributes | Identical attributes in the corresponding state machine. |
| Message sent from test component | Signal Sending action with the same signal and parameters. |
| Message received by test component | See "Message reception" on page 1782. |
| Action symbol (`CompoundActionOccurrence`) | A compound action with the same content. |
| Reference symbol (`InteractionOccurrence`) | A call to the referenced signature, or more specifically an expression action with a call expression with the referenced signature and parameters. |
| Time specification line | See "Time specification line" on page 1783. |

### Message reception

A message received by a test component is translated to a state and a triggered transition in the state machine corresponding to the test case.

First the state machine will wait in a state until the given signal is received. The reception of any other signal is treated as an error (the operation will set the verdict to fail and will then return). If the specified signal is received the signal parameters are handled and after that the next action will be handled. Signal parameters are handled in either of two ways depending on the signal parameter expression. If the signal parameter is an attribute, then the value of the signal parameter is assigned to the attribute. If the signal parameter is anything else (a general expression), then the received signal parameter value is tested against this value. If the two values are not equal then this will be treated as an error (verdict will be set to fail and operation returns).

**Example 591:**

Reception of message:

```
sig(a, 1);
```

in a sequence diagram is translated to (in u2p syntax):

```
      ..... /* previous action */
      nextstate Wait;
    }
  state Wait;
    input sig(a, tmp) {
      if (tmp != 1) {
        arbiter.setVerdict(fail);
        return;
      }
      ..... /* action following the signal reception */
    }
    input * {
      arbiter.setVerdict(fail);
      return;
    }
```

### Time specification line

A time specification line is translated in different ways depending on the time specification value. All mappings share the usage of a timer to implement timing constraints.

### Time < or <= time value

A time specification with a time that should be < or <= a specified time value is translated to a timer definition and a set action on the timer at the position of the beginning of the time specification.

At the endpoint of the time specification a reset action on the timer is inserted, as the timer should still be active at that point. If the timer signal is received in any state this is an error, as the timer has expired. So for all states (state *), the receiving of the timer signal is included, leading to a transition that sets the verdict to fail and a return from the operation.

**Example 592: Time specification less than a specified value**

Timer declaration:

```
timer t = 3;
```

Time specification start:

```
set(t);
```

Time specification end:

```
reset(t);
```

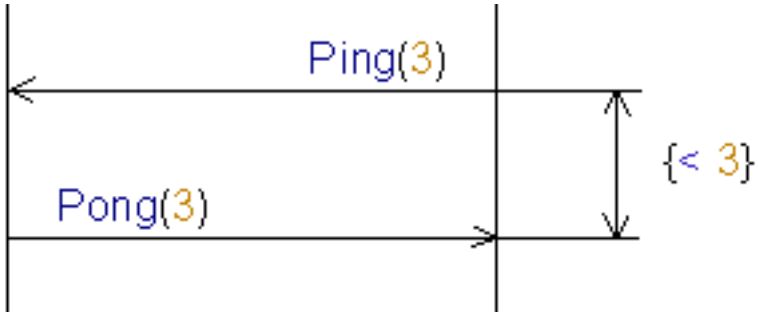



*Figure 264: A time specification with < or <= a specified time value*

Error in all states:

```
state *;
  input t {
    arbiter.setVerdict(fail);
    return;
  }
```

---

### Time > or >= time value

A time specification with a time that should be > or >= a specified time value is translated to a timer definition and a set action on the timer at the position of the beginning of the time specification.

At the endpoint of the time specification a test is made if the timer is still active. If it is, this is treated as an error (verdict set to fail, followed by return from operation). If the timer signal is received in any state this is ok and the same state is reentered.

**Example 593: Time specification greater than a specified value**———————

Timer declaration:

```
timer t = time-value;
```

Time specification start:

```
set(t);
```

Time specification end:

```
if (active(t)) {
  arbiter.setVerdict(fail);
  return;
}
```

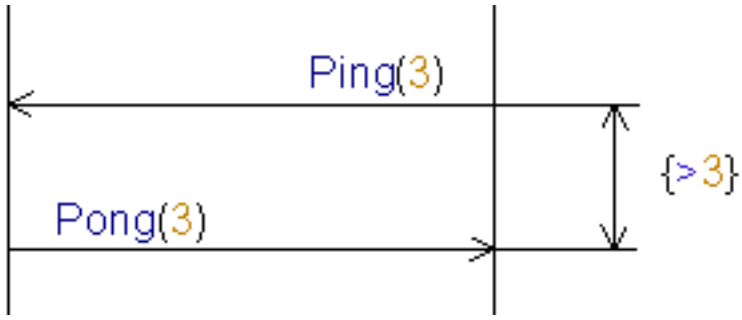



*Figure 265: A time specification with > or >= a specified time value.*

For all states:

```
state *;
  input t {
    nextstate -;
  }
```

---

**Time range**

If a time range is given in a time specification, such a time specification is translated as one time specification with time greater than time value and one with time less than time value.

## Running a test application

To run a test application, right-click the Build Artifact used to build it and select **Run Tests** from the **Build** menu. This will start execution of the test application and show the results in the built-in web browser.

The tests are executed using a generated Test input file, and the result is stored in a Test log file produced by the Test Driver during test execution.

Test applications can also be run outside of Tau in batch mode using the Test Driver executable.

### Run a test application in the Model Verifier

To run a test application in the Model Verifier, right-click the build artifact and select **Launch** from the **Build** menu. Test execution can now be controlled manually.

1. Click **Go** and wait until prompted to enter a test case in the Model Verifier Console window.
2. Enter the name of the test case you wish to run, enclosed in double-quotes, e.g. "`TestCase1`" to run `TestCase1`.

The result of the test case is written to the Model Verifier output window.

**Note**

*The Model Verifier has to be restarted to run a second test case.*

## Test execution results

When running a test application from Tau, the test execution result is written to a Test log file that is shown when execution is finished.

# Test Execution and Logging

This section explains how test are executed once a test application has been built. The architecture is depicted in Figure 266 on page 1787.

An executable called the Test Driver is responsible for invoking the test application to execute the test cases. The execution can be explained in the following steps:

1. The test driver reads the Test input file to decide which test cases to run.
2. It launches the test application which executes the test case.
3. When the Test case has been executed, the test application reports the verdict to the test driver.
4. The test driver writes the verdict to the Test log file.
5. Steps 1-4 are repeated once for each test case in the input file.

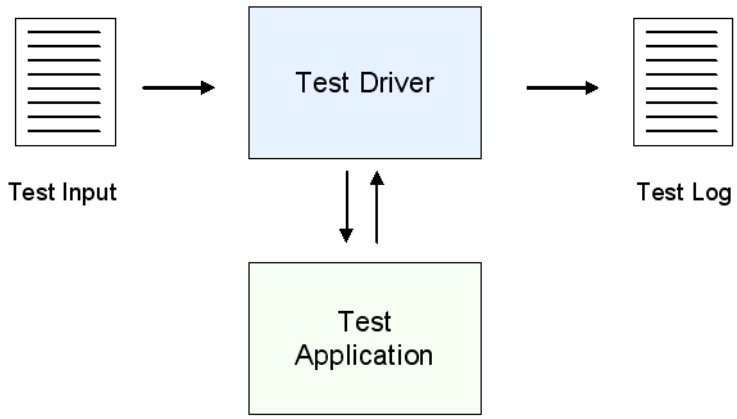



*Figure 266: Test execution architecture*

Each component in the test execution chain is described in detail in the following sections.

## Test Driver

The Test Driver is an executable for driving a test application, i.e. sequentially execute a number of test cases in a test application.

The test driver executable is called `TestDriver` and can be found in the bin folder of the installation.

```
TestDriver -type c testApplication inputFile [logFile]
```

The `-type` option is used to specify the type of the test application. Currently only c applications are supported, and c is the only applicable value.

The `testApplication` parameter specifies the full path to the generated test application.

The `inputFile` parameter specifies the full path to the Test input file that should be used during execution.

The optional `logFile` parameter specifies the full path to the Test log file that will be created by the test driver during execution. If no value is specified, the test log is written to standard output of the test driver context.

## Test input file

The test input file is a file read by the test driver to specify which test cases it shall execute, in which order and possibly a timeout value.

The test input file is a text file in XML with the following format:

```
<testinput>
  <testcase>
    <name>testcaseName        </name>
    <timeout>timeout value     </timeout>
  </testcase>
  ...
</testinput>
```

Where `testcaseName` is the name of a Test case to be executed and `timeout value` is an integer value specifying a timeout (in milliseconds) after which the execution of the test case will be aborted if the final verdict hasn't been produced. The timeout element is optional, and if it is not present, no timeout is applied.

When building a test model in Tau, a test input file is automatically generated in the Target Directory of the test application. This input file contains all test cases of the test context each with the timeout specified in the Test generation stereotype.

To remove a test case or execute one test case more than once, or to change the timeout, the input file has to be manually changed.

## Test log file

The test log contains the result of a test run. All executed test cases are listed along with their verdict and some additional information to facilitate navigation from the log file to the model. The test log is created by the test driver during test execution in the location specified in the Test generation stereotype.

The test log is written in XML format and a `XSL` style-sheet is used to present the log file in a browser. The style-sheet file, called `ttdTestLog.xsl` is copied to the same location as the test log when the log file is created. To change the style sheet, simply replace the file with another file with the same name. If the style-sheet file exists when the log is created, it will not be overwritten.

To change the appearance of the test log, replace the style-sheet.

## Test generation stereotype

The test generation stereotype is applied to a Build Artifact (in addition to a code generator specific stereotype to indicate that a test model should be built from a test context.

The test generation stereotype has the following tagged values:

- Test Log

  The full path to the log file. The default value is empty meaning that the log file will be created in the Target directory of the build artifact.
- Startup timer

  An integer value specifying a timeout value for the initialization of the test framework and test component.
- Testcase timer

  An integer value specifying a timeout value for the execution of a single test case. By default all test cases share the same timeout value. To use different values for individual test case, the Test input file has to be modified manually after code generation.
- Include Guids in input file

  Enables navigation from the test case name in the test log to the diagram in Tau that represents the corresponding test case.

- Show intermediate model

  Inserts the intermediate test model into the project and makes it visible.

The test generation stereotype can only be applied to build artifacts with a Test context as Build Root.

# Known Restrictions

This section summarizes the known restrictions of the testing profile support.

### Build artifact types

The only supported build type is Model Verifier. It is used for all C code test applications, no matter if they are run interactively or in batch using the test driver.

### Test case behavior

Only sequence diagrams and state machine diagram can be used to specify test behavior.

In addition test cases owned by a test context must be specified using sequence diagrams, and test cases owned by a test component must be specified using a state machine.

### Test cases

Test cases can not have any parameters.

Test cases can contain at most one sequence diagram (if owned by a test context)

### Test case behavior - sequence diagram

A test case sequence diagram can only contain the following diagram elements:

- Lifeline symbol
- Action symbol

  Action symbol can only be used on the test component lifeline
- Reference symbol
- Message line

- Time specification line

  Both ends of a Time specification line must be attached to the test component lifeline

  Absolute time values are not supported for Time specification line, i.e. `{== x}` or `{x}` since they are not meaningful
- Comment Symbol
- Text Symbol

### Test component

There can only be one test component per test context.

### Testing profile

The following concepts from the testing profile are not supported for test generation and execution purposes:

- Defaults and default application
- Validation action, Log action, Finish action
- `InteractionOperator`
- Test log and `Testlog` application
- Test Data: Wild cards and coding rules
- Time zones

### Timers

Timers can only be used in test cases specified using state machines.

# *UML Modeling with System Architect*

The chapters listed under this section describe how to work with UML models using Tau with System Architect.

# 63

# *Using Tau with System Architect*

This chapter describes how to work with UML models using Tau with System Architect.

System Architect is a modelling tool capable of supporting many different notations, including UML 2. One key feature of System Architect is that it is a repository based tool, that enables *encyclopedias* (the System Architect term for models) to be stored on a server and accessed over a network from either a web browser based client application or the standard System Architect client application.

The UML 2 models you create in System Architect are fully compatible with Tau. You can create UML 2 models with System Architect, store them in System Architect encyclopedias on a network server, then open the models in Tau, modify them, and save the changes back to the System Architect encyclopedia.

It is also possible to create and store models locally in Tau, move these models into System Architect encyclopedias, and then open them in System Architect. Models created in System Architect repositories can also be copied to the local file system.

The key feature in Tau that makes this possible is the possibility to associate a System Architect encyclopedia with a Tau project. When an encyclopedia is associated with a project, all UML 2 elements in the encyclopedia will be part of the Tau project as if they had been created in Tau. The way this works is that whenever a Tau project with an associated System Architect encyclo-

pedia is loaded into Tau, then System Architect will automatically be started and the corresponding encyclopedia opened. If System Architect is already running it will load the encyclopedia into the running instance of System Architect.

# Associating an Encyclopedia with a UML Model

The System Architect integration is implemented as an addin to Tau, so before the integration can be used, you must enable the addin as follows:

1. Select the command *Tools->Customize*
2. Select the *Addins* tab
3. Locate and select the *System Architect Integration* addin
4. Close the dialog

When the addin is activated an encyclopedia can be associated with a UML project using the command *Associate System Architect Encyclopedia*. This command is available in the Model View in Tau for all nodes representing the root of the UML model. If the Standard View is used, then this is the node called *Model*.

If System Architect has been manually started and has an encyclopedia loaded when the *Associate System Architect Encyclopedia* command in invoked, then this encyclopedia also be loaded into the Tau model.

If System Architect has not been manually started, then it will be started and the user will have to open a suitable encyclopedia.

If System Architect is started but has no encyclopedia open, then an error message will be given to give the user a possibility to open an encyclopedia.

It is only possible to associate one encyclopedia with a Tau project. If an encyclopedia has already been associated with a project, the menu choice will not be available.

The UML elements that are stored in encyclopedias are handled differently compared to elements stored locally on files from several different points of view. The main difference is that they are not immediately loaded into the project when the project is opened. Instead the elements are loaded as needed. See section "Incremental Loading of Elements" on page 1800 for more details.

Another difference is that there are special mechanisms to ensure that multiple users do not access the same elements or diagrams. This is described in the section "Creating, Editing and Saving UML Elements" on page 1801.

The association to an encyclopedia can be removed as described in"Removing the Association with an Encyclopedia" on page 1799

When creating a new UML project it is possible to automatically associate a System Architect encyclopedia with the project. See also "System Architect Storage and New Wizards" on page 1803.

# Removing the Association with an Encyclopedia

The command *Remove System Architect encyclopedia association* is used to remove the association to an encyclopedia. The command is available in the Model View in Tau for nodes representing the root of the UML model. If the Standard View is used, then this is the node called *Model*.

When selecting this command the association will be removed and all model elements that were stored in the encyclopedia will be removed from the model.

The model elements will however not be deleted in the encyclopedia only removed from the Tau project. So if the encyclopedia once more is associated with the project the elements will reappear.

# Incremental Loading of Elements

For performance reasons the model elements stored in an encyclopedia are not all loaded when associating an encyclopedia with a Tau project or when opening a Tau project that has an associated encyclopedia.

Instead elements are only loaded at three situations:

- Elements are loaded when the containing model element is expanded in the Model View
- Elements are loaded when a diagram is opened in an editor
- Elements are loaded when giving the command *Load all child elements*

When expanding the Model View only the elements that become visible as direct children of the expanded node will be loaded.

When opening an editor all elements that are visible in the diagram will be loaded.

The command *Load all child elements* is available on nodes in the Model View that represent elements stored in System Architect encyclopedias. The result of this command is that all child elements, both contained directly inside the selected element or indirectly by any number of intermediate elements will be loaded.

When an element has been loaded it will remain loaded even if the Model View nodes are collapsed again or the editor window is closed.

Note that due to this incremental loading scheme the model loaded in the tool usually is incomplete. The implication of this is that semantic tools like the checker, report generation tools and code generation tools may give error messages due to the fact that elements may be missing in the loaded part of the model. It is thus recommended to always use the command *Load all child elements* to load complete parts of the model before using these kinds of semantic tools.

# Creating, Editing and Saving UML Elements

Since System Architect encyclopedias can simultaneously be accessed by more than one user there is a scheme based on locking elements used whenever modifying the model in order to ensure the consistency of the encyclopedia.

The key principles that guide the updating of the encyclopedia based on changes in the UML model in the Tau client application are the following:

- All elements created in Tau will immediately be created also in the encyclopedia.
- All modifications to the model elements using the graphical editors, property pages and model view in Tau will be immediately propagated to the encyclopedia. If the element is locked by another user the modification will fail with an error message.
- All elements that are modified in Tau using any other mechanism, like for example using scripts, will be modified in the encyclopedia when the model is saved in Tau.
- All elements that are deleted in Tau will be deleted in the encyclopedia when the model is saved in Tau.
- Diagrams are locked when they are opened in Tau and unlocked when the diagram is closed. Changing the graphical properties of the symbols in diagrams will not lock or change the referenced model elements.
- When entering text edit mode for a text label in a diagram the text is first refreshed from the encyclopedia, then the referenced model elements are locked.
- When exiting text edit model for a text label in a diagram the encyclopedia is updated with the changes and the referenced model elements are unlocked.

From the point of view of someone accessing an encyclopedia there are several consequences of this:

- It may in some situations not be possible to save some of the modifications done to the model since someone else may have locked modified elements. The work around for this is to save it again later.
- Changes done to the elements in the encyclopedia for elements that are loaded in Tau are *not* propagated to Tau immediately. The consequence is that changes done in Tau may override changes done by other users.

# System Architect Storage and New Wizards

For the wizards used to create new UML projects a convenience option called “Associate with System Architect encyclopedia” is available in one of the wizard pages. If this option is selected the currently open System Architect encyclopedia will be associated with the newly created project and the elements from the encyclopedia loaded into Tau. The System Architect integration addin will also automatically be activated.

If System Architect is not started with a loaded encyclopedia, it will be started to give the user a possibility to select a suitable encyclopedia.

# Specifying Encyclopedia Storage for Root Elements

It is possible to use the command *Save in System Architect Encyclopedia* to specify that a model root element should be stored in the encyclopedia instead of in a local file.

**Note**

*Only model root elements can be specified to be stored in an encyclopedia.*

**Note**

*It is not possible to move an element from the encyclopedia to a local file using for example the Save in new file command. To move an element from the encyclopedia to a local file, use copy/paste instead.*

# Moving Information from System Architect to Tau

In many situations it is useful to move information from System Architect to Tau, like the common case is when an analysis model stored in the System Architect encyclopedia should be used as the basis of an implementation activity.

To accomplish this workflow the most convient functionality to use is the regular copy and paste functionality in Tau:

1. Select the element stored in System Architect that should be moved out from the encyclopedia.
2. Give the command “Load All Child Elements” to ensure that all elements are available in Tau
3. Copy the element
4. Paste the element at a suitable location in the model.

When planning to implement an analysis model from System Architect in Tau it is often useful to keep traceability from the implemenation model to the analysis model. This can be simplified by using the automatic traceability creation available in Tau. To accomplish this do as follows:

1. Select the element stored in System Architect that should be used as basis for the implementation.
2. Give the command “Load All Child Elements” to ensure that all elements are available in Tau.
3. Drag the element using the right mouse button from the source position to the suitable target location for the copy.
4. Choose the “Copy with Traceability” alternative in the pop-up menu

The result will be a copy where all definitions in the copy will include a <<trace>> dependency to the corresponding element in the original.

See also “Copy with Traceability” on page 138.

# Known Restrictions

## Unnamed elements

Elements in Tau are allowed to have an empty name. In System Architect encyclopedias this is not allowed. To still support unnamed elements they will in System Architect be given a name based on the metaclass and a number. So when an element that has an empty name in Tau is stored in a System Architect encyclopedia it will appear in the System Architect browser and property pages as if it had a name based on the metaclass of the element.

## Undo/Redo

Undo and Redo will not work for elements that have been stored in a System Architect encyclopedia.

## Restrictions on Model Root Elements in Encyclopedias

Only packages are supported as root elements (elements on the top level of the model hierarchy) in encyclopedias. If other root elements are created, they will not be loaded into Tau when opening a project with an associated encyclopedia. The elements will however be visible from System Architect and the recommended work around is to move the element into a package using System Architect.

## Profiles and Model Libraries

The only profiles and model libraries available in System Architect are the predefined package and the TtdPredefinedStereotypes. The consequence is that stereotypes that are used which are defined in some other profile will not be available if the model is opened in System Architect. As an example: Assume that a stereotype defined in another profile is used in a model. If this model is opened in System Architect the result will be that the stereotype instances are not shown in the diagrams.

## Accessing Diagrams from both System Architect and Tau

It is possible to open a diagram both from the System Architect user interface and the Tau user interface at the same time. Note however if the same user on the same machine opens a diagram twice, the fact that a diagram is locked does not prohibit the second access to the diagram to be succesful for read/write. The consequence is that the diagram can at the same time be modified both from the Tau and the System Architect diagram editors. When the diagram is saved there will be no merging of the diagram. The version of the diagram that is changed last will overwrite any previously saved changes to the diagram.

The recommendation is thus to only do modifications of the diagram using one user interface at a time and do frequent saves and updates if both user interfaces are used at the same time.

# *UML and Web Services*

The following chapters describe how to model web services using the Web Services Description Language (WSDL) in UML and how to import and export web service description files.

The features support WSDL version 1.1.

In addition to the features described in the below chapters, it is also possible to simulate web services using the Model Verifier. This capability is described in Web Service Simulation.

# 65

# *Web Services Support*

This chapter describes how to use the support for web services in Tau. This includes features such as modeling web services in UML and to generate WSDL from such UML models. It is also possible to import existing WSDL documents into a UML model.

# Modeling Web Services in UML

Web services can be represented in a UML model at two different levels of abstraction:

1. **WSDL centric representation.** In this kind of model WSDL constructs are represented by stereotyped UML entities. A WSDL Profile is provided which contains all stereotypes needed for representing all possible WSDL constructs.
2. **UML centric representation.** In this kind of model standard UML constructs are used, which can be translated into WSDL constructs by a WSDL generator. WSDL specific stereotypes are only used for WSDL constructs which do not have a corresponding UML construct.

Which web service representation to choose depends on the purpose of the modeling. For modelers which prefer the WSDL terminology rather than the UML terminology when modeling a web service, the WSDL centric representation is usually the best. However, if the web service not only shall be specified in the model, but also be implemented by generating code (Java, C# etc.) from the model, then the UML centric representation is more useful. The UML centric representation is usually also much more compact than the more verbose WSDL centric representation.

**Note**

*Although both the WSDL centric and the UML centric representation can be used when generating WSDL documents from a model, Tau currently only supports the WSDL centric representation when importing existing WSDL files into a model. It is, however, possible to obtain a UML centric representation of an existing web service by using the support for simulation of web services. See* Web Service Simulation *for more information.*

# Creating a WSDL Project

Regardless of whether you prefer to use a WSDL or UML centric representation of web services in your model, you should start by creating a WSDL project:

1. Choose the command File->New...
2. Create a new **UML for WSDL Modeling** project. If you intend to import existing WSDL documents for simulating web services, make sure the "Support simulation of web services" checkbox is marked. Simulation of web services is described in Web Service Simulation.

You can also activate the WSDL support for an existing project:

1. From the **Tools** menu select Customize.
2. Click the Add-Ins tab and check the **WSDL** add-in. You may also want to check the **XSDFramework** add-in if you plan to work with XSD together with WSDL.
3. Click **Close**.

## WSDL Add-in

The main features of the WSDL add-in are:

- A WSDL Profile for annotating UML models with WSDL information. This is mainly used when working with a WSDL centric web service model.
- A WSDL centric model view, the WSDL View.
- A WSDL generator which can generate WSDL documents from UML models. The UML model that is input to the code generator can either be at the abstraction of WSDL (WSDL centric representation), or it can be a standard UML model (UML centric representation). Since the WSDL and UML centric representation are quite different there are actually two different WSDL generators. They are used in slightly different ways as described in Generating WSDL.
- A WSDL importer which can generate a UML model of a web service. The resulting model will use the WSDL centric representation of the web service.

**Note**

*Activating the WSDL add-in loads the WSDL profile and view, but also the XSD profile and view since XML schemas can be embedded in WSDL specifications.*

# WSDL View

The WSDL view provides a WSDL centric view of a model. In this view only WSDL elements are shown, and only WSDL elements can be created. When using the **New...** command in the Model View context menu, the WSDL view enforces a correct WSDL structure to be created.

The WSDL view is designed to be used when working with a WSDL centric model representation of web services in Tau.

To activate the WSDL view:

1. In the **View** menu, select **Reconfigure Model View...**
2. Select **WSDL View** in the dialog and click **OK**.

You can switch between the Standard View and the WSDL View at any time.

**Note**

*Not all UML elements are visible in the WSDL View. To see all elements switch to the Standard View.*

**See also**

"Model View" on page 19 in Chapter 4, *Introduction to Tau 4.2*

"Default Model View" on page 2462 in Chapter 91, *Dialog Help*

# WSDL Profile

The WSDL profile extends UML with concepts for modeling web services as defined by WSDL. The profile contains WSDL specific stereotypes, and is mainly used when working with a WSDL centric web service model.

The WSDL profile package is called ‘wsdl’ and can be found in the **Libraries** section of the model.

More details about the contents of the WSDL profile can be found in WSDL Profile Contents.

# Generating WSDL

A Tau web service model can be translated into a WSDL file using a WSDL generator. How to perform WSDL generation depends on whether the model uses a WSDL or UML centric representation of the web service.

**Note**

*If the model contains multiple web services it is possible to use a WSDL centric representation for some, and a UML centric representation for others. However, it is not recommended to mix these representations for a single web service.*

## WSDL Generation from WSDL Centric Models

To generate WSDL from a WSDL centric model follow these steps:

1. In the context menu of a WSDL package, select **Generate WSDL...**
2. Select a folder where to place the generated WSDL file. Click **Save**.

Step 2 is only necessary the first time you generate WSDL for the package. The path to the generated WSDL file is stored in a WSDL file artifact that will be generated next to the WSDL package. Use the Properties Editor if you want to change this path later. You can also delete the WSDL file artifact and invoke the **Generate WSDL...** command again in order to generate the WSDL file to another location.

The rules for transating a WSDL centric model to WSDL files are the reverse of the translation rules described in WSDL/XSD Importer Reference. Note that UML entities must be annotated by the stereotypes described in these translation rules in order to obtain the expected WSDL.

## WSDL Generation from UML Centric Models

To generate WSDL from a UML centric model follow these steps:

1. In the Model View select an interface or a package that contains interfaces.
2. In the context menu select **WSDL Generator->New Artifact**. This will create a new WSDL build artifact manifesting the selected element.
3. In the context menu of the WSDL build artifact select **Build (WSDL Generator)->Generate**.

Step 1 and 2 is only necessary the first time you generate WSDL for the package or interface in order to obtain a WSDL build artifact. Regeneration of the WSDL file is made by performing step 3 again.

The rules for translating a UML centric model to WSDL files are described in WSDL Generator Reference.

# Importing WSDL

The WSDL/XSD Importer is used for importing WSDL definitions (and/or XSD) into a Tau model. After the import the web services described by imported WSDL files may be further developed in Tau, and then generated to WSDL files again (see WSDL Generation from WSDL Centric Models).

The initial import is done using the WSDL/XSD Import Wizard and re-import can later be perfomed by using the context menus on WSDL and XSD file artifacts.

## WSDL/XSD Import Wizard

To import WSDL definitions or an XSD schema into Tau:

1. Select a package or the “Model” node.
2. Start the **Import Wizard** by selecting **File/Import...**
3. Select **Import WSDL/XSD** and click **OK**

### First step of the WSDL/XSD import wizard

The first step of the import process is to specify the kind of XML files to import and if diagram generation should take place.

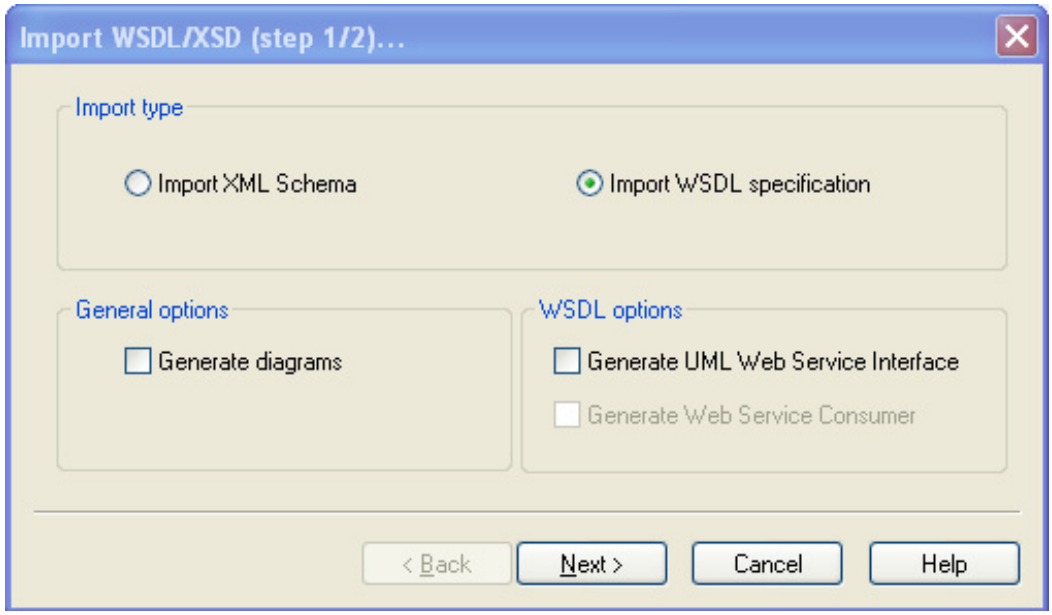



*Figure 267: The first step of the WSDL/XSD import wizard*

If you want to import XSD files you select the **Import XML Schema** option and if you would like to import WSDL files you select **Import WSDL specification** instead.

If you would like to have diagrams showing the imported elements, select the **Generate diagrams** option.

If you decide to import WSDL you can also set some importer options for generating additional information into the model. These options are used when simulating web services; see Calling a Web Service from UML for more information about these options.

To complete the first step of the wizard press **Next**.

### Second step of the WSDL/XSD import wizard

The purpose of the second step is to determine the files to be imported. The set of files is listed in the the dialog and can be changed by means of the buttons **Add Files...**, **Remove** and **Clear**.

- Pressing the **Add Files…** button invokes the standard dialog for opening files. Multiple selection is allowed. The files selected in the dialog are added to the file list.
- Pressing the **Remove** button removes the files currently selected from the list.
- Pressing the **Clear** button removes all entries from the list.

It is also possible to import from a URL rather than a file. To do this press the **Add URL** button and enter the URL where to get the WSDL/XSD file from.

**Note**

*If you access the web through a proxy server you may need to set appropriate Proxy settings to be able to import from an URL..*

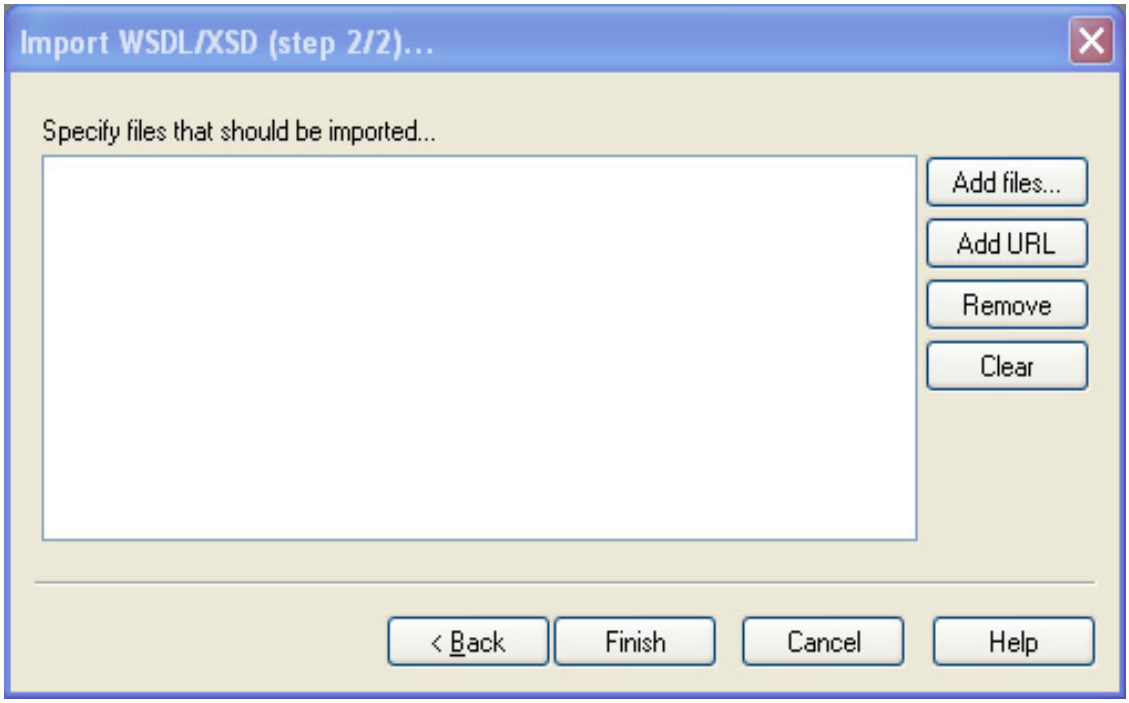



*Figure 268: The second step of the WSDL/XSD Import Wizard*

Pressing **Finish** will import the selected files and/or URLs.

### Result of import

If the **Model** node in the Model View was selected before the import, the importer will create one u2 file per imported WSDL/XSD file, add it to the project and store all imported definitions from that WSDL/XSD file into that u2 file. If a package was selected before the import, the imported definitions will be created inside that selected package and stored in the same u2 file as the package.

For each file that was added in the import wizard, there will be one file artifact generated. For each WSDL file there will be an artifact with the stereotype <<WSDL file>> and for each XSD file there will be an artifact with the stereotype <<XSD file>>. These stereotypes have a tagged value set to point to the corresponding file in the file system.

For each file there will also be one package generated. It contains the imported WSDL or XSD definitions. Each artifact has a <<manifest>> dependency to one package.

## Re-import

Once import of WSDL or XSD has taken place it is possible to re-import the content of the files. This is done by right clicking a WSDL or XSD artifact and select **Import WSDL...** or **Import XSD...** respectively. This command will remove all imported elements and redo the import.

**Note**

*Re-importing WSDL or XSD will remove all the content in the imported package, so diagrams or other UML entities that has been added after the first import will be deleted.*

# 66

# *WSDL Generator Reference*

This chapter is a reference guide to the WSDL generator which translates an unannotated UML model (UML centric web service representation) to WSDL. It describes how UML constructs are translated to WSDL constructs.

# General

The WSDL generator implements a mapping of UML constructs to WSDL constructs. The translation rules have been designed to use standard UML constructs, without stereotype annotations, to an as large extent as possible. Only when a certain WSDL construct has no corresponding UML construct it is necessary to apply stereotypes to the UML elements.

## WSDL Build Artifact

The WSDL generator is integrated with the Application Builder. To define which parts of the model to translate to WSDL, and to specify Translation Options, a WSDL build artifact is used.

The WSDL build artifact is recognized by its applied `<<WSDL Generator>>` stereotype. See WSDL Generation from UML Centric Models to learn how a WSDL build artifact can be created in the model.

WSDL is generated by performing the **Generate** command on the build artifact. Which WSDL files that are then generated depends on the kind of element that is manifested by the build artifact:

1. **An interface.** The WSDL file artifact that manifests the interface will be generated.
2. **A package.** All WSDL file artifacts manifesting interfaces contained in the package will be generated.

### Default Model-to-File Mapping

If no WSDL files are found when performing the **Generate** command on the build artifact, the WSDL generator will create WSDL file artifacts according to a default model-to-file mapping. For each manifested interface (direct manifestation, or indirect by the manifestation of a containing package) a WSDL file artifact will be created manifesting the interface. The name of the file artifact is the same as the name of the interface.

The WSDL generator also supplies an explicit command **Generate File Mapping** which can be used to generate file artifacts according to the default model-to-file mapping rules, without actually generating the WSDL files. This can be useful if additional information needs to be added to the file artifacts prior to WSDL generation, for example `xsd:import` or `xsd:include` dependencies (see Dependency).

## Document Structure

The following chapters describe the subset of UML that can be translated to WSDL by the WSDL generator. UML constructs not mentioned here are not supported, and will be ignored during translation.

For each supported UML construct a translation rule is given. If there are exceptions to the rule, these are also mentioned.

For most translation rules examples are given using textual UML and WSDL syntax.

**Note**

*The purpose of each example is only to illustrate the translation rule at hand, not to give a precise description of how the generated WSDL will look like. Also note that examples for brevity reasons typically are fragmental. Omitted sections of UML or WSDL are marked with triple dots (...).*

# Interface

**A UML interface is translated to a `<service>` element and a `<portType>` element within the WSDL file.**

The name of the WSDL service and the port type is the name of the UML interface. The same name is also used for the containing `<definitions>` element.

**Example 594: Translation of interfaces**

**UML**

```
interface MyWebService {}
```

**WSDL**

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="MyWebService" ...>
  <wsdl:portType name="MyWebService">
  </wsdl:portType>
  <wsdl:service name="MyWebService">
  </wsdl:service>
</wsdl:definitions>
```

# Comment

**A UML comment attached to an element that has a specified mapping to WSDL is translated to a `<documentation>` tag on the resulting WSDL element.**

If the UML element has multiple representations in the WSDL file (as is the case for an Interface for example) the `<documentation>` tag is only placed on one of them (for an interface, it is placed on the `<service>` element).

**Example 595: Translation of comments**

**UML**

```
interface MyWebService comment "My first webservice" {}
```

**WSDL**

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions name="MyWebService" ...>
  <wsdl:portType name="MyWebService">
  </wsdl:portType>
  <wsdl:service name="MyWebService">
    <wsdl:documentation>
My first webservice
    </wsdl:documentation>
  </wsdl:service>
</wsdl:definitions>
```

# Dependency

**An `<<xsd::import>>` or `<<xsd::include>>` dependency from a WSDL file artifact to another file artifact or package is translated to an `<import>` specification in the WSDL file.**

If a ‘namespace’ tagged value is specified in the `<<xsd::import>>` stereotype instance it is mapped to a corresponding ‘namespace’ value of the `<import>` element.

This use of dependencies provides a means to include other files (typically WSDL or XSD files) into the generated WSDL file.

**Example 596: Translation of dependencies from WSDL file artifacts**

**UML**

```
<<wsdlFile(.path = "MyInterface.wsdl".)>> artifact
'MyInterface.wsdl'
<<xsd::import(.namespace = "ns".)>> dependency to
Artifact2 {}

<<xsdFile(.path = "C:\\x.xsd".)>> artifact Artifact2 {}
```

**WSDL**

```
<wsdl:types>
  <xsd:schema ...>
    <xsd:import namespace="ns" location="C:\x.xsd"/>
  </xsd:schema>
</wsdl:types>
```

Note that since the imported file is an XSD file the generated `<xsd:import>` will be placed in the `<xsd:schema>` tag.

---

# Operation

**A UML operation contained in an interface is translated to a WSDL `<operation>` within the `<portType>` element of the WSDL document.**

The name of the WSDL operation is the name of the UML operation.

See Example 597 on page 1828 for an example.

## Parameter

**Each UML operation is also mapped to one or two WSDL `<message>` elements in the WSDL document.**

The details of this mapping depend on the kinds of parameters the operation has.

In case the UML operation has no parameters that can carry data back to the caller (that is no 'return', 'in/out' or 'out' parameters) there will just be one WSDL message generated. Otherwise there will be two WSDL messages generated. The name of the first WSDL message is the name of the UML op-

eration with the suffix “Request” appended. The name of the second WSDL message is the name of the UML operation with the suffix “Response” appended.

In each message a WSDL `<part>` declaration is made for each parameter that carries data in the direction represented by the message. That is, in the first message (“Request”) there will be one WSDL part for each 'in' and 'in/out' parameter of the UML operation. In the second message (“Response”) there will be one WSDL part for each 'in/out', 'out' or 'return' parameter of the UML operation.

The name of each WSDL part is the name of the corresponding UML parameter. If the UML parameter has no name, the WSDL part gets the name “par_<index>”, where <index> is an index to make the part name unique. For an unnamed 'return' parameter the name “result” is used instead.

If an auto generated part name (“result” or “par_<index>”) conflicts with the name of another part in the message, the index is incremented by one until a unique name is obtained (“result_<index>” or “par_<index+1>”).

Each WSDL part also gets a type (or element) reference which is the translation of the type of the UML parameter. See Type for more information.

**Example 597: Translation of operations with and without parameters**

**UML**

```
interface MyWebService {
  void Do();
  void DoParam( Boolean);
  Charstring GetOne();
  Integer GetTwo(out Charstring p1);
  Integer GetInOut(inout Charstring result);
}
```

**WSDL**

```
...
<wsdl:message name="DoRequest"/>
<wsdl:message name="DoParamRequest">
  <wsdl:part name="par_0" type="xsd:boolean"/>
</wsdl:message>
<wsdl:message name="GetOneRequest"/>
<wsdl:message name="GetOneResponse">
  <wsdl:part name="par_0" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="GetTwoRequest"/>
<wsdl:message name="GetTwoResponse">
  <wsdl:part name="p1" type="xsd:string"/>
```

```
  <wsdl:part name="par_0" type="xsd:integer"/>
</wsdl:message>
<wsdl:message name="GetInOutRequest">
  <wsdl:part name="result" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="GetInOutResponse">
  <wsdl:part name="result" type="xsd:string"/>
  <wsdl:part name="par_0" type="xsd:integer"/>
</wsdl:message>
<wsdl:portType name="MyWebService">
  <wsdl:operation name="Do">
    <wsdl:input message="tns:DoRequest"/>
  </wsdl:operation>
  <wsdl:operation name="DoParam">
    <wsdl:input message="tns:DoParamRequest"/>
  </wsdl:operation>
  <wsdl:operation name="GetOne">
    <wsdl:input message="tns:GetOneRequest"/>
    <wsdl:output message="tns:GetOneResponse"/>
  </wsdl:operation>
  <wsdl:operation name="GetTwo">
    <wsdl:input message="tns:GetTwoRequest"/>
    <wsdl:output message="tns:GetTwoResponse"/>
  </wsdl:operation>
  <wsdl:operation name="GetInOut">
    <wsdl:input message="tns:GetInOutRequest"/>
    <wsdl:output message="tns:GetInOutResponse"/>
  </wsdl:operation>
</wsdl:portType>
...
```

## Overloaded Operations

To be able to uniquely identify an operation from a set of overloaded operations (which all have the same name) the `<input>` and `<output>` elements of such a WSDL operation will contain a 'name' attribute. The value of this attribute is composed by the name of the operation followed by an index.

WSDL messages generated for overloaded operations also get the same index as suffix to make the message names unique.

**Example 598: Translation of overloaded operations**

**UML**

```
interface MyWebService {
  void foo(Integer);
  void foo(Boolean);
}
```

**WSDL**

```
<wsdl:message name="foo_1Request">
  <wsdl:part name="par_1" type="xsd:integer"/>
</wsdl:message>
<wsdl:message name="foo_2Request">
  <wsdl:part name="par_1" type="xsd:boolean"/>
</wsdl:message>
<wsdl:portType name="MyWebService">
  <wsdl:operation name="foo">
    <wsdl:input name="foo_1"
message="tns:foo_1Request"/>
  </wsdl:operation>
  <wsdl:operation name="foo">
    <wsdl:input name="foo_2"
message="tns:foo_2Request"/>
  </wsdl:operation>
</wsdl:portType>
```

# Signal

**A UML signal contained in an interface is translated to a WSDL `<operation>` within the `<portType>` element of the WSDL document.**

The name of the WSDL operation is the name of the UML signal.

When the signal only contains 'in' parameters (which is the normal case) it is thus an alternative to using an operation for modeling a one-way WSDL operation. However, it can also be used to model a WSDL request-response operation if it is given 'out' or 'inout' parameters. A signal may not have a return parameter.

The translation rules for signal parameters are the same as for operation parameters (see Parameter).

**Example 599: Translation of signals**

**UML**

```
interface MyWebService {
  signal Call(Charstring, Boolean);
}
```

**WSDL**

```
<wsdl:message name="CallRequest">
  <wsdl:part name="par_1" type="xsd:string"/>
  <wsdl:part name="par_2" type="xsd:boolean"/>
```

```
</wsdl:message>
<wsdl:portType name="MyWebService">
  <wsdl:operation name="Call">
    <wsdl:input message="tns:CallRequest"/>
  </wsdl:operation>
</wsdl:portType>
```

# Attribute

**A UML attribute contained in an interface is translated to a WSDL message with one part corresponding to the attribute.**

The name of both the message and the part is the name of the attribute. The WSDL part gets a type (or element) reference which is the translation of the type of the UML parameter. See Type for more information.

**Example 600: Translation of attributes**

**UML**

```
interface MyWebService {
  Integer sessionId;
  UserData nameAndPassword;
}
```

**WSDL**

```
<wsdl:message name="sessionId">
  <wsdl:part name="sessionId" type="xsd:integer"/>
</wsdl:message>
<wsdl:message name="nameAndPassword">
  <wsdl:part name="nameAndPassword"
element="tns:UserData"/>
</wsdl:message>
<wsdl:portType name="MyWebService"/>
```

The WSDL messages generated for interface attributes can for example be referenced in a SOAP binding using the `<soap:header>` tag (see soap-Header::part). SOAP headers are typically used for transmitting additional data with a SOAP request than what is carried by the parameters of the called operation. They are often used for data that is common to many of the operations in a web service, such as user authentication information or session token data. In UML you can set-up the value of such data by assigning values

to the interface attributes. These values will then be transmitted whenever an operation is called (provided the appropriate SOAP binding has been defined of course).

# Exception

**A UML exception specification for an interface operation is translated to a `<fault>` element for the corresponding WSDL operation.**

A WSDL message is also generated which contains one single part, named "data", whose type is the WSDL translation of the exception type (see Type).

The name of the WSDL fault is "exception_<index>", where <index> is an index to make the name unique within the operation. The name of the message is "<opName>Exception_<index>", where <opName> is the name of the operation.

**Example 601: Translation of exception specifications for operations**

**UML**

```
interface MyWebService {
  void DoSomething() throw Integer,   // Predefined type
                           ErrorInfo; // User-defined
type
}
```

**WSDL**

```
<wsdl:message name="DoSomethingRequest"/>
<wsdl:message name="DoSomethingException_0">
  <wsdl:part name="data" type="xsd:integer"/>
</wsdl:message>
<wsdl:message name="DoSomethingException_1">
  <wsdl:part name="data" element="tns:UserData"/>
</wsdl:message>
<wsdl:portType name="MyWebService">
  <wsdl:operation name="DoSomething">
    <wsdl:input message="tns:DoSomethingRequest"/>
    <wsdl:fault name="exception_0"
message="tns:DoSomethingException_0"/>
    <wsdl:fault name="exception_1"
message="tns:DoSomethingException_1"/>
  </wsdl:operation>
</wsdl:portType>
```

# Type

UML types used as the type of interface operation parameters, interface attributes and exception types, are translated to XSD types included in the WSDL document in the `<types>` section. Predefined UML types are directly mapped to built-in XSD types according to the table in XS Profile Contents.

Note that tags representing typed elements in WSDL use an attribute called "type" if the type is an XSD simpleType or complexType, while the attribute is called "element" if the type is an XSD element.

If the WSDL file artifact has an `<<xsd::import>>` dependency to an XSD file artifact or package the WSDL generator assumes that the imported XSD file contains all type definitions used in the WSDL document. It will then not generate XSD types in the `<types>` section automatically. However, if no XSD file is manually imported, a `<types>` section will be generated. It will contain all types that are used within the WSDL document (types of interface attributes, operation parameters, operation exception types etc.). It will also contain all types used by such types, recursively, until a closed type system is obtained.

See also the translator option Generate XSD File which makes it possible to tell the WSDL code generator to generate all XSD types in a separate file and import it.

# Binding Artifact

**A UML artifact stereotyped by a binding stereotype and with a dependency to an interface manifested by a WSDL file artifact, is translated to a `<binding>` element in the WSDL document. The dependency is translated to a `<port>` element in the WSDL service.**

The dependency should be stereotyped by an address stereotype specifying the location of the WSDL port described by the dependency.

The name of the WSDL binding is the name of the artifact. The name of the WSDL port is the name of the artifact with the suffix "Port" appended. The port refers to the binding which in turn refers to the portType.

**Note**

*The only binding currently supported is the SOAP binding. See Non-SOAP Bindings for more information about this limitation.*

## Default Binding

The WSDL generator supports the generation of a default binding (a SOAP binding) when generating WSDL for an interface for which no binding artifact has been specified. The generated binding artifact uses default values for all properties, except the `<soap:adress>` 'location' which must be specified explicitly. To specify the location of a web service use the Property Editor on the <<soapAddress>> dependency.

**Example 602: Generation of a default SOAP binding for a web service**

**UML**

```
interface MyWebService {}

<<soapBinding>> artifact MyWebServiceSOAPBinding
<<soapAddress>> dependency to MyWebService {}
```

**WSDL**

```
<wsdl:portType name="MyWebService">
</wsdl:portType>
<wsdl:binding name="MyWebServiceSOAPBinding"
type="tns:MyWebService">
  <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
</wsdl:binding>
<wsdl:service name="MyWebService">
  <wsdl:port name="MyWebServiceSOAPBindingPort"
binding="tns:MyWebServiceSOAPBinding">
    <soap:address/>
  </wsdl:port>
</wsdl:service>
```

To change some of the binding properties from their default values use the Property Editor on the `<<soapBinding>>` artifact.

The available properties of the `<<soapBinding>>` and `<<soapAddress>>` stereotypes are described in SOAP Binding Properties.

## Common Binding

It is possible to use a common binding for multiple web services. This is specified by letting the supplier of the dependency from the binding artifact be a package instead of an interface. All interfaces contained in that package will use the same binding.

.

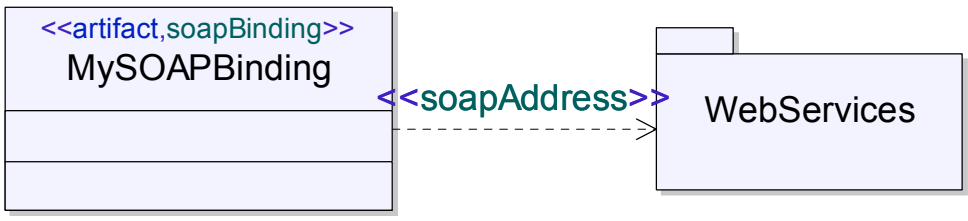



*Figure 269: Specifying a common binding for multiple web services*

### Name variable

It is possible to use a `${NAME}` variable in the 'location' tagged value of the `<<soapAddress>>`. This variable expands to the name of the web service interface. For example, `http://www.telelogic.com/${NAME}?WSDL`

Although use of this variable is mostly useful when specifying a common binding for multiple web services, it can also be used on per-interface bindings. Then the location URI does not need to be updated if the interface is renamed.

## SOAP Binding Properties

The following properties can be set to define a SOAP binding. They are defined as attribute of the `<<soapBinding>>` stereotype, and on classes used as the type of such attributes.

### soapBinding::style

This property specifies the default style for each operation of the binding. Eligible values for the property is `document` and `rpc`. The default value is `document`. The style for a particular operation can be overridden by setting the value of the soapOperation::style property.

### soapBinding::transport

This property specifies the transport which the SOAP binding corresponds to. The default value specifies the HTTP binding of the SOAP specification (`http://schemas.xmlsoap.org/soap/http`).

### soapBinding::action

This property specifies the default SOAP action for each operation of the binding. A common pattern is that this URI consists of a common part followed by the name of the operation. The WSDL generator therefore supports use of the variable `${NAME}` which expands to the name of the operation.

The action for a particular operation can be specified by setting the value of the soapOperation::action property.

### soapBinding::operationBinding

This property allows you to customize the binding for a particular operation. The value of the property is a list of `soapOperation` instances, each of which specifies an operation in the interface of the binding, and properties for the binding of that operation.

### soapOperation::operation

This property specifies one of the operations in the interface of the binding. The other properties in `soapOperation` specify binding properties for that operation.

### soapOperation::style

This property specifies the style for the binding of a particular operation. Eligible values are `document` and `rpc`. If the value is unspecified it defaults to the value of soapBinding::style. If that value also is unspecified, the default style is `document`.

### soapOperation::action

This property specifies the SOAP action for a particular operation. Its value should be a valid URI.

### soapOperation::input

This property specifies binding information related to the <input> element of the SOAP binding for a particular operation. Its value is an `soapInput` instance.

**soapOperation::output**

This property specifies binding information related to the <output> element of the SOAP binding for a particular operation. Its value is an `soapOutput` instance.

**soapOperation::fault**

This property specifies binding information related to the <fault> element of the SOAP binding for a particular operation. Its value is a `soapFault` instance.

**soapInput::body**

This property specifies how the message parts appear inside the SOAP body element of a SOAP operation input. Its value is a `soapBody` instance.

**soapInput::header**

This property allows the specification of data that is to be transmitted inside the header element of the SOAP envelope of a SOAP operation input. Its value is a `soapHeader` instance.

**soapOutput::body**

This property specifies how the message parts appear inside the SOAP body element of a SOAP operation output. Its value is a `soapBody` instance.

**soapOutput::header**

This property allows the specification of data that is to be transmitted inside the header element of the SOAP envelope of a SOAP operation output. Its value is a `soapHeader` instance.

**soapBody::parts**

The value of this property is a list of names referring to parts within a WSDL input or output message. It defines which parts that appear within the SOAP body portion of the message. The default value for this property is an empty list, which means that all parts defined by the message will be included.

### soapBody::use

This property specifies whether the parts of an input or output message are encoded or not when transmitted. Eligible values for this property are `encoded` (parts are encoded) or `literal` (parts are not encoded). It is mandatory to specify a value for this property - if it is omitted it defaults to `literal`.

### soapBody::encodingStyle

The value of this property is used if soapBody::use specifies that message parts are encoded. It specifies which rules that are used for the encoding. Its value is a list of URIs.

### soapBody::namespace

The value of this property is used if soapBody::use specifies that message parts are encoded. It should be a valid URI and will be passed as input to the specified encoding.

### soapHeader::isFault

This property specifies whether the containing `soapHeader` instance shall be translated to a `<soap:header>` or `<soap:headerfault>` element within the generated binding. The default value for this property is false.

Note that `<soap:headerfault>` elements are nested within the preceding `<soap:header>` element. If a `soapHeader` instance with `isFault` specified to true has no preceding `soapHeader` instance with `isFault` specified to false in the list, it will be ignored by the WSDL generator.

### soapHeader::part

The value for this property should be the name of an attribute of the binding interface, or the name of a parameter for the operation for which the SOAP header is specified. Its value is mapped according to the following rules:

- If an interface attribute is specified, the `<soap:header>` (or `<soap:headerfeault>`) element gets its ‘message’ attribute set to the name of the WSDL message that corresponds to the attribute. The ‘part’ attribute is set to the name of the (one and only) WSDL part of that message.

- If an operation parameter is specified, the `<soap:header>` (or `<soap:headerfeault>`) element gets its ‘message’ field set to the name of the WSDL message that contains the part that corresponds to the parameter. The ‘part’ attribute is set to the name of that part.

**soapHeader::use**

The value of this property is used in the same way as the value of soapBody::use, but for the SOAP header.

**soapHeader::encodingStyle**

The value of this property is used in the same way as the value of soapBody::encodingStyle, but for the SOAP header.

**soapHeader::namespace**

The value of this property is used in the same way as the value of soapBody::namespace, but for the SOAP header.

**soapFault::name**

This property specifies a WSDL fault for an operation. See Exception for more information about the naming of WSDL faults corresponding to exceptions in UML. If no value is specified for this property it defaults to “exception_0”, which is the name of the WSDL fault generated for the first exception specified for the UML operation.

**soapFault::use**

The value of this property is used in the same way as the value of soapBody::use, but for the SOAP fault.

**soapFault::encodingStyle**

The value of this property is used in the same way as the value of soapBody::encodingStyle, but for the SOAP fault.

**soapFault::namespace**

The value of this property is used in the same way as the value of soapBody::namespace, but for the SOAP fault.

# Limitations

WSDL documents generated by the WSDL generator complies with the WSDL 1.1 standard. Deviations and limitations from the standard are listed below.

## Transmission Primitives

Only the one-way and request-response transmission primitives are supported. The remaining two (solicit-response and notification) are not supported. They are not frequently used in many web service specifications, and the WSDL standard does not specify a binding for these transmission primitives.

## Non-SOAP Bindings

Only the SOAP binding is supported. Other bindings from the WSDL standard, such as HTTP POST/GET and MIME bindings, are not supported.

# Translation Options

Translation options are represented by means of tagged values on the WSDL build artifact. They can be set using the Properties Editor.

## Target Namespace

This is a string option specifying which target namespace to use for the generated WSDL document. The value of this option should be a URN, and it will be placed in the <definitions> element of the generated WSDL document.

If this option is unspecified the target namespace will default to the name of the build artifact.

## Generate Parameter Order

This is a boolean option. If enabled the “parameterOrder” attribute will be generated on WSDL operations. The value of this attribute is a space separated list of parameters of the operation, in the order as they are defined in UML. The ‘return’ parameter is ignored.

**Example 603: Parameter order generation**

**UML**

```
interface MyWebService {
  Boolean Op(inout Real a, Integer b, out Charstring c);
}
```

**WSDL**

```
<wsdl:message name="OpRequest">
  <wsdl:part name="a" type="xsd:double"/>
  <wsdl:part name="b" type="xsd:integer"/>
</wsdl:message>
<wsdl:message name="OpResponse">
  <wsdl:part name="a" type="xsd:double"/>
  <wsdl:part name="c" type="xsd:string"/>
  <wsdl:part name="result" type="xsd:boolean"/>
</wsdl:message>
<wsdl:portType name="MyWebService">
  <wsdl:operation name="Op" parameterOrder="a b c">
    <wsdl:input message="tns:Op_0Request"/>
    <wsdl:output message="tns:Op_0Response"/>
  </wsdl:operation>
```

```
</wsdl:portType>
```

## Generate XSD File

This is a boolean option which controls how the WSDL generator should handle types used in the WSDL definition, for the case when the WSDL file artifact does not have an `<<xsd::import>>` dependency to an XSD file artifact or package. If such a dependency exists the WSDL generator assumes the user has manually specified the XSD types in the imported file, and the value of this option is then ignored (see Dependency).

Without `<<xsd::import>>` dependencies present the WSDL code generator will translate all types that are used within the WSDL document (types of interface attributes, operation parameters, operation exception types etc.). This includes indirectly used types. If this option is false the result of this translation is included in the WSDL document in the `<types>` section. However, if the option is true the WSDL generator will generate the types into an XSD file placed next to the generated WSDL file. An `<import>` element is then added to import the generated XSD file into the WSDL document.

# 67

# *WSDL/XSD Importer Reference*

This chapter is a reference guide to the WSDL importer which translates WSDL files to an annotated UML model (WSDL centric web service representation). It also covers the translation of XSD files to UML.

# WSDL to UML Mapping Rules

To understand this document fully the following information sources are needed:

- The XSD and XS Profiles
- The WSDL Profile
- The SOAP Profile
- The SOAPENC Profile
- The XSD to UML Mapping Rules
- The XML namespace mapping rules

## WSDL Profile Contents

The WSDL profile contains a set of stereotypes used to annotate a UML model with WSDL information. When importing a WSDL file, the resulting UML model is always explicitly annotated.

The stereotypes of the WSDL profile are listed in the table below.

| Stereotype | Extends | Description |
|---|---|---|
| `<<documentation>>` | `Comment` | Represents `wsdl:documentation` |
| `<<message>>` | `Class` | Represents `wsdl:message` |
| `<<part>>` | `Attribute` | Represents `wsdl:part` |
| `<<portType>>` | `Interface` | Represents `wsdl:portType` |
| `<<input>>` | `Parameter` | Represents `wsdl:input` |
| `<<output>>` | `Parameter` | Represents `wsdl:output` |
| `<<fault>>` | `Parameter` | Represents `wsdl:fault` |
| `<<binding>>` | `Interface` | Represents `wsdl:binding` |
| `<<service>>` | `Class` | Represents `wsdl:service` |
| `<<wsdlPackage>>` | `Package` | Represents `wsdl:definitions` |

| Stereotype | Extends | Description |
|---|---|---|
| <<wsdlImport>> | Dependency | Represents wsdl:import |
| <<extensibilityElement>> | Entity | A base stereotype for all WSDL extensibiliy elements.<br>Any stereotypes specifying extensions, for example SOAP encoding, should be derived from this stereotype. |
| <<wsdlDiagram>> | ClassDiagram | |
| <<elementReference>> | Dependency | see mapping of wsdl:part for details |
| <<parameterOrder>> | Operation | see mapping of wsdl:operation for details |
| | | |

# Mapping Rules

This chapter contains the actual mapping rules. It is organized by the different WSDL elements.

### Overview

| WSDL | UML |
|---|---|
| <definition> | Package with stereotype <<wsdlPackage>> |
| <import> | Dependency with stereotype <<wsdlImport>> |
| <documentation> | Comment with stereotype <<documentation>> |
| <type> | Package with stereotype <<schema>>. |
| <message> | Class with stereotype <<message>> |
| <part> | Attribute with stereotype <<part>> |
| <portType> | Interface with stereotype <<portType>> |
| <operation> | Operation with stereotype <<operation>> |

| WSDL | UML |
|---|---|
| <input> | Parameter with stereotype <<input>> |
| <output> | Parameter with stereotype <<output>> |
| <fault> | Parameter with stereotype <<fault>> |
| <binding> | Interface with stereotype <<binding>> |
| <service> | Class with stereotype <<service>> |
| <port> | Port with stereotype <<port>> |
| | |

Each WSDL element may have XML namespace declarations. These declaration are mapped to the stereotype <<xmlNamespace>> according to rules described in the chapter [XML Namepsace Mapping Rules].

### extensibility elements

WSDL elements can have extensibility elements representing a specific technology (e.g. SOAP, HTTP, etc). WSDL Importer keeps these elements either in specific stereotypes (for SOAP) or in the special stereotype <<extensibilityElement>>.

The stereotype instance representing extensibility element is owned by UML element which represents WSDL element owning this extensibility element.

Possible locations of the extensibility elements are defined in the WSDL specification[http://www.w3.org/TR/wsdl#A3].

### definitions

```
<wsdl:definitions name="nmtoken"?
targetNamespace="uri"?>
```

This is a root level of WSDL specification. It is mapped to a Package with stereotype <<wsdlPackage>>.

The attribute name is mapped to the Name of the Package. If the attribute name is omitted, then imported Package will have the same name as value of the targetNamespace attribute.

The attribute targetNamespace is mapped to the targetNamespace tagged value of the stereotype <<wsdlPackage>>.

**Example 604**

WSDL:

```
<wsdl:definitions
targetNamespace="http://interpressfact.net/webservices/"
>
```

UML:

```
<<wsdlPackage(. targetNamespace =
http://interpressfact.net/webservices/ .)>>
package 'http://interpressfact.net/webservices/'
{
...
}
```

### import

```
<import namespace="uri" location="uri"/>
```

The `import` element is mapped to a Dependency with stereotype `<<wsdlImport>>`.

The attribute `namespace` is mapped to the `namespace` tagged value of the `<<wsdlImport>>` stereotype.

The attribute `location` is mapped to the `location` tagged value of the `<<wsdlImport>>` stereotype.

WSDL Importer attempts to automatically resolve location of the referenced document and import it. If this operation is successfull, the `Supplier` of the dependency will be set to the package that corresponds to the imported document. If WSDL Importer cannot automatically resolve location of the referenced document, the the `Supplier` of the dependency will be empty.

**Example 605**

WSDL:

UML:

### documentation

```
<wsdl:documentation .... /> ?
```

Documentation is mapped to a Comment with <<documentation>> stereotype owned by a UML representation of the enclosing element.

**Example 606**

WSDL:

```
<documentation>This is a comment</documentation>
```

UML:

```
<<wsdl::documenation>> comment "This is a comment"
```

### types

An XSD schema defined in a WSDL file is mapped to a schema package withing WSDL package. See the XSD to UML Mapping chapter for details.

### message

```
<wsdl:message name="nmtoken">
```

The message element is mapped to a Class with stereotype <<message>>.

The attribute name is mapped to the Name of the Class.

**Example 607**

WSDL:

```
<wsdl:message name="getJokeSoapIn">
...
</wsdl:message>
```

UML:

```
<<message>> class getJokeSoapIn {
...
}
```

### part

```
<part name="nmtoken" element="qname"? type="qname"?/>
```

The part element is mapped to an Attribute with stereotype <<part>>.

The attribute name is mapped to the Name of the Attribute.

The attribute `type` is mapped to the `Type` of the Attribute.

The attribute `element` is mapped to a Dependency with stereotype `<<elementReference>>` owned by the Attribute. The `Supplier` of the Dependency will be set to UML representation of the XSD `<element>`.

**Example 608**

WSDL:

```
<wsdl:part name="City" type="s:string"/>
<wsdl:part name="parameters" element="tns:getJoke"/>
```

UML:

```
<<'part'>> xs::string City;
<<'part'>> '' parameters <<elementReference>> dependency
to 'http://interpressfact.net/webservices/'::getJoke;
```

### portType

```
<wsdl:portType name="nmtoken">
```

The `portType` element is mapped to an Interface with stereotype `<<portType>>`.

The `Name` of the Interface is set to `<name>PortType` where the `<name>` is a value of the attribute `name`. The original name is kept in the stereotype `<<originalName>>`.

**Example 609**

WSDL:

```
<wsdl:portType name="getJokeSoap">
....
</wsdl:portType>
```

UML:

```
<<portType, originalName(.name = "getJokeSoap".)>>
interface getJokeSoapPortType {
...
}
```

### operation

```
<wsdl:operation name="nmtoken"
```

```
parameterOrder="nmtokens"?>
```

The `operation` element is mapped to an Operation.

The attribute `name` is mapped to the `Name` of the Operation.

The attribute `parameterOrder` is mapped to the stereotype `<<parameterOrder>>`. The value of this attribute is mapped to the value of the attribute `order` of the stereotype `<<parameterOrder>>`.

**Example 610**

WSDL:

```
<wsdl:operation name="getJoke">
...
</wsdl:operation>
```

UML:

```
void getJoke(...) {
...
}
```

### input

```
<wsdl:input name="nmtoken"? message="qname">
```

The `input` element is mapped to a Parameter with the stereotype `<<input>>`.

The `direction` of the Parameter is set to `in`.

The attribute `name` is mapped to the `Name` of the Parameter.

If the attribute `name` was omitted, the Name of the Parameter is set to `'<wsdl:input>'` and the stereotype `<<originalName>>` with empty name is applied to the Parameter.

The attribute `message` is mapped to the `Type` of the Parameter.

**Example 611**

WSDL:

```
<wsdl:input message="getJokeSoapIn"/>
```

UML:

```
void getJoke(
<<'input', originalName(.name = "".)>> in getJokeSoapIn
'<wsdl:input>'
....
)
```

**output**

```
<wsdl:output name="nmtoken"? message="qname">
```

The `output` element is mapped to a Parameter with the stereotype `<<output>>`.

The `direction` of the Parameter is set to `out`.

The attribute `name` is mapped to the `Name` of the Parameter.

If the attribute `name` was omitted, the `Name` of the Parameter is set to `'<wsdl:output>'` and the stereotype `<<originalName>>` with empty name is applied to the Parameter.

The attribute `message` is mapped to the `Type` of the Parameter.

**Example 612**

WSDL:

```
<wsdl:output message="getJokeSoapOut"/>
```

UML:

```
void getJoke(
<<'output', originalName(.name = "".)>>
outgetJokeSoapOut '<wsdl:output>'
....
)
```

**fault**

```
<wsdl:fault name="nmtoken" message="qname">
```

The `output` element is mapped to a Parameter with the stereotype `<<output>>`.

The `direction` of the Parameter is set to `out`.

The attribute `name` is mapped to the `Name` of the Parameter.

The attribute `message` is mapped to the `Type` of the Parameter.

**Example 613**

WSDL:

```
<wsdl:fault name="getJokeFault"
message="getJokeSoapFault"/>
```

UML:

```
void getJoke(
<<'fault', originalName(.name = "".)>> out
getJokeSoapFault getJokeFault
....

)
```

### binding

```
<wsdl:binding name="nmtoken" type="qname"> *
    <-- extensibility element (1) --> *
    <wsdl:operation name="nmtoken"> *
       <-- extensibility element (2) --> *
       <wsdl:input name="nmtoken"? > ?
           <-- extensibility element (3) -->
       </wsdl:input>
       <wsdl:output name="nmtoken"? > ?
           <-- extensibility element (4) --> *
       </wsdl:output>
       <wsdl:fault name="nmtoken"> *
           <-- extensibility element (5) --> *
       </wsdl:fault>
    </wsdl:operation>
</wsdl:binding>
```

The `binding` element is mapped to an Interface with the stereotype `<<binding>>`.

The `Name` of the Interface is set to `<binding name>Binding` where the `<binding name>` is a value of the attribute `name`. The original name of the binding is saved in the `<<originalName>>` stereotype.

The attriibute `type` is mapped to a `Generalization`. The `Parent` of the `Generalization` is set to the `Interface` which corresponds to UML representation of the `type`.

The `operation` element in the binding is mapped to an `Operation`. The attribute `name` is mapped to the `Name` of the `Operation`.

The `input` element in the `operation` element is mapped to a `Parameter` with the same properties as corresponding parameter from the `portType` operation.

The `output` element in the `operation` element is mapped to a `Parameter` with the same properties as corresponding parameter from the `portType` operation.

The `fault` element in the `operation` element is mapped to a `Parameter` with the same properties as corresponding parameter from the `portType` operation.

**Example 614**

WSDL:

```
<wsdl:binding name="getJokeSoap" type="tns:getJokeSoap">
...
</wsdl:binding>
```

UML:

```
<<binding, originalName(.name = "getJokeSoap".)>>
interface getJokeSoapBinding : getJokeSoapPortType{
....
}
```

**service**

```
<wsdl:service name="nmtoken">
```

The `service` element is mapped to a `Class` with the stereotype `<<service>>`.

The attribute `name` is mapped to the `Name` of the `Class`.

**Example 615**

WSDL:

```
<wsdl:service name="getJoke">
...
</wsdl:service>
```

UML:

```
<<service>> class getJoke {
...
```

```
}
```

**port**

```
<wsdl:port name="nmtoken" binding="qname">
```

The `port` element is mapped to a `Port`.

The attribute `name` is mapped to the `Name` of the `Port`.

The attribute `binding` is mapped to the `Realized` attribute of the `Port`.

**Example 616**

WSDL:

```
<wsdl:port name="getJokeSoap" binding="getJokeSoap">
...
</wsdl:port>
```

UML:

```
port getJokeSoap in with getJokeSoap;
```

# SOAP 1.1 Mapping Rules

WSDL includes a binding for SOAP 1.1 as part of the specification. This chapter describes mapping rules for SOAP 1.1 extensibility elements.

## SOAP Profile Overview

The SOAP profile contains a set of types and stereotypes used to represent SOAP 1.1 elements when importing WSDL document.

Stereotypes of the SOAP profile:

| Stereotype | Extends | Description |
|---|---|---|
| `<<binding>>` | `Interface` | Represents `soap:binding` |
| `<<operation>>` | `Operation` | Represents `soap:operation` |
| `<<body>>` | `Parameter` | Represents `soap:body` |

| Stereotype | Extends | Description |
|---|---|---|
| <<header>> | Parameter | Represents soap:header |
| <<fault>> | Parameter | Represents soap:fault |
| <<address>> | Port | Represents soap:address |

Types of the SOAP profile:

| Type | Description |
|---|---|
| enum UseKind | Specify encoding rules for parts, headers, etc |
| enum StyleKind | Specify operation style |
| | |
| | |
| | |

**soap:address**

```
<wsdl:port>
<soap:address location="uri"/>
</wsdl:port>
```

The soap:address element extends the wsdl:port. It is mapped to the stereotype <<soap::address>>.

The attributes of the soap:address are mapped to the attributes of the stereotype according to the table:

| address attribute | stereotype attribute |
|---|---|
| location | location : xs::anyURI |
| | |

**soap:binding**

```
<wsdl:binding>
<soap:binding style="rpc|document" transport="uri">
...
```

```
</wsdl:binding>
```

The `soap:binding` element extends the `wsdl:binding`. It is mapped to the stereotype `<<soap::binding>>`.

The attributes of the `soap:binding` are mapped to the attributes of the stereotype according to the table:

| binding attribute | stereotype attribute |
|---|---|
| `style` | `style : StyleKind` |
| `transport` | `transport : xs::anyURI` |
| | |

**soap:operation**

```
<wsdl:binding>
    <wsdl:operation>
       <soap:operation soapAction="uri"?
style="rpc|document"?>?
    </wsdl:operation>
</wsdl:binding>
```

The `soap:operation` element extends the `wsdl:operation` element. It is mapped to the stereotype `<<soap::operation>>`.

The attributes of the `soap:operation` are mapped to the attributes of the stereotype according to the table:

| operation attribute | stereotype attribute |
|---|---|
| `soapAction` | `soapAction : xs::anyUri [0..1]` |
| `style` | `style : StyleKind [0..1]` |
| | |

**soap:body**

```
<wsdl:binding>
        <wsdl:operation>
           <wsdl:input>
               <soap:body parts="nmtokens"?
```

```
use="literal|encoded"?
                              encodingStyle="uri-list"?
namespace="uri"?>
           </wsdl:input>
           <wsdl:output>
               <soap:body parts="nmtokens"?
use="literal|encoded"?
                              encodingStyle="uri-list"?
namespace="uri"?>
           </wsdl:output>
        </wsdl:operation>
</wsdl:binding>
```

The `soap:body` element extends the `wsdl:input` or `wsdl:output` elements. It is mapped to the stereotype `<<soap::body>>`.

The attributes of the `soap:body` are mapped to the attributes of the stereotype according to the table:

| body attribute | stereotype attribute |
|---|---|
| parts | parts : xs::NMTOKEN [*] |
| use | use : UseKind |
| encodingStyle | encodingStyle : xs::anyUri[*] |
| namespace | namespace : xs::anyUri [0..1] |

**soap:fault**

```
<wsdl:binding>
        <wsdl:operation>
          <wsdl:fault>*
            <soap:fault name="nmtoken"
use="literal|encoded"
                                  encodingStyle="uri-list"?
namespace="uri"?>
            </wsdl:fault>
        </wsdl:operation>
</wsdl:binding>
```

The `soap:fault` element extends the `wsdl:fault` element. It is mapped to the stereotype `<<soap::fault>>`.

The attributes of the `soap:fault` are mapped to the attributes of the stereotype according to the table:

| fault attribute | stereotype attribute |
|---|---|
| name | name : xs::NMTOKEN |
| use | use : UseKind |
| encodingStyle | encodingStyle : xs::anyUri[*] |
| namespace | namespace : xs::anyUri [0..1] |

### soap:header

```
<wsdl:binding>
        <wsdl:operation>
           <wsdl:input>
            <soap:header message="qname" part="nmtoken"
use="literal|encoded"
                            encodingStyle="uri-list"?
namespace="uri"?>*
              <soap:headerfault message="qname"
part="nmtoken" use="literal|encoded"
                                    encodingStyle="uri-
list"? namespace="uri"?/>*
             </soap:header>
           </wsdl:input>
           <wsdl:output>
              <soap:header message="qname"
part="nmtoken" use="literal|encoded"
                           encodingStyle="uri-list"?
namespace="uri"?>*
                <soap:headerfault message="qname"
part="nmtoken" use="literal|encoded"
                                     encodingStyle="uri-
list"? namespace="uri"?/>*
              </soap:header>
           </wsdl:output>
</wsdl:operation>
```

The `soap:header` element extends `wsdl:input` or `wsdl:output` elements. It is mapped to the stereotype `<<soap::header>>`.

The attributes of the `soap:header` are mapped to the attributes of the stereotype according to the table:

| header attribute | stereotype attribute |
|---|---|
| message | message : xs::QName |
| part | part : xs:NMTOKENS |
| use | use : UseKind |
| encodingStyle | encodingStyle : xs::anyURI [*] |
| namespace | namespace : xs::anyURI [0..1] |
| | |
| | |

The soap:headerfault element is mapped to the instance of the class soap::headerfault. This instance is set as value of the faults attribute of the stereotype <<soap::header>>. The attributes of the soap:headerfault are mapped to the attributes of the class soap::headerfault according to the table:

| headerfault attribute | class attribute |
|---|---|
| namespace | namespace : xs::anyURI [0..1] |
| use | use : UseKind |
| part | part : xs::NMTOKEN |
| encodingStyle | encodingStyle : xs:anyURI [*] |
| name | name : xs:QName |

# XSD to UML Mapping Rules

## XSD Profile Contents

The XSD profile contains a set of stereotypes and helper types used to annotate a UML model with XML Schema information. When importing a XSD file, the resulting UML model shall always be explicitly annotated.

For each XML Schema entity there is the stereotype in the XSD Profile with the same name.

| Stereotype | Description |
|---|---|
| `<<any>>` | Represents `xsd:any` |
| `<<key>>` | Represents `xsd:key` |
| `<<unique>>` | Represents `xsd:unique` |
| `<<selector>>` | Represents `xsd:selector` |
| `<<keyref>>` | Represents `xsd:keyref` |
| `<<field>>` | Represents `xsd:field` |
| `<<notation>>` | Represents `xsd:notation` |
| `<<appInfo>>` | Represents `xsd:appInfo` |
| `<<documentation>>` | Represents `xsd:documentation` |
| `<<simpleType>>` | Represents `xsd:simpleType` |
| `<<restriction>>` | Represents `xsd:restriction` |
| `<<list>>` | Represents `xsd:list` |
| `<<union>>` | Represents `xsd:union` |
| `<<include>>` | Represents `xsd:include` |
| `<<import>>` | Represents `xsd:import` |
| `<<simpleContent>>` | Represents `xsd:simpleContent` |

| Stereotype | Description |
|---|---|
| <<complexContent>> | Represents xsd:complexContent |
| <<all>> | Represents xsd:all |
| <<choice>> | Represents xsd:choice |
| <<sequence>> | Represents xsd:sequence |
| <<extension>> | Represents xsd:extension |

## XS Profile Contents

The XS profile contains a set of types representing XML Schema Datatypes.

| XSD type | UML type |
|---|---|
| anyType | datatype anyType |
| anySimpleType | datatype anySimpleType : anyType |
| duration | datatype duration : anySimpleType<br>implicit conversions and assignment operators:<br>• from the Predefined::Charstring |
| dataTime | datatype dateTime : anySimpleType<br>implicit conversions and assignment operators:<br>• from the Predefined::Charstring |
| time | datatype time : anySimpleType<br>implicit conversions and assignment operators:<br>• from the Predefined::Charstring |
| date | datatype date : anySimpleType<br>implicit conversions and assignment operators:<br>• from the Predefined::Charstring |
| gYearMonth | datatype gYearMonth : anySimpleType<br>implicit conversions and assignment operators:<br>• from the Predefined::Charstring |

| XSD type | UML type |
|---|---|
| `gYear` | `datatype gYear : anySimpleType`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Charstring` |
| `gMonthDay` | `datatype gMonthDay : anySimpleType`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Charstring` |
| `gDay` | `datatype gDay : anySimpleType`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Charstring` |
| `gMonth` | `datatype gMonth : anySimpleType`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Charstring` |
| `boolean` | `datatype boolean : anySimpleType`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Boolean` |
| `base64Binary` | `datatype base64Binary : anySimpleType`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Integer` |
| `hexBinary` | `datatype hexBinary : anySimpleType`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Integer`<br>• from the `Predefined::Charstring` |
| `float` | `datatype float : anySimpleType`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Real` |
| `double` | `datatype double : anySimpleType`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Real` |

| XSD type | UML type |
|---|---|
| anyURI | datatype anyURI : anySimpleType<br>implicit conversions and assignment operators:<br>• from the Predefined::Charstring |
| QName | datatype QName : anySimpleType<br>implicit conversions and assignment operators:<br>• from the Predefined::Charstring |
| NOTATION | datatype NOTATION : anySimpleType<br>implicit conversions and assignment operators:<br>• from the Predefined::Charstring |
| string | datatype string : anySimpleType<br>implicit conversions and assignment operators:<br>• from the Predefined::Charstring |
| normalizedString | datatype normalizedString: string<br>implicit conversions and assignment operators:<br>• from the Predefined::Charstring |
| decimal | datatype decimal : anySimpleType<br>implicit conversions and assignment operators:<br>• from the Predefined::Real |
| integer | datatype integer : decimal<br>implicit conversions and assignment operators:<br>• from the Predefined::Integer |
| token | datatype token : normalizedString<br>implicit conversions and assignment operators:<br>• from the Predefined::Charstring |
| nonPositiveInteger | datatype nonPositiveInteger : integer<br>implicit conversions and assignment operators:<br>• from the Predefined::Integer |
| long | datatype long : integer<br>implicit conversions and assignment operators:<br>• from the Predefined::Integer |

| XSD type | UML type |
|---|---|
| nonNegativeInteger | `datatype nonNegativeInteger : integer`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Integer` |
| language | `datatype language : token`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Charstring` |
| Name | `datatype Name : token`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Charstring` |
| NMTOKEN | `datatype NMTOKEN : token`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Charstring` |
| negativeInteger | `datatype negativeInteger : nonPositiveInteger`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Integer` |
| int | `datatype int : long`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Integer` |
| unsignedLong | `datatype unsignedLong : nonNegativeInteger`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Integer` |
| positiveInteger | `datatype positiveInteger : nonNegativeInteger`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Integer` |
| NCName | `datatype NCName : Name`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Charstring` |
| NMTOKENS | `datatype NMTOKENS` |

| XSD type | UML type |
|---|---|
| `short` | `datatype short : int`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Integer` |
| `unsignedInt` | `datatype unsignedInt : unsignedLong`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Integer` |
| `ID` | `datatype ID : NCName`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Charstring` |
| `IDREF` | `datatype IDREF : NCName`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Charstring` |
| `ENTITY` | `datatype ENTITY : NCName`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Charstring` |
| `byte` | `datatype byte : short`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Integer` |
| `unsignedShort` | `datatype unsignedShort : unsignedInt`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Integer` |
| `IDREFS` | `datatype IDREFS` |
| `ENTITIES` | `datatype ENTITIES` |
| `unsignedByte` | `datatype unsignedByte : unsignedShort`<br>implicit conversions and assignment operators:<br>• from the `Predefined::Integer` |

# SOAPENC Profile Overview

The SOAPENC profile contains a set of types representing SOAP 1.1-specific encoding information. Most of these types extends types from the XS profile.

# Mapping rules

### Overview

| XSD | UML |
|---|---|
| <xsd:attribute> | attribute with the stereotype <<attirbute>> |
| <xsd:attributeGroup> | attribute or class with the stereotype <<attirbuteGroup>> |
| <xsd:complexType> | class with the stereotype <<complexType>> |
| <xsd:complexContent> | class with the stereotype <<complexContent>> |
| <xsd:documentation> | comment with the stereotype <<documentation>> |
| <xsd:appinfo> | comment with the stereotype <<appinfo>> |
| <xsd:element> | attribute with the stereotype <<element>> |
| <xsd:schema> | package with the stereotype <<schema>> |
| <xsd:simpleContent> | class with the stereotype <<simpleContent>> |
| <xsd:sequence> | class with the stereotype <<sequence>> |
| <xsd:key> | informal contstraint with the stereotype <<key>> |
| <xsd:keyref> | informal constraint with the stereotype <<keyref>> |
| <xsd:unique> | informal constraint with the stereotype <<unique>> |
| <xsd:selector> | informal constraint with the stereotype <<selector>> |

| XSD | UML |
|---|---|
| <xsd:field> | informal constraint with the stereotype <<field>> |
| <xsd:group> | attribute or class with the stereotype <<group>> |
| <xsd:all> | class with the stereotype <<all>> |
| <xsd:any> | attribute with the stereotype <<any>> |
| <xsd:anyAttribute> | attribute with the stereotype <<anyAttribute>> |
| <xsd:choice> | class with the stereotype <<choice>> |
| <xsd:import> | dependency with the stereotype <<import>> |
| <xsd:include> | dependency with the stereotype <<include>> |
| <xsd:redefine> | package with the stereotype <<redefine>> |
| <xsd:simpleType> | data type with the stereotype <<simpleType>> |
| <xsd:list> | data type with the stereotype <<list>> |
| <xsd:union> | data type with the stereotype <<union>> |
| <xsd:annotation> | stereotype <<annotation>> or artifact with the stereotype <<annotation>> |
| <xsd:extension> | generalization with the stereotype <<extension>> |
| <xsd:restriction> | generalization with the stereotype <<restriction>> |
| <xsd:notation> | artifact with the stereotype <notation> |
| | |

### Importing non-schema elements

When importing XSD document all non-schema elements are ignored.

### Importing non-schema attributes

A non-scheme attribute on a XSD entity is mapped to the stereotype <<xmlAttribute>> from the TTDXmlFramework profile.

The name of the attribute is mapped to the attribute name of the stereotype <<xmlAttribute>>.

The value of the attribute is mapped to the attribute value of the stereotype <<xmlAttribute>>.

### ID attribute

The id attribute on XSD entities is mapped to the stereotype <<xmlAttribute>>.

### attribute declaration

```
<xsd:attribute
  default = string
  fixed = string
  form = (qualified | unqualified)
  id = ID
  name = NCName
  ref = QName
  type = QName
  use = (optional | prohibited | required) : optional
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, simpleType?)
</xsd:attribute>
```

The xsd:attribute element is mapped to an Attribute with the stereotype <<attribute>>.

The name of the element xsd:attribute is mapped to the Name of the Attribute.

The type of the element xsd:attribute is mapped to the Type of the Attribute.

The ref of the element xsd:attribute is mapped to a Dependency with the stereotype <<ref>>. The value of the ref is mapped to the Supplier of the Dependency.

The Type of the Attribute is set to Predefined::ExternalObject if the ref is present, but the type is not specified.

The other attributes of the xsd:attribute are mapped to the attributes of the stereotype <<attribute>> according to the table:

| XSD | UML |
|---|---|
| default | default : Charstring [0..1] |
| form | form : FormKind [0..1] |
| use | use : UseKind [0..1] |
| fixed | fixed : Charstring [0..1] |
| | |

If xsd:attribute contained a xsd:simpleType, the UML element corresponded to the xsd:simpleType is inserted in the InlineType of the Attribute.

### element declaration

```
<xsd:element
  abstract = boolean : false
  block = (#all | List of (extension | restriction |
substitution))
  default = string
  final = (#all | List of (extension | restriction))
  fixed = string
  form = (qualified | unqualified)
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded)  : 1
  minOccurs = nonNegativeInteger : 1
  name = NCName
  nillable = boolean : false
  ref = QName
  substitutionGroup = QName
  type = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, ((simpleType | complexType)?,
(unique | key | keyref)*))
</xsd:element>
```

The xsd:element element is mapped to an Attribute with the stereotype <<element>>.

The name of the element xsd:element is mapped to the Name of the Attribute.

The type of the element xsd:element is mapped to the Type of the Attribute.

The `ref` attribute of the element `xsd:element` is mapped to a `Dependency` with the stereotype `<<ref>>`. The value of the `ref` is mapped to the `Supplier` of the `Dependency`. If the `type` attribute is not present, the `Type` of the `Attribute` is set to `Predefined::ExternalObject`.

The `abstract` attribute of the element `xsd:element` is mapped to the `Abstract` property of the `Attribute`.

The other attributes of the element `xsd:element` are mapped to the attributes of the stereotype `<<element>>` according to the table:

| XSD | UML |
|---|---|
| block | block : BlockKind [*] |
| default | default : xs::string [0..1] |
| final | final : FinalKind [*] |
| fixed | fixed : xs:string [0..1] |
| form | form : FormKind [0..1] |
| maxOccurs | maxOccurs : MaxOccurence |
| minOccurs | minOccurs : xs:nonNegativeInteger [0..1] |
| substitutionGroup | substitutionGroup : xs::QName [0..1] |
| nillable | nillable : xs::boolean [0..1] |

If the `Content` of the `xsd:element` is a `simpleType` or a `complexType`, the corresponding UML element is inserted in the `InlineType` of the `Attribute`.

If the `Content` of the `xsd:element` is a `unique, key or keyref`, the corresponding UML element is inserted in the `Constraints` of the `Attribute`.

**complex type**

```
<xsd:complexType
  abstract = boolean : false
  block = (#all | List of (extension | restriction))
  final = (#all | List of (extension | restriction))
  id = ID
  mixed = boolean : false
  name = NCName
  {any attributes with non-schema namespace . . .}>
```

```
  Content: (annotation?, (simpleContent | complexContent
| ((group | all | choice | sequence)?, ((attribute |
attributeGroup)*, anyAttribute?))))
</xsd:complexType>
```

The `xsd:complexType` element is mapped to a `Class` with the stereotype `<<complexType>>`.

The attribute `name` is mapped to the `Name` of the `Class`.

The other attributes of the element `xsd:complexType` are mapped to the attributes of the stereotype `<<complexType>>` according to the table:

| XSD | UML |
|---|---|
| `block` | `block : BlockKind [0..1]` |
| `final` | `final : FinalKind [0..1]` |
| `mixed` | `mixed : xs::boolean` |
| | |
| | |

The UML representation of the `Content` of the `xsd:complexType` is inserted in the `OwnedMember` of the `Class`.

### simple content

```
<xsd:simpleContent
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (restriction | extension))
</xsd:simpleContent>
```

The `xsd:simpleContent` element is mapped to a `Class` with the stereotype `<<simpleConent>>`.

The `Name` of the `Class` is set to `"<simpleContent`.

### simple content : restriction

```
<xsd:restriction
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (simpleType?, (minExclusive |
minInclusive | maxExclusive | maxInclusive | totalDigits
| fractionDigits | length | minLength | maxLength |
```

```
enumeration | whiteSpace | pattern)*)?, ((attribute |
attributeGroup)*, anyAttribute?))
</xsd:restriction>
```

The `xsd:restriction` element is mapped to a `Generalization` with the stereotype `<<restriction>>`.

The attribute `base` is mapped to the `Parent` of the `Generalization`.

If the `Content` of the `xsd:restriction` is a `simpleType, attribute, attributeGroup or anyAttribute`, the corresponding UML element is inserted in the `OwnedMember` of the UML element representing parent element of the `xsd:restriction`. In other case the `Content` of the `xsd:restriction` is mapped to the attributes of the stereotype `<<restriction>>` according to the table:

| Content | Stereotype attribute |
|---|---|
| minExclusive | minExclusive : anySimpleTypeRestriction |
| minInclusive | minInclusive : anySimpleTypeRestriction |
| maxInclusive | maxInclusive : anySimpleTypeRestriction |
| maxExclusive | maxExclusive : anySimpleTypeRestriction |
| totalDigits | totalDigits : positiveIntegerRestriction |
| fractionDigits | fractionDigits : nonNegativeIntegerRestriction |
| length | length : nonNegativeIntegerRestriction |
| minLength | minLength : nonNegativeIntegerRestriction |
| maxLength | maxLength : nonNegativeIntegerRestriction |
| enumeration | enumeration : Charstring [0..*] |
| whiteSpace | whiteSpace : whiteSpaceRestriction |
| pattern | pattern : stringRestriction |

**simple content : extension**

```
<xsd:extension
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
```

```
  Content: (annotation?, ((attribute | attributeGroup)*,
anyAttribute?))
</xsd:extension>
```

The `xsd:extension` element is mapped to a `Generalization` with the stereotype `<<extension>>`.

The attribute `base` is mapped to the `Parent` of the `Generalization`.

The UML element representing `Content` of the `xsd:extension` is inserted in the `OwnedMember` of the UML element representing parent element of the `xsd:extension`.

**simple content : attrbute group**

```
<xsd:attributeGroup
  id = ID
  ref = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</xsd:attributeGroup>
```

The `xsd:attributeGroup` element is mapped to an `Attribute` with the stereotype `<<attributeGroup>>`.

The `Name` of the `Attribute` is set to `"<attributeGroup>"`.

The value of the `ref` attribute is set as `Type` of the `Attribute`. A `Dependency` with the stereotype `<<ref>>` is created in the Attribute. The `Supplier` of the `Dependency` is set to the value of the `ref` attribute.

**simple content: anyAttribute**

```
<xsd:anyAttribute
  id = ID
  namespace = ((##any | ##other) | List of (anyURI |
(##targetNamespace | ##local)) )  : ##any
  processContents = (lax | skip | strict) : strict
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</xsd:anyAttribute>
```

The `xsd:anyAttribute` element is mapped to an `Attribute` with the stereotype `<<anyAttribute>>`.

The `Name` of the `Atttribute` is set to `"<anyAttribute>"`.

The attributes of the `xsd:anyAttribute` are mapped to attributes of the stereotype `<<anyAttribute>>` according to the table:

| XSD | UML |
|---|---|
| namespace | namespace : namespaceSpecification |
| processContents | processContents : ProcessKind [0..1] |

**complex content**

```
<xsd:complexContent
  id = ID
  mixed = boolean
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (restriction | extension))
</xsd:complexContent>
```

The `xsd:complexContent` is mapped to a `Class` with the stereotype `<<complexContent>>`.

The `Name` of the `Class` is set to `"<complexContent>"`.

The attributes of the `xsd:complexContent` are mapped to the attributes of the stereotype `<<complexContent>>` according to the table:

| XSD | UML |
|---|---|
| mixed | mixed : xs::boolean [0..1] |
| | |

**complex content:restriction**

```
<xsd:restriction
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (group | all | choice |
sequence)?, ((attribute | attributeGroup)*,
anyAttribute?))
</xsd:restriction>
```

The `xsd:restriction` element is mapped to a `Generalization` with the stereotype `<<restriction>>`.

The `base` attribute is mapped to the `Parent` of the `Generalization`.

UML representation of the Content of the xsd:restriction is inserted in the OwnedMember of UML representation of the parent element.

**complex content: extension**

```
<xsd:extension
  base = QName
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, ((group | all | choice |
sequence)?, ((attribute | attributeGroup)*,
anyAttribute?)))
</xsd:extension>
```

The xsd:extension element is mapped to a Generalization with the stereotype <<extension>>.

The base attribute is mapped to the Parent of the Generalization.

UML representation of the Content of the xsd:extension is inserted in the OwnedMember of UML representation of the parent element.

**attribute group definition**

```
<xsd:attributeGroup
  id = ID
  name = NCName
  ref = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, ((attribute | attributeGroup)*,
anyAttribute?))
</xsd:attributeGroup>
```

If the xsd:attributeGroup element has the ref attribute, it is mapped to an Attribute with the stereotype <<attributeGroup>>. The attribute name is mapped to the Name of the Attribute. The value of the attribute ref is mapped to the Type of the Attribute. A Dependency with the stereotype <<ref>> is insertd in the Attribute. The Supplier of the Dependency is set to the value of the ref attribute.

If the xsd::attributeGroup element does not have the ref attribute, it is mapped to a Class with the stereotype <<attributeGroup>>. The attribute name is mapped to the Name of the Class. UML representation of the Content of the xsd:attributeGroup is inserted in the OwnedMember of UML representation of the parent element.

**model group definition**

```
<xsd:group
```

```
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded)  : 1
  minOccurs = nonNegativeInteger : 1
  name = NCName
  ref = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (all | choice | sequence)?)
</xsd:group>
```

If the `xsd:group` element has the `ref` attribute, it is mapped to an `Attribute` with the stereotype `<<group>>`. The attribute `name` is mapped to the `Name` of the `Attribute`. The value of the attribute `ref` is mapped to the `Type` of the `Attribute`. A `Dependency` with the stereotype `<<ref>>` is insertd in the Attribute. The `Supplier` of the `Dependency` is set to the value of the `ref` attribute.

If the `xsd::group` element does not have the `ref` attribute, it is mapped to a `Class` with the stereotype `<<group>>`. The attribute `name` is mapped to the `Name` of the `Class`. UML representation of the `Content` of the `xsd:group` is inserted in the `OwnedMember` of UML representation of the parent element.

Other attributes of the `xsd:group` element are mapped to the attributes of the stereotype `<<group>>` according to the table:

| XSD | UML |
|---|---|
| maxOccurs | maxOccurs : MaxOccurence |
| minOccurs | minOccurs : xs::nonNegativeInteger [0..1] |

**model group schema component: all**

```
<xsd:all
  id = ID
  maxOccurs = 1 : 1
  minOccurs = (0 | 1) : 1
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, element*)
</xsd:all>
```

The `xsd:all` element is mapped to a `Class` with the stereotype `<<all>>`.

The `Name` of the `Class` is set to `"<all>"`.

The UML representation of the `Content` of the `xsd:all` is inserted in the `OwnedMember` of the `Class`.

The attributes of the `xsd:all` are mapped to the attributes of the stereotype `<<all>>` according to the table:

| XSD | UML |
|---|---|
| maxOccurs | maxOccurs : MaxOccurence |
| minOccurs | minOccurs : xs::nonNegativeInteger [0..1] |

**model group schema component: choice**

```
<xsd:choice
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded)  : 1
  minOccurs = nonNegativeInteger : 1
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (element | group | choice |
sequence | any)*)
</xsd:choice>
```

The `xsd:choice` element is mapped to a `Class` with the stereotype `<<choice>>`.

The `Name` of the `Class` is set to `"<choice>"`.

The UML representation of the `Content` of the `xsd:choice` is inserted in the `OwnedMember` of the `Class`.

The attributes of the `xsd:all` are mapped to the attributes of the stereotype `<<choice>>` according to the table:

| XSD | UML |
|---|---|
| maxOccurs | maxOccurs : MaxOccurence |
| minOccurs | minOccurs : xs::nonNegativeInteger [0..1] |

**model group schema component: sequence**

```
<xsd:sequence
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded)  : 1
  minOccurs = nonNegativeInteger : 1
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (element | group | choice |
```

```
sequence | any)*)
</xsd:sequence>
```

The `xsd:sequence` element is mapped to a `Class` with the stereotype `<<sequence>>`.

The `Name` of the `Class` is set to `"<sequence>"`.

The UML representation of the `Content` of the `xsd:sequence` is inserted in the `OwnedMember` of the `Class`.

The attributes of the `xsd:sequence` are mapped to the attributes of the stereotype `<<sequence>>` according to the table:

| XSD | UML |
|---|---|
| maxOccurs | maxOccurs : MaxOccurence |
| minOccurs | minOccurs : xs::nonNegativeInteger [0..1] |

**wildcard schema component : any**

```
<xsd:any
  id = ID
  maxOccurs = (nonNegativeInteger | unbounded)  : 1
  minOccurs = nonNegativeInteger : 1
  namespace = ((##any | ##other) | List of (anyURI |
(##targetNamespace | ##local)) )  : ##any
  processContents = (lax | skip | strict) : strict
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</xsd:any>
```

The `xsd:any` element is mapped to an `Attribute` with the stereotype `<<any>>`.

The `Name` of the `Attribute` is set to `"<any>"`.

The attributes of the `xsd:any` are mapped to the attributes of the stereotype `<<any>>` according to the table:

| XSD | UML |
|---|---|
| maxOccurs | maxOccurs : MaxOccurence |
| minOccurs | minOccurs : xs::nonNegativeInteger [0..1] |
| namespace | namespace : NamespaceSpecification [0..1] |
| processContents | processContents : ProcessKind [0..1] |

### identity-constraint definition schema component:unique

```
<xsd:unique
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (selector, field+))
</xsd:unique>
```

The `xsd:unique` element is mapped to an `InformalConstraint` with the stereotype `<<unique>>`.

The attributes of the `xsd:unique` element are mapped to the attributes of the stereotype `<<unique>>` according to the table:

| XSD | UML |
|---|---|
| name | name : xs::NCName |

### identity-constraint definition schema component:key

```
<xsd:key
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (selector, field+))
</xsd:key>
```

The `xsd:key` element is mapped to an `InformalConstraint` with the stereotype `<<key>>`.

The attributes of the `xsd:key` element are mapped to the attributes of the stereotype `<<key>>` according to the table:

| XSD | UML |
|---|---|
| name | name : xs::NCName |

### identity-constraint definition schema component:keyref

```
<keyref
  id = ID
  name = NCName
  refer = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (selector, field+))
</keyref>
```

The `xsd:keyref` element is mapped to an `InformalConstraint` with the stereotype `<<keyref>>`.

The attributes of the `xsd:keyref` element are mapped to the attributes of the stereotype `<<keyref>>` according to the table:

| XSD | UML |
|---|---|
| name | name : xs::NCName |
| refer | refer : xs::QName |

### identity-constraint definition schema component:selector

```
<xsd:selector
  id = ID
  xpath = a subset of XPath expression
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</xsd:selector>
```

The `xsd:selector` element is mapped to an `InformalConstraint` with the stereotype `<<selector>>`.

The attributes of the `xsd:selector` element are mapped to the attributes of the stereotype `<<selector>>` according to the table:

| XSD | UML |
|---|---|
| xpath | name : xs::string |

### identity-constraint definition schema component:field

```
<xsd:field
  id = ID
  xpath = a subset of XPath expression
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</xsd:field>
```

The `xsd:field` element is mapped to an `InformalConstraint` with the stereotype `<<field>>`.

The attributes of the `xsd:field` element are mapped to the attributes of the stereotype `<<field>>` according to the table:

| XSD | UML |
|---|---|
| xpath | name : xs::string |

### notation declaration

```
<xsd:notation
  id = ID
  name = NCName
  public = token
  system = anyURI
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</xsd:notation>
```

The `xsd:notation` element is mapped to an `Artifact` with the stereotype `<<notation>>`.

The `name` of the `xsd:notation` is mapped to the `Name` of the `Artifact`.

Other attributes of the `xsd:notation` are mapped to the attributes of the stereotype `<<notation>>` according to the table:

| XSD | UML |
|---|---|
| public | public : xs::token [0..1] |
| system | system : xs::anyURI [0..1] |

**annotation**

```
<xsd:annotation
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (appinfo | documentation)*
</xsd:annotation>
```

If the xsd:annotation is defined in the xsd:schema or in the xsd:redefine, it is mapped to an Artifact with the stereotype <<annotation>>.

In other cases, the xsd:annotation is mapped to the stereotype <<annotation>> which is applied to the UML element representing parent element of the xsd:annotation

**annotation : appinfo**

```
<xsd:appinfo
  source = anyURI
  {any attributes with non-schema namespace . . .}>
  Content: ({any})*
</xsd:appinfo>
```

The xsd:appinfo element is mapped to a Comment with the stereotype <<appinfo>>.

The attributes of the xsd:appinfo are mapped to the attributes of the stereotype <<appinfo>> according to the table:

| XSD | UML |
|---|---|
| source | source : xs::anyURI [0..1] |

The Content of the xsd:appinfo is mapped to the Text of the Comment.

**annotation : documentation**

```
<xsd:documentation
  source = anyURI
```

```
  xml:lang = language
  {any attributes with non-schema namespace . . .}>
  Content: ({any})*
</xsd:documentation>
```

The `xsd:documentation` element is mapped to a `Comment` with the stereotype `<<documentation>>`.

The attributes of the `xsd:documentation` are mapped to the attributes of the stereotype `<<documentation>>` according to the table:

| XSD | UML |
|---|---|
| `source` | `source : xs::anyURI [0..1]` |
| `xml:lang` | `'xml:lang' : xs::language [0..1]` |

The `Content` of the `xsd:documentation` is mapped to the `Text` of the `Comment`.

### simple type definition

```
<xsd:simpleType
  final = (#all | List of (list | union | restriction))
  id = ID
  name = NCName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (restriction | list | union))
</xsd:simpleType>
```

The `xsd:simpleType` element is mapped to a `DataType` with the stereotype `<<simpleType>>`.

The `name` of the `xsd:simpleType` is mapped to the `Name` of the `DataType`.

The attributes of the `xsd:simpleType` are mapped to the attributes of the stereotype `<<simpleType>>` according to the table:

| XSD | UML |
|---|---|
| `final` | `final : FinalKind` |

### simple type : restriction

```
<xsd:restriction
  base = QName
```

```
  id = ID
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, (simpleType?, (minExclusive |
minInclusive | maxExclusive | maxInclusive | totalDigits
| fractionDigits | length | minLength | maxLength |
enumeration | whiteSpace | pattern)*))
</xsd:restriction>
```

The `xsd:restriction` element is mapped to a `Generalization` with the stereotype `<<restriction>>`.

The attribute `base` is mapped to the `Parent` of the `Generalization`.

If the `Content` of the `xsd:restriction` is a `simpleType`, the corresponding UML element is inserted in the `OwnedMember` of the UML element representing parent element of the `xsd:restriction`. In other case the `Content` of the `xsd:restriction` is mapped to the attributes of the stereotype `<<restriction>>` according to the table:

| Content | Stereotype attribute |
|---|---|
| minExclusive | minExclusive : anySimpleTypeRestriction |
| minInclusive | minInclusive : anySimpleTypeRestriction |
| maxInclusive | maxInclusive : anySimpleTypeRestriction |
| maxExclusive | maxExclusive : anySimpleTypeRestriction |
| totalDigits | totalDigits : positiveIntegerRestriction |
| fractionDigits | fractionDigits : nonNegativeIntegerRestriction |
| length | length : nonNegativeIntegerRestriction |
| minLength | minLength : nonNegativeIntegerRestriction |
| maxLength | maxLength : nonNegativeIntegerRestriction |
| enumeration | enumeration : Charstring [0..*] |
| whiteSpace | whiteSpace : whiteSpaceRestriction |
| pattern | pattern : stringRestriction |

**simple type : list**

```
<xsd:list
```

```
  id = ID
  itemType = QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, simpleType?)
</xsd:list>
```

The `xsd:list` element is mapped to a `DataType` with the stereotype `<<list>>`.

The `Name` of the `DataType` is set to `"<list>"`.

The attributes of the `xsd:list a`re mapped to the attributes of the stereotype `<<list>>` according to the table:

| XSD | UML |
|---|---|
| itemType | itemType : xs::QName |

**simple type : union**

```
<xsd:union
  id = ID
  memberTypes = List of QName
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?, simpleType*)
</xsd:union>
```

The `xsd:union` is mapped to a `DataType` with the stereotype `<<union>>`.

The `Name` of the `DataType` is set to `"<union>"`.

The attributes of the `xsd:union a`re mapped to the attributes of the stereotype `<<union>>` according to the table:

| XSD | UML |
|---|---|
| memberTypes | memberTypes : xs::QName [*] |

**schema**

```
<xsd:schema
  attributeFormDefault = (qualified | unqualified) :
unqualified
  blockDefault = (#all | List of (extension |
restriction | substitution))  : ''
  elementFormDefault = (qualified | unqualified) :
```

```
unqualified
  finalDefault = (#all | List of (extension |
restriction | list | union))  : ''
  id = ID
  targetNamespace = anyURI
  version = token
  xml:lang = language
  {any attributes with non-schema namespace . . .}>
  Content: ((include | import | redefine | annotation)*,
(((simpleType | complexType | group | attributeGroup) |
element | attribute | notation), annotation*)*)
</xsd:schema>
```

The `xsd:schema` element is mapped to a `Package` with the stereotype `<<schema>>`.

The `targetNamespace` attribute is mapped to the `Name` of the `Package`. If the `targetNamespace` is omitted, the `Name` of the `Package` is set to `"<schema>"`.

The other attributes of the `xsd:schema` element are mapped to the attributes of the stereotype `<<schema>>` according to the table:

| XSD | UML |
|---|---|
| attributeFormDefault | attributeFormDefault : FormKind |
| blockDefault | blockDefault : BlockKind [0..1] |
| elementFormDefault | elementFormDefault : FormKind |
| finalDefault | finalDefault : FinalKind [0..1] |
| version | version : xs::token [0..1] |
| xml:lang | 'xml:lang' : xsd::language [0..1] |

### include element

```
<xsd:include
  id = ID
  schemaLocation = anyURI
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</xsd:include>
```

The `xsd:include` element is mapped to a `Dependency` with the stereotype `<<include>>`.

The attributes of the `xsd:include` are mapped to the attributes of the stereotype `<<include>>` according to the table:

| XSD | UML |
|---|---|
| schemaLocation | schemaLocation : xs::anyUri |

**redefine element**

```
<xsd:redefine
  id = ID
  schemaLocation = anyURI
  {any attributes with non-schema namespace . . .}>
  Content: (annotation | (simpleType | complexType |
group | attributeGroup))*
</xsd:redefine>
```

The `xsd:redefine` element is mapped to a `Package` with the stereotype `<<redefine>>`.

The `Name` of the `Package` is set to `"<redefine>"`.

The attributes of the `xsd:redefine` are mapped to the attributes of the stereotype `<<redefine>>` according to the table:

| XSD | UML |
|---|---|
| schemaLocation | schemaLocation : xs::anyUri |

**import element**

```
<xsd:import
  id = ID
  namespace = anyURI
  schemaLocation = anyURI
  {any attributes with non-schema namespace . . .}>
  Content: (annotation?)
</xsd:import>
```

The `xsd:import` element is mapped to a `Dependency` with the stereotype `<<import>>`.

The attributes of the `xsd:import` are mapped to the attributes of the stereotype `<<import>>` according to the table:

| XSD | UML |
|---|---|
| schemaLocation | schemaLocation : xs::anyUri [0..1] |
| namespace | namespace : xs::anyUri [0..1] |

## Postprocessing

In order to reduce the number of levels in the imported UML model, the XSD Importer performs postprocessing when import of the XSD document is completed. During the postprocessing the XSD Importer merges some elements in the imported UML model.

The following merge rules are applied recursively (until merge is not possible) to the built model:

- A `Class` with the stereotype `<<complexType>>` is merged with a nested `Class` with the stereotype `<<comlexContent>>`.
- A `Class` with the stereotype `<<complexType>>` is merged with a nested `Class` with the stereotype `<<simpleContent>>`.
- A `Class` with the stereotype `<<complexType>>` is merged with a nested `Class` with the stereotype `<<all>>`.
- A `Class` with the stereotype `<<complexType>>` is merged with a nested `Class` with the stereotype `<<sequence>>`.
- A `Class` with the stereotype `<<complexType>>` is merged with a nested `Class` with the stereotype `<<choice>>`.

The merge between two elements `E1` and `E2` (`E1` owns `E2`) is possible if and only if:

- Both elements `E1` and `E2` do not have the `<<xmlNamespace>>` stereotype.
- Both elements `E1` and `E2` do not have the `<<xmlAttribute>>` stereotype.
- If element `E1` has any of the stereotypes `<<sequence>>,<<all>> or <<choice>>` and element `E2` has any of these stereotype then merge is not possible.
- Both elements `E1` and `E2` do not have comments.

# XML Namespace Mapping Rules

Elements and attributes in a XML document may be placed in a XML namespace using the mechanisms described in the related specification [http://www.w3.org/TR/REC-xml-names/].

The Tau XML Framework supports XML namespace declarations by means of the special stereotype `<<xmlNamespace>>` which is defined in the profile `TTDXmlFramework`.

The stereotype `<<xmlNamespace>>` has the following attribute:

```
declarations : xmlNamespaceDecl[*]
```

The type `xmlNamespaceDecl` is defined as follows:

```
class xmlNamespaceDecl
{
    public Charstring name;
    public Charstring [0..1] prefix;
}
```

The instance of `xmlNamespaceDecl` with empty defines default namespace.

The XML namespace declarations on a XML element are mapped to one instance of the stereotype `<<xmlNamespace>>` according to the following rule:

XML:

```
<element
xmlns:prefix1="ns1"
xmlns:prefix2="ns2"
...
xmlns="default ns"/>
```

UML:

```
<<<xmlNamespace (. declarations =
{
xmlNamespaceDecl (. name = "ns1", prefix = "prefix1" .),
xmlNamespaceDecl (. name = "ns2", prefix = "prefix2" .),
...
xmlNamespaceDecl (. name = "ns1" .)
}
.)>>
```

# *XML Schema Modeling with UML*

The chapters listed under XML Schema Modeling with UML describe how to model XML Schemas using UML and how to import and export schema (XSD) files.

# 69

# *Modeling XML Schemas*

This chapter describes how to model XML Schema in UML.

# Getting started

To start modeling a XML schema in Tau:

- Start Tau and create a new “UML for XSD Modeling” project
- Create a XML Schema Diagram in the schema and start modeling.

To generate a XSD file from the schema model:

- Right-click the schema and select **Generate XSD...**
- Select a folder, enter a file name and click **Save**

**See also**

“XMLFramework add-in” on page 1895

“XSD view” on page 1896

“XSD profile” on page 1897

“Importing XSD Files” on page 1897

“Generating XSD Files” on page 1897

# XMLFramework add-in

The XMLFramework add-in adds support for modeling XML schemas in Tau.

The main features of the add-in are:

- A XSD profile for annotating UML models with XSD information
- A XSD centric model view, the XSD view.

To activate the add-in, see Activating the XMLFramework add-in.

## Activating the XMLFramework add-in

To activate the XMLFramework add-in:

1. In the **Tools** menu, select **Customize...**.
2. Click the Add-Ins tab and check the `XMLFramework` add-in.
3. Click **OK**.

# XSD view

The XSD View provides a XSD centric view over a model. In this view only XSD elements are shown, and only XSD elements can be created.

When using the New... command in the Model View context menu, the view enforces a correct XSD structure.

To activate the XSD View:

1. In the **View** menu, select **Reconfigure Model View...**
2. Select **XSD View** in the dialog and click **OK**.

You can switch between the Standard View and the XSD View at any time.

**Note**

*Not all UML elements are visible in the XSD View. To see all elements switch to the Standard View.*

**See also**

"Model View" on page 19 in Chapter 4, *Introduction to Tau 4.2*

"Default Model View" on page 2462 in Chapter 91, *Dialog Help*

# XSD profile

The XSD profile extends UML with concepts for modeling XML schemas.

The `xs` package in the profile contains all primitive types from the XML Schema specification.

## *Importing XSD Files*

For information on how to import and existing XSD file, see WSDL/XSD Importer Reference.

## *Generating XSD Files*

For information on how to generate XSD files from the model, see WSDL/XSD Importer Reference.

**Note**

*XSD files can be generated from any UML package. If the content is not annotated with XSD information from the XSD profile, a default mapping will be used.*

# *Exploring UML Models*

The chapters listed under this section describes how to use the Tau Explorer.

# 71

# *The Tau Explorer*

This chapter is a reference to the Explorer user interface and a reference for the terminology used when exploring a system.

**Note**

*The Tau Explorer is available on Windows only.*

# Exploring an Application

## Underlying Principles and Terms

The Tau 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 UML are based on the idea of an automatic generation of the reachable state space for the UML systems.

### Behavior Trees

The Tau Explorer operates on structures known as *behavior trees* or reachability graphs. A behavior tree is a tree structure that represents the behavior of a UML system.

The nodes of the tree represent UML *system states*. A system state is defined by:

- The active class instances that are active
- The variable values of these active classes
- The UML control flow state of the active class instances
- The local state of any called operations (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 UML events that transfers the UML system from one system state to another. Therefore, the edges are also called *behavior tree transitions*. They can be individual UML statements like tasks, inputs, outputs, etc. but also complete UML 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 a UML 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 UML 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 an 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 UML 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 active class 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.

## Performing Automatic State Space Explorations

This section describes how to perform an automatic state space exploration and how to examine the results.

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 UML systems.
- Random walk, a simple algorithm that can be used for very large UML systems.
- Exhaustive exploration, an algorithm suited only for small UML systems.

The characteristics of these algorithms are further described in the sections describing their respective options views: Bit State Options View, Random Walk Options View and Exhaustive Options View. 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 further described in section State Space Options View.

### Rules Checked During Exploration

During state space exploration, a number of rules are checked to detect errors or possible problems in the UML 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 UML 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 Define-Rule 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.
- `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 UML 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 Options View). 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.
- `Symbol coverage: x`
  The percentage of the UML 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.

## Generating and Starting an Explorer

This section describes how to build and run a Model Explorer application.

**Important!**

*You must have a C/C++ compiler installed to generate an executable Model Explorer application. The compiler also needs to be in your PATH.*

### Creating a Build Artifact

To activate the Model Explorer, make sure that the *ModelExplorer* checkbox is checked in *Tools -> Customize -> Add-ins*.

In order to build a Model Explorer application a Build Artifact has to be created by right clicking in the Model View and selecting *Model Explorer -> New Artifact* in the context menu. A new build artifact will be created. Right click on it and choose *Select build root...* to set which model element that should be the build root for this build.

Another option is to right click on the desired build root in the Model View and select *Model Explorer -> New Artifact*.

### Building and Launching the Explorer application

Right clicking on the build artifact and selecting *Build (Model Explorer)* opens a sub-menu with the following options: *Check*, *Generate*, *Make*, *Build*, *Launch* and *Clean*. To build, just select the *Build* command.

### Customizing the Build Artifact

The following properties can be set in the Model Explorer build artifact:

- Target Directory, specifies where to put the files generated by the Model Explorer.
- Error Limit. If the number of errors associated with a build exceeds this limit, the build is aborted.
- Target Kind, specifies what compiler to use for the generated files.
- Make-Template File, instructs the make tool to include a user created template file.
- Supress C Level warnings, filters out warnings from the c compiler and linker.
- Generate reference package. If set, the generated files will be added to the model in a package named “Result of C generation”.
- TCP/IP Port, which port Tau will use for the communication with the explorer application.

- Launch Console. If set, a read-only console window will be opened echoing the commands sent from the Explorer UI and the responses from the Explorer application.

**Note**

*To remove a property from the build artifact you need to select Delete Value from the context menu since an empty property can be valid in many cases. A yellow input field signals that the property is not set.*

## The Explorer User Interface

The Explorer User Interface (Explorer UI) is opened when the command *Launch* on the Explorer build artifact is issued, or by selecting the menu command *Show Model Explorer*, found in the View menu in Tau.

**Note**

*It is not possible to open the Explorer UI from the file system, it must be opened by Tau.*

### Explorer states

When the Explorer UI is open it will automatically detect the state of the Explorer executable and will all the time keep track on its state.The Explorer UI can be in four different states:

- Explorer is not running. The Explorer UI is disabled and gray.
- Explorer is starting up. This is indicated by a blinking Tau icon.
- Explorer is ready and is waiting for a command. The Explorer UI is enabled and has a standard Tau icon.
- Explorer is executing a command. This is indicated by a rotating Tau icon and the Explorer UI is disabled. In this state it is only possible to send commands from the Explorer command prompt or select the *Break* and *Exit* options.

### Explorer Views

The Explorer UI is based on a set of views, accessable via the menu:

- Explore View
- Navigator View
- Reports View

- Test Values View
- General Options View
- Bit State Options View
- Report Options View
- State Space Options View
- Random Walk Options View
- Exhaustive Options View
- Tree Search Options View

### Explorer command prompt

In all views you are able to send commands to the Explorer through the Explorer command prompt. The command prompt is found under the menu. Enter a command and press enter (or click the *Send Command* button) to send a command to the Explorer. Typically the result from the command is to be found in the Log Window.

### Log Window

All information sent to and received from the Explorer executable is shown in the output tab *Model Explorer* in Tau.

### Explore View

In this view there is a sub menu with four different explorations:

- Bit State, see the command Bit-State-Exploration.
- Random Walk, see the command Random-Walk.
- Exhaustive, see the command Exhaustive-Exploration.
- Tree Search, see the command Tree-Search.

When selecting one of these options, the corresponding Explorer command is sent asynchronously via Tau to the Explorer executable.

### Navigator View

The Explorer provides the possibility to interactively walk around in the behavior tree of a UML system. This is also known as manually *navigating* in the state space.

The Navigator is intExplorerended 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 a UML 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 UML 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.

*Moving up in the behaviour tree*

To move one level up in the behavior tree, select the link describing the previous state, found right under the *Up Transition* heading. If the Explorer executable is already in the first node in the behaviour tree, there is no link available and the text *No up node* is shown. The Explorer UI will now send the *Up 1* command to the Explorer UI. When this is done, the Navigator View will send several commands to investigate what the new up state will be and what available next states are available.

*Moving down in the behaviour tree*

To move down in the behaviour tree, select one of the links under the *Down Transition* heading. If there are no more states from the current state, the list of down transitions will be empty. When selecting one of the down transitions, the Explorer UI will send the command *Next x*, where x is the index in the list of available down transitions corresponding to the selected link. When this is done, the Navigator View will send several commands to investigate what the new up state will be and what available next states are available.

*Locate in Model*

When selecting the *Locate in Model* option, the Explorer UI will present a list of elements in the output tab *Explorer System State*. Each element is a part of the current state space. When clicking on the elements in that list, the corresponding element is located. Typically a diagram will be opened and the specified element will be selected.

It is of course possible to navigate in the state space using the command prompt. There are several commands available for this, see Alphabetical List of Commands.

*Open Sequence Diagram Trace*

When selecting the *Open Sequence Diagram Trace* a new tab will open in Tau with a Sequence Diagram displaying the events that led to the current location in the state space. This is usually very helpful when trying to understand the events that led up to a report being generated. A Sequence Diagram can also be launched with the command Generate-SQD-Trace.

**Note**

*Sending commands via the command prompt will not refresh the Explorer UI, so using textual navigation commands while in the Navigator View will make the state in the Navigator View obsolete. Therefore it is highly recommended to refresh the Explorer UI before taking a transition in the Navigator View after issuing textual commands. The easiest way to do this is to click on the Navigate option in the menu. This is valid for all views (no views are updated after sending a command to the Explorer executable) but most obvious in the Navigator view.*

### Reports View

After running explorations the reports View will show a list of found problems in your UML model. Clicking on one of the reports will make the Explorer executable go to the state where this problem occured, and then the Explorer UI will show the Navigator View. This will enable debugging of the specific problem. If no reports are found by the Explorer executable, this view will be empty. Each report will have a *Show Trace* option that opens a new tab with a Sequence Diagram displaying the events that led up to that report.

### Test Values View

This view will show defined test values and makes it possible to delete them or add new test values. There are three types of test values:

- Signals
- Values
- Parameters

When the Test Values View is selected, the Explorer UI sends a set of commands to check what test values that are defined. This is also done after adding or deleting test values. The defined test values are then listed in the *Signal Definitions*, *Test Values* and *Parameter Test Values* lists. For more information on test values, see Defining Signals from the Environment.

*Adding a signal test value*

If a signal doesn't have any parameters, you add a signal by adding the signal name in the *Add Signal* section. If the signal has parameters you need to also add values for each parameter using space as separator. Example:

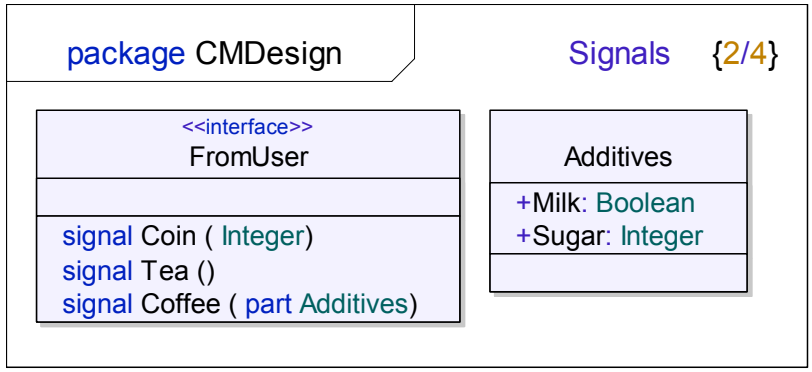
```
Coffee true 1
```

*Adding a value test value*

To add a value, add the value in the first text field and its type in the second field in the *Add Value* section.

*Adding a parameter test value*

To add a parameter test value for a certain signal, add the signal name in the third text field, the parameter number in the second test field and value for the parameter in the first text field. Example:




Value:

```
true 2
```

Parameter:

```
1
```

Signal:

```
Coffee
```

*Deleting test values*

To delete a test value, select the *Remove* link next to the defined test value.

To delete all test values of a certain type, select the *Remove All* link next to the heading for that type.

### General Options View

Selecting the Options menu will show this view. It is the first of all the options views. It will also be shown if the *General* sub menu option is selected.

In the General Options View it is possible to issue the following options:

- Set Advanced
- Set options to default
- Reset all options
- Show all options

*Set Advanced*

*Set Advanced* will send the following commands to the Explorer executable:

```
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).

*Set options to default*

See the command Default-Options.

Before this command is sent, the Explorer UI will show a confirmation dialog. This makes it possible to abort the command.

*Reset all options*

See the command Reset.

Before this command is sent, the Explorer UI will show a confirmation dialog. This makes it possible to abort the command.

*Show all options*

See the command Show-Options.

### Bit State Options View

Bit state exploration is an efficient automatic state space exploration algorithm for reasonably large UML systems. 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.

In the Bit State Options View it is possible to set the following options:

- Iteration Step
- Search Depth
- Hash size

*Iteration Step*

Default value is 0, i.e. the feature is not activated.

See also the command Define-Bit-State-Iteration-Step.

*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 is 100.

See the command Define-Bit-State-Depth.

*Hash 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 is 1,000,000 (bytes)

See also the command Define-Bit-State-Hash-Table-Size.

### Report Options View

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.

The default value is Prune for all report types.

See also the commands Define-Report-Continue, Define-Report-Prune and Define-Report-Abort.

**Note**

*For some report types, like Deadlock, the Continue choice is impossible.*

*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. However, the report action still is performed, even though the report is not logged.

The default value is On for all report types.

See also the commands Define-Report-Log.

### State Space Options View

The structure and size of the state space that can be generated for any given UML 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.

In the State Space Options View it is possible to set the following options:

- Symbol Time
- Maximum State Size
- Timer Progress
- Maximum Instances
- Transition Length
- Input Port Length
- Event Priorities
- Scheduling
- Transition

*Symbol Time*

A common simplification made in the analysis of UML systems is to consider the time it takes 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 an active class 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 UML symbols is Zero or Undefined using this option.

Default value for this option is Zero.

See also the command Define-Symbol-Time.

*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 is 100000 (bytes).

See also the command Define-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 UML 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

Default value is On.

See also the command Define-Timer-Progress.

*Maximum Instances*

To avoid infinite chains of create actions in the state space, the Explorer uses a maximum number of allowed active class instances for any type. If this number is exceeded during state space exploration, a "Create" report is generated.

Default value is 100.

See also the command Define-Max-Instance.

*Transition Length*

To make it possible to detect infinite loops within a transition in the state space, the maximum number of UML 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 is 1000.

See also the command Define-Max-Transition-Length.

*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 is 3.

See also the command Define-Max-Input-Port-Length.

*Event Priorities*

The events that are represented in a behavior tree can be divided into five classes:

- Internal events: Events local to the active classes in the system, e.g., tasks, decisions, inputs, outputs.
- Input from the Envionment: Reception of signals from the environment. The signal is put in the input port of an active class instance or on a connector queue.
- Timeout events: Expiration of UML timers. The timer signal is put in the input port of an active class instance.
- Connector outputs: A signal is removed from a connector queue and put into another connector queue or the input port of an active class' instance.
- Spontaneous transitions: A transition in an active class 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 cur-

rent 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**

*Note that also the setting of the symbol time option will have an impact on the events that can be executed in each system state; see section Symbol Time.*

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 priority for Spontaeous Transitions is 2.

Default priority for Connector Output is 1.

Default priority for Timeout Events is 2.

Default priority for Input from ENV is 2.

Default priority for Internal Events is 1.

See also the command Define-Priorities.

*Scheduling*

The scheduling algorithm defines which of the class instances in a system state will be allowed to execute. There are two possible alternatives:

- All of the active class instances in the ready queue are allowed to execute (the value "All" in the command)
- Only the first active class instance in the ready queue is allowed to execute (the value "First" in the command).

Default value is First.

See also the command Define-Scheduling.

*Transition*

There are two alternatives possible for the type of a behavior tree transition during state space exploration:

It can be equal to a complete UML class graph transition (the value "UML" in the command)

It can be a part of such an UML transition (the value "Symbol-Sequence" in the command).

If it is equal to a UML class graph transition, whenever such a transition is started, it is completed before anything else is allowed to happen. This implies that all active class instances in all system states in the behavior tree will always be in an UML class graph state.

If it is only a part of an UML process graph transition, a transition in the behavior tree is considered to be a sequence of events that are local to the active class 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 class instances.

Default value is UML.

See also the command Define-Transition.

### Random Walk Options View

Random walk is an automatic state space exploration algorithm that can be useful for very large UML 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.

In the Random Walk Options View it is possible to set the following options:

- Search Depth
- Repititions

*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 is 100.

See also the command Define-Random-Walk-Depth.

*Repititions*

The number of times the random walk search will be repeated from the start state before the exploration is finished.

Default value is 100.

See also the command Define-Random-Walk-Repetitions.

### Exhaustive Options View

Exhaustive exploration is an automatic state space exploration algorithm intended for small UML 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 UML 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 is 100.

See also the command Define-Exhaustive-Depth.

### Tree Search Options View

Tree search exploration is an automatic state space exploration algorithm where all possible combinations of actions are executed.

*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 is 100.

See also the command Define-Tree-Search-Depth.

# Guidelines for Model Exploration

## Exploring a UML Model

This section describes how to use the automatic state space exploration facilities in the Explorer to look for inconsistencies and design errors in a UML model. The idea is to test the robustness of the application, the responses to unexpected situations. Essentially the exploration 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 UML system is of moderate size and complexity; techniques for exploring large UML systems are described in Exploring Large Systems.

### Using a Default Exploration

When you use the Explorer for finding errors in a new UML system for the first time, you are advised to start a bit state exploration using the default options.

To explore 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 on the top right of the Explorer UI.
2. You should also make sure you use the default state space and exploration options. Enter the command Default-Options, or click the *Set options to default* option in the Options View.
3. Start a bit state exploration by clicking *Bit State* in the Explore View. Let the exploration run for at least 5-10 minutes.
4. When the exploration is finished, it will print some statistics in the Model Explorer output window. Note the Symbol Coverage value.

5. Go to the Report View to see if any reports were created. When you click on a report you will navigate to that node in the behaviour tree. You can use any of the introspection commands (List-Active-Class, List-Input-Port, Examine-Variable, etc.) to get more information on the specific state or use the tracing and viewing facilities to examine the report. The best start is usually to select the option *Show Trace* to launch a Sequence Diagram displaying the events that led up to the report. This can also be done by the command Generate-SQD-Trace.
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 exploration, as described above. Otherwise, you should check if the exploration is to be considered finished (see below).

### Determining if the Exploration is Finished

When all reports have been checked and the found errors possibly have been corrected, the next question arises: When are we finished exploring 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 exploration; 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.

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 exploration 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 UML 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.

- 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* in the Options View under the Report sub view.

- Some parts of the system are, in fact, unreachable.

If some parts of the UML 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).

- 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 for that exploration method under the Options View.

- The state space is too big.

Many UML 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 Exploring Large Systems.

### Using Advanced Exploration

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 exploration of a system. They are intended first of all to test for internal inconsistencies in the UML system and to get a good 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 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 UML 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 UML systems. It is thus

usually not possible to get a complete coverage of the state space, even if some of the techniques described in Exploring Large Systems 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 for more information.

To set advanced options, click the *Set Advanced* button in the Options View.

## Exploring Large Systems

This section discusses various techniques that are useful when designing and exploring large UML 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*
- *More Efficient Bit-State Exploration*
- *Reducing the State Space Size*
- *Using Random Walk Exploration*
- *Incremental Exploration*

### 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 UML 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 UML 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 Exploration.

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 exploration is finished; otherwise, continue with the next step.
4. Go to a suitable system state and start a new exploration from there.
5. Repeat the process until the symbol coverage is 100%.

There are two issues with 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 UML system. It is usually very helpful to inspect the Sequence Diagrams for the available reports to help identify a good starting place, use the Generate-SQD-Trace command or the *Show Trace* option in the Reports View. Once a system state has been chosen the next issue is how to get there in the Explorer. There are a number of possible ways to do this:

- Using the Navigator View

  Use the buttons in the Navigator View to navigate to the target state. Additionally, the command Command-Log-On can be used to save the navigation to a file. Close the log with Command-Log-Off. The saved commands can be loaded using the command Include-File.

- Using Path commands

  When in a target state, use the command Print-Path to get a path from the root to the current state. At a later stage you can use the command Goto-Path followed by the path printed earlier.

- Using user-defined rules

  Create a user rule that describes the state and other conditions for the target stat, see Managing User-Defined Rules and User-Defined Rules. Use an exploration to get a report for this state. You can then use the path or MSC techniques described above.

**Note**

*If you make changes to your model, path commands and MSC-trace files may become invalid and have to be re-done.*

*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.

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 Managing User-Defined Rules and User-Defined Rules.

### 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.

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. For example the command:

```
define-variable-mode monitor subscrTab skip
```

will make the Explorer skip all `subscrTab` variables in `monitor` in instances of the monitor class.

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 constant 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 Where to Start a Partial Exploration.
2. Redefine the root to the current state, see Define-Root.
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.

### 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 UML system. Usually this is based on the fact that the state space of an UML system contains various “sub state spaces” that are equivalent except for some detail, which is not very interesting for the purpose of the exploration. Some examples of such details are:

- The value of local variables
- The number of active class instances
- 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 an active class 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 especially interesting to try to investigate the behavior of the UML 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 Managing User-Defined Rules, 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 to 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 Active Class Instances*

Another issue is the number of class instances that are used for each class 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. If preferred, it is also possible to use a user-defined rule 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 UML 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 UML system before performing exploration. Another possibility is to skip the variable when computing hash values as described in More Efficient Bit-State Exploration.

*Variables Not Influencing the Dynamic Behavior*

In many situations an UML 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 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. 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 class ‘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 exploring large UML systems. The time and resources available for the exploration may simply be too limited. A possible strategy to use when exploring very large UML 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.

### Incremental Exploration

A common way to develop large UML 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 exploration 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 coverage of the UML 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 UML 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.

## Defining Signals from the Environment

A problem common to all state space exploration techniques is related to the treatment of the environment of the UML 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 Tau Explorer allows two different strategies to avoid situations like this:

1. Create a closed system by specifying the environment of the system using UML. This will solve the problem but introduces a new one; it is necessary to create an UML 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.

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
- Classes and other structured datatypes: All combinations of the test values of the individual fields
- Arrays: All combinations of the test values of the component type.
- Reference types: NULL + pointers to the test values for whatever the reference points to.

### 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 Reducing the State Space Size.*

### 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 Reducing the State Space Size.*

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* View in the Explorer UI.

*Test Values for Data Types*

The following commands operate on the test values for a data type.

- To define a new test value for a sort, use the command Define-Test-Value, or the *Add Value* form in the Explorer UI.
- To list the new test values defined for all types, enter the command List-Test-Values, or look at the *Test Value* section in the Test Value View.
- To clear all test values for a type, enter the command Clear-Test-Values, or click the *Remove All* button. As parameter, you either specify the type, or '-' which means **all** types.

*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 use the *Add Parameter* form. 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 look at the *Signal Definitions* section in the Test Value.
- To clear all test values for a signal parameter, enter the command Clear-Parameter-Test-Values, or click the *Remove All* 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 *Add 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.*

- To list all currently defined signal instances, enter the command List-Signal-Definitions, or look at the *Parameter Test Values* section in the Test Values .
- To clear all test values for a signal type, enter the command Clear-Signal-Definitions, or click the *Remove All* 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 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.

## Exploring Systems with External C/C++ Code

Tau allows the usage of external C/C++ code together with an UML 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/C++ code. However, due to the special requirements of state space exploration, some restrictions must hold for the external C/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/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/C++ code cannot be handled by the Explorer,
- C/C++ unions may not contain pointers, data types implemented by pointers (like the UML types String and Bag) or active class references, and

If there are variables in C/C++ code, this will not be detected by the Explorer. It may appear as if the Explorer works, but the variables defined in C/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/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/C++ union type if the union may contain pointer-based types, since the Explorer cannot know the current type of the union and thus cannot deduce whether to treat the union as a pointer or not. References to active classes are also treated specially in the Explorer and can also not be part of a C/C++ union.

When using dynamic memory allocation in extern C/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/C++ function allocates memory, the Explorer must be informed about the data area that was allocated, and the same holds when a C/C++ function releases memory. This is accomplished by calling two functions from the C/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 type of data was allocated or its size. This is handled automatically by the Explorer simply by finding the UML entity (e.g. a variable) that points at the data area and assuming that the type and size given by this entity is correct. If no UML 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 macro XVALIDATOR_LIB that can be used to check external C/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 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 UML 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 UML 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 an active class of type "Proc" with an attribute "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 active class 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 active class instance "Game:1" is equal to "Winning" and the type of signal to be consumed by the same active class 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.

### Managing User-Defined Rules

To define the user-defined rule 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

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 an expression which calls the the C function `xAssertError`. See the following example.

**Example 617: Using Assertions**

```
[[if (#(I) < #(K))
    xAssertError("I is less than K");
]]
```

In the Tau 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 UML 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.

# Model Explorer Reference

This section provides an alphabetical listing of all available commands in the Explorer. All input to the Explorer is case insensitive.

## Alphabetical List of Commands

## ? (Interactive Context Sensitive Help)

**Parameters:**

```
(None)
```

The Explorer 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 Explorer 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 class instance, or operation given by the current scope.

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 (classes, strings and arrays) by appending the class' attribute name or a valid array index before the value to be assigned. Nested classes and arrays can be handled by entering a list of index values and class attribute names.

If a instance identifier is given within parenthesis, the scope is temporarily changed to this 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 or the *Break* command is sent. 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 View is by default opened. 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.

## Connector-Disable

**Parameters:**

```
<connector> | '-'
```

Disables all test values defined for all signals using the given connector. '-' means *all connectors*. Use Connector-Enable to start using them again. See also Signal-Disable. Test values for a signal are only used if both the signal and the connector that transports the signal are enabled. By default, all signals and connectors are enabled.

## Connector-Enable

**Parameters:**

```
<connector> | '-'
```

Enables all test values defined for all signals using the given connector. '-' means *all connectors*. See also Signal-Enable. Test values for a signal are only used if both the signal and the connector that transports the signal are enabled. By default, all signals and connectors are enabled.

## Clear-Coverage-Table

**Parameters:**

```
(None)
```

Clears the test coverage information.

## 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 Explorer. 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-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 is 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-Connector-Queue

**Parameters:**
`<connector name> ( "On" | "Off" )`

Adds or removes a queue for the specified connector. If a queue is added for a connector, it implies that when a signal is sent that is transported on this connector, it will be put into the queue associated with the connector. By default no connectors have queues.

## Define-Exhaustive-Depth

**Parameters:**
`<Depth>`

The parameter defines the depth of the search when performing exhaustive exploration. The default value is 100.

## 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. 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. 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 UML 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. The default value is 1,000.

## 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>
<Connector 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 State Space Options View.

The default priorities are:

- Internal events: 1
- Input from ENV: 2
- Timeout events: 2
- Connector outputs: 1
- Spontaneous transitions: 2

Note that the priorities of events processed in the Explorer also is affected by the command Define-Symbol-Time.

## 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 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.. 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 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.. 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. 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. If the parameter is specified as '-', all report types will be defined in the same way.

## 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 UML 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 active class instances in the ready queue are allowed to execute at each state. The parameter defines the scheduling as follows:

- `All`
  All active class instances in the ready queue are allowed to execute at each state.
- `First`
  Only the first active class 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 UML 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".

## Define-Symbol-Time

**Parameters:**
`"Zero" | "Undefined"`

The time it takes to execute one symbol, e.g. an input, task or decision, in an UML active class is defined either to be zero or undefined. If it is set to zero, it is assumed that all actions performed by active class 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 active classes to execute symbols. Consider for example a situation where an active class 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-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".

## Define-Transition

**Parameters:**
`"UML" | "Symbol-Sequence"`

Defines the semantics and length of the transitions in the behavior tree. The parameter defines the transitions as follows:

- `UML`
  The behavior tree transitions correspond to complete UML statemachine graph transitions.
- `Symbol-Sequence`
  The behavior tree transitions correspond to the longest sequence of UML symbols that can be executed without any interaction with other active class instances.

The default is "UML".

## Define-Tree-Search-Depth

**Parameters:**
`<Depth>`

Defines the maximum search depth for tree search.

The default value is 100.

## Define-Variable-Mode

**Parameters:**
```
<Active class> [ <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 ac-

count, 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 are 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 data structures 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-Connector-Signal

**Parameters:**
`<connector name> <Entry number>`

The parameters of the signal instance at the position equal to the entry number in the queue of the specified connector are printed. The entry number is the number associated with the signal instance when the command List-Connector-Queue is used.

## Examine-PId

**Parameters:**

```
(None)
```

Information about the active class 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 operation calls made by the active class instance. The list starts with the latest operation call and ends with the active class instance itself.

## Examine-Signal-Instance

**Parameters:**

```
<Entry number>
```

The parameters of the signal instance at the specified position in the input port of the active class 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 active class 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 the value of an attribute of a class, or a component of a string or array variable, by appending the class' attribute name or a valid array index value as an additional parameter. The selection can handle classes and arrays within classes and arrays to any depth by giving a list component selection parameters. 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.

If a PId is given within parenthesis, the scope is temporarily changed to this active class 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 UML 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* command is sent. 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 Explorer asks for confirmation. If any of the Explorer options have been changed, the Explorer 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 Explorer executable 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.

## Generate-SQD-Trace

Launch a Sequence Diagram displaying the events that led to the current location in the state space from the root.

## 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 Explorer 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-Connector-Queue

**Parameters:**

```
<connector name>
```

A list of all signal instances in the specified connector 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-Connector-Signal.

## List-Input-Port

**Parameters:**

```
[ '(' <PId value> ')' ]
```

A list of all signal instances in the input port of the active class 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 active class 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 active class instance. The entry number can be used in the command Examine-Signal-Instance.

If a PId is given within parenthesis information about this active class instance is printed instead.

## 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-Parameter-Test-Values

**Parameters:**
`(None)`

Lists all currently defined test values for signal parameters.

## List-Active-Class

**Parameters:**
`<Optional active class name>`

A list of all active class instances with the specified active class name is printed. If no active class name is specified all active class 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 of active class instances in the ready queue is printed. For more information, see the Model Verifier command List-Ready-Queue.

## 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 oc-

curred. 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-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-Report-File

**Parameters:**
`<File name>`

An existing report file is opened and the reports in it are added to the current reports.

## 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.

## 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-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 Explorer asks for confirmation. If any of the Explorer options have been changed, the Explorer 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 Explorer executable 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* command is sent. The system is then returned to the state it was in before the exploration was started.

When the exploration is finished or stopped, a few statistics are printed; see the command Bit-State-Exploration for an explanation of these.

## 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 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-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 the Model Verifier Print-Coverage-Table command.

## Save-Options

**Parameters:**
`<File name>`

Creates a 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-State-Space

**Parameters:**
`<File name>`

A Labelled Transition System (LTS) representing the generated state space is saved to a file.

**Note**

*It is necessary to have executed an Exhaustive-Exploration command before the state space can be saved on a file.*

## 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 operation call stack. See also the commands Stack, Set-Scope and Scope-Up.

## Scope-Up

**Parameters:**
`(None)`

Moves the scope one step up in the operation call stack. See also the commands Set-Scope, Stack and Scope-Down.

## 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 UML Code Generator. No assumptions are made about the performance of the UML 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 UML
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 C Code Generator. The assumption is made that the time it takes for the UML 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 UML
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 active class, at the bottom operation call. A scope is a reference to an active class instance and possibly a reference to an operation instance called from this process. The scope is used for a number of other commands for examining the local properties of an active class 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 so that no assumptions are made about the performance of the UML 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 so the assumption is made that the time it takes for the UML 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-Mode

**Parameters:**
(None)

This command displays a summary of the current execution mode and some other information about the current state of the Tau Explorer.

## 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-Versions

**Parameters:**
(None)

The versions of the C Code 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 Connector-Disable. Test values for a signal are only used if both the signal and the connector that transports the signal are enabled. By default, all signals and connectors are enabled.

## Signal-Enable

**Parameters:**
<Signal> | '-'

Enables all test values defined for the given signal. '-' means *all signals*. See also Connector-Enable. Test values for a signal are only used if both the signal and the connector that transports the signal are enabled. By default, all signals and connectors 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 operation call stack for the class instance defined by the scope is printed. For each entry in the stack, the type of instance (operation/process), 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).

## 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 sending the *Break* command.

## 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 sending the *Break* command.

Tree Walk creates a number of “TreeWalk” reports.

## 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.

## 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 active class 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 active class instances or signals. The rule variables can be used in active class or signal functions described later in this section.

```
exists <RULE VARIABLE> [: <ACTIVE CLASS TYPE>]
[ | <PREDICATE>]
```

This predicate is true if there exists an active class instance (of the specified type) for which the specified predicate is true. Both the active class type and the predicate can be excluded. If the active class type is excluded all active class instances are checked. If the predicate is excluded it is considered to be true.

```
all <RULE VARIABLE> [ : <ACTIVE CLASS TYPE>]
[ | <PREDICATE>]
```

This predicate is true for all active class instances (of the specified type) for which the specified predicate is true. Both the active class type and the predicate can be excluded. If the active class type is excluded all active class instances are checked. If the predicate is excluded it is considered to be true.

```
siexists <RULE VARIABLE> [ : <SIGNAL TYPE>]
[ - <ACTIVE CLASS INSTANCE>] [ | <PREDICATE>]
```

This predicate is true if there exists a signal (of the specified type) in the input port of the specified active class for which the specified predicate is true. If no signal type is specified, all signals are considered. If no active class instance is specified the input ports of all active class instances are considered. If no predicate is specified it is considered to be true. The specified active class can be either a rule variable that has previously been defined in an exists or all predicate, or a active class instance identifier (<ACTIVE CLASS TYPE>:<INSTANCE NO>).

```
siall <RULE VARIABLE> [ : <SIGNAL TYPE>]
[ - <ACTIVE CLASS INSTANCE>] [ | <PREDICATE>]
```

This predicate is true for all signals (of the specified type) in the input port of the specified active class for which the specified predicate is true. If no signal type is specified all signals are considered. If no active class is specified the input ports of all active class instances are considered. If no predicate is specified it is considered to be true. The specified active class can be either a rule variable that has previously been defined in an exists or all predicate, or a active class instance identifier (<ACTIVE CLASS 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:

- Active class functions: Extract values from active class instances.
- Signal functions: Extract values from signals.
- Global functions: Examine global aspects of the system state.
- UML literals: Conventional UML constant values.

### Active Class Functions

Most of the active class functions must have a class instance as a parameter. This class instance can be either a rule variable that has previously been defined in an exists or all predicate, a class instance identifier (<ACTIVE CLASS TYPE>:<INSTANCE NO>), or a function that returns a class instance, e.g. sender or from.

```
state( <ACTIVE CLASS INSTANCE> )
```

Returns the current state of the class instance.

```
type( <ACTIVE CLASS INSTANCE> )
```

Returns the type of the class instance.

```
iplen( <ACTIVE CLASS INSTANCE> )
```

Returns the length of the input port queue of the class instance.

```
sender( <ACTIVE CLASS INSTANCE> )
```

Returns the value of the imperative operator sender (a class instance) for the active class instance.

```
parent( <ACTIVE CLASS INSTANCE> )
```

Returns the value of the imperative operator parent (a class instance) for the active class instance.

```
offspring( <ACTIVE CLASS INSTANCE> )
```

Returns the value of the imperative operator offspring (a class instance) for the active class instance.

```
self( <ACTIVE CLASS INSTANCE> )
```

Returns the value of the imperative operator self (a class instance) for the active class instance.

```
signal( <ACTIVE CLASS INSTANCE> )
```

Returns the signal that is to be consumed if the active class instance is in a UML state. Otherwise, if the active class instance is in the middle of an transition, it returns the signal that was consumed in the last input statement.

```
<ACTIVE CLASS INSTANCE> -> <VARIABLE NAME>
```

Returns the value of the specified variable. If <ACTIVE CLASS INSTANCE> is a previously defined rule variable, the exists or all predicate that defined the rule variable must also include a active class type specification.

```
<RULE VARIABLE>
```

Returns the active class instance value of <RULE VARIABLE>, which must be a rule variable bound to an active class 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 active class instance value of the receiver of the signal.

```
from( <SIGNAL> )
```

Gives the active class 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 active class instances is returned.

```
depth( )
```

Gives the depth of the current system state in the behavior tree/state space.

UML Literals

```
<STATE ID>
```

The name of an UML state.

```
<ACTIVE CLASS TYPE>
```

The name of an active class type.

```
<ACTIVE CLASS INSTANCE>
```

A active class instance identifier of the format <ACTIVE CLASS TYPE>:<INSTANCE NO>, e.g. Initiator:1.

```
<SIGNAL TYPE>
```

The name of a signal type.

```
null
```

UML null active class instance value

```
env
```

Returns the value of the active class instance for the environment, the sender of all signals sent from the environment of the UML system.

```
<INTEGER LITERAL>
true
false
<REAL LITERAL>
<CHARACTER LITERAL>
<CHARSTRING LITERAL>
```

# *Customizing Tau*

The chapters that are listed under Customizing Tau describe how to develop and add new features into Telelogic Tau.

# 73

# *Customizing Telelogic Tau*

This chapter provides an introduction to the possibilities of customizing Tau. It describes how to:

- Extend the user interface
- Start Tau from the command-line
- Extend the information contents of UML models using profiles
- Access and modify the contents of UML models using a programmable interface (public APIs in various technologies)
- Write and use Tcl Add-Ins
- Extend the semantic checker with custom semantic checks
- Define custom code generators and other tools integrated with the Application Builder
- Define custom importers integrated with the Import Wizard
- Define custom extension modules integrated with the File/Folder Importer

# Introduction

Tau provides many possibilities for extending the functionality of its standard features. This ranges from simple extensions of the user interface to complete changes of the information stored in the UML models and the way it is presented to a user. The purpose of this chapter is to give an introduction to these possibilities.

### Tcl API

If you prefer a scripting solution the preferred scripting language is Tcl. You can implement a customized addition by adding scripts or programs that use the Tcl API of Tau to modify the behavior of the tool.

### COM API

If you prefer a COM-enabled language (such as Visual Basic, VB Script, JavaScript, C# etc.) Tau provides a COM API. The COM API is also a good choice when accessing Tau from common Windows applications whose scripting support often uses the COM-enabled language Visual Basic for Applications.

### C++ API

If you prefer a compiled language Tau provides a C++ API that is beneficial for complex applications where performance and cross-platform support is critical.

### Agents

Reusable functionality for customization tasks can be implemented as Agents, which in turn can be implemented using any of the above mentioned APIs. Agents let you define commonly useful behavior in the UML model, and can be used in a uniform way regardless of the technology chosen for their implementation. Agents thus act as a bridge between the different APIs and enable functionality implemented with one API to call functionality implemented with another API.

Agents are also used extensively by Tau for the customization of many features (such as queries, diagram generators, property editor customization, code generator customization etc.)

### Object model

When you use a programmable API to access or modify the contents of a UML model it is important to distinguish between the following concepts:

- The Object Model
- UML Metamodels
- UML Profiles

The Object Model is the built-in representation of the information manipulated in the UML tool set and stored in UML (`.u2`) files. The Object Model is composed of approximately 200 classes and numerous attributes and associations. When you use Tcl scripts or a program based on the COM or C++ API certain API functions require you to know and understand the details of this object model, since these API functions operate directly on the Object Model classes and their features.

Contact Tau Support for more information about the Object Model.

### Metamodel

A Metamodel is a UML package stereotyped by «metamodel». A metamodel will provide a view on the information stored in the UML models. In practise the used metamodel will determine what an end user will see in the Model View, the Property Pages and in the Model Navigator. If you want to develop an advanced customized addition for a specific application domain you might find it useful to create a new view using a new metamodel. When doing this, studying an existing metamodel, such as **TTDMetamodel**, is of great help.

### Profiles

A Profile is a UML package stereotyped by «profile». The profile mechanism will give you a convenient way to:

- Attach extra information to various UML model elements.
- Introduce new concepts to be used in the application modeling.
- Introduce new symbols that can be used in the graphical editors.

When you design profiles they must always be based on a specific metamodel. Most often the **TTDMetamodel** that is provided in the installation is your best choice so you do not need to create a new metamodel when you want to add extra information to an element in the UML model.

The relationship between the Object Model, metamodels and profiles is illustrated in Figure 270 on page 1984.

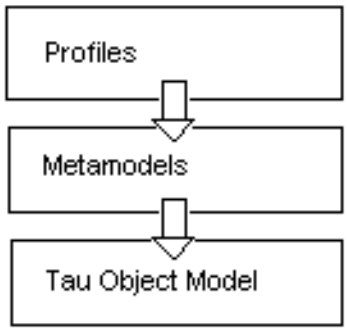



*Figure 270*

Profiles define extensions to the information stored in UML models. Profiles are defined based on metamodels. Metamodels provide a view of the underlying repository and are defined in terms of the built-in Object Model.

### Add-ins

One more concept is important to understand when designing a customized addition. This is the concept of Add-Ins. You will use an add-in as a container where you will put your profiles, metamodels, Tcl scripts, COM- or C++-based applications and other files that you want to include in a customized addition. This makes the add-in concept crucial when extending the built-in functionality and most of this chapter will be centered around different things that can be included in Add-Ins.

The Telelogic Enterprise Architect for DoDAF is an add-in that is delivered separately with templates and help information.

SysMLis an add-in that is delivered inside Tau.

### Stereotypes and attributes

A number of profiles are available in UML packages that are supplied with Tau. These packages are located in the model beneath the node named **Library**.

By applying stereotypes from these profiles, and providing values for the attributes they contain, the settings that controls various tool features can be changed. One example out of many is the attributes that control the code generation settings, model-to file mapping scheme and adoption of the conventions used by the host computer, operating system, make and compiler.

# Launch from command line

### Launching Tau with a Tcl script as argument

Tau can be started from the command-line with a Tcl script as argument. In addition, parameters can be passed to the script. When Tau is started it will run the provided script.

**Example 618: Start Tau from the command-line**

Windows

```
VCS.EXE -Script u2merge.tcl forceVersion1
reviewDifferencesAlways suppressSetupNever false
C:/work/version1/project.ttp C:/work/version1/project.ttp
C:/work/version2/project.ttp [C:/work/ancestor/project.ttp
[C:/work/version2ancestor/project.ttp]]
```

UNIX

```
tau -script u2merge.tcl forceVersion1
reviewDifferencesAlways suppressSetupNever false
/home/user/version1/project.ttp
/home/user/version1/project.ttp
/home/user/version2/project.ttp
[/home/user/ancestor/project.ttp
[/home/user/version2ancestor/project.ttp]]
```

This will start Tau with the u2merge.tcl script. The parameters following the script's name are then passed to the script.

For an example of how to parse the parameters in Tcl, please refer to:

```
$TAU_INSTALLATION/etc/u2merge.tcl.
```

**Note**

*The –Script option must be the last option on the command-line and cannot be mixed with the following feature – loading a *.ttw or *.ttp file. The script itself must do the loading using the Tcl API.*

### Launching Tau with a Workspace (*.ttw) or Project (*.ttp)

Tau can be started with a specified Workspace or Project.

**Example 619: Start Tau with a Project**

Windows

```
VCS.EXE c:\MyProject\Version1\MyProject.ttp
```

UNIX

```
tau /home/me/MyProject/Version1/MyProject.ttp
```

### Suppressing the splash screen

To suppress the Tau splash screen on Windows:

```
VCS.EXE -SuppressSplashScreen
```

To suppress the Tau splash screen on UNIX:

```
tau -SuppressSplashScreen
```

**Note**

*The described command-line parameters are case-insensitive.*

# Add-Ins

## Application areas for add-ins

Add-ins provide the basic mechanism that you can use to package an own-developed set of features. There are a number of built-in add-ins in Telelogic Tau that are used to enable features required to support e.g. generation of application code.

## Activating add-ins

Add-ins are activated and de-activated from the Add-ins tab, which is found in the **Tools** menu, select Customize.

The built-in add-ins are found in the '`addins`' subdirectory of the installation directory, typically:

```
<installation path>/addins/
```

Each add-in is placed in its own subdirectory with the following contents:

- An '`<addin name>.mod`' file
- A subdirectory called `bin` containing the executable files or shared libraries that are part of the add-in.
- A subdirectory called 'Script' containing Tcl script files
- A subdirectory called '`Etc`' containing other files, often UML models with profile definitions.

The '`<addin name>.mod`' file defines various properties of the add-in and also lists the other files that are contained in the add-in.

User add-ins are located in the user specific directory:

```
$HOME/Telelogic/Tau 2/addins
```

The directory structure for user add-ins is the same as for built-in add-ins.

New add-ins are placed in the user add-ins directory. When you Create a new addin module through Tau this directory is created if it does not already exist.

User defined add-ins can be stored in locations defined by the environment variable`TAU_USER_ADDINS_DIR`. When this variable is set new add-ins will be stored in this location.

```
%TAU_USER_ADDINS_DIR%/addins
```

Add-ins can also be placed in the following locations available for add-ins shared by a group of users.

Team add-ins are located in the default directory:

```
$HOME/Telelogic/Shared/TeamAddins
```

The team add-ins location can be overridden with the environment variable TAU_TEAM_ADDINS_DIR to this location.

```
%TAU_TEAM_ADDINS_DIR%/
```

Company add-ins are located in the default directory:

```
$HOME/Telelogic/Shared/CompanyAddins
```

The company add-ins location can be overridden with the environment variable TAU_COMPANY_ADDINS_DIR to this location.

```
%TAU_COMPANY_ADDINS_DIR%/
```

The predefined URN mappings `u2useraddins, u2teamaddins` and `u2companyaddins` exists. They are directed to the add-in directories and supports both the cases when the environment variables are set and when they are not. These URN mappings can not be overridden.

# Contents and structure of an add-in

In the example below is discussed the contents of an add-in. You can create your own add-ins, by creating the necessary file set and placing it in the add-ins directory.

### Create a new addin module

To set-up an add-in through Tau you go to the **Tools** menu, select **Customize**, then point to the Add-ins tab. Click **Create** and the **"Create a new addin module"** dialog will open. Entering values in the different tabs will result in a `.mod` file for your add-in.

**Example 620: A module file for an add-in 'MyAddin'** ——————————

An example of a `.mod` file for an add-in '`MyAddin`' is the following:

```
[MyAddin]
"scope"="PROJECT"
"version"="1.0"
```

```
"longname"="My Addin"
"description"="Gives an example of a .mod file."
"product"="u2"
[MyAddin/Bin]
"listBin"=""
[MyAddin/Script]
"listScript"="MyAddinScript.tcl"
[MyAddin/Etc]
"listEtc"="MyAddinProfile.u2"
```

- In Example 620 on page 1988 the string "`MyAddin`" corresponds to the Identifier field in the **"Create a new addin module"** dialog. This is the string that will be shown in the Add-ins tab to check if the add-in is activated.
- The `"scope"` property defines when the add-in is loaded.
  - If the value is `"PROJECT"` the add-in is only loaded when a project is loaded.
  - If the value is `"GENERAL"` the add-in is loaded even if Tau is started without any project.
- `"version"`, `"longname"` and `"description"` are used in various dialogs to give a user information about the add-in. `"longname"` corresponds to the **Name** field in the **"Create a new addin module"** dialog. The text in "`description`" is shown in the dialog where a user can switch on/off different add-ins.
- `"product"` specifies what tool the add-in is intended for. Add-ins intended for UML projects should for compatibility reasons use the value `"u2"` as product. If `"scope"` is set to "`GENERAL`", "`product`" should be set to "". When editing the value through the **"Create a new addin module"** dialog, the choice Tau will set product to `"u2"`.
- The `"listBin"` property is usually not necessary in a customized addition, you can leave this empty. The "`listBin`" property corresponds to the files listed in the **Binaries** tab.
- The `"listScript"` property corresponds to the files listed in the **Scripts** tab. The Tcl scripts mentioned in this list will be executed when a user switches on the add-in. You must enter the names of all Tcl scripts that you want automatically invoked here.
- The "`listEtc`" property corresponds to the files listed in the **Other files** tab. The "`listEtc`" is used for *.u2 files which are to be loaded when loading a project which has the add-in selected. The referenced .u2 files will be loaded as libraries, and their contents thus appear under the Library folder in the Model View.

Once you have placed your `.mod` file in the appropriate subdirectory, your add-in will show up in the list of available add-ins found in the **Customize** dialog Add-Ins tab.

**Note**

*To insert multiple file names when you edit the* `.mod` *file with a text editor, the names must be separated by semicolons.*

### Add-in Tcl script execution

Every Add-in that wants to add functionality to Tau (i.e. not just load a UML library or profile) must have a Tcl script which will be executed when the Add-in is activated in Tau. Although all Add-in functionality can be implemented in Tcl using the Tcl API, it is also possible to do some part of the implementation using the COM API or the C++ API and then invoke this functionality from the Tcl script using the Tcl command u2::InvokeAgent.

When the Add-in is deactivated in Tau a Tcl procedure called `BeforeUnload` will be executed if present. This procedure can be used to perform finalization tasks, such as unloading libraries or profiles that were loaded when the Add-in was activated.

# Customizing the User Interface

## General

The simplest way to extend the user interface of Tau is to use Add-Ins with Tcl scripts.

## Writing the add-in

The following shows how to extend "`MyAddin`", discussed in section "Add-Ins" on page 1987, with a simple script. The script adds a menu called **My-Menu** with one menu entry **MyCommand** that upon invocation displays a message box.

Create a Tcl file called `MyAddinScript.tcl` in the `script` subdirectory of your add-in. Then copy and paste the following text to `MyAddin-Script.tcl`:

```
std::Output "MyAddin loading ... "

package require commands
package require dialogs

proc OnMyCommand {cmd} {
  std::MessageDialog -name "My message dialog" -message
"This is my message!"
}

proc OnTrue {e} {
  return 1
}

proc Init {} {
  std::AddCommand -variable cmdMyCommand -name "My
command" -statusmessage "My command status msg" -tooltip
"My tooltip" -imagefile "" -OnActivateCommand
OnMyCommand -OnEnableCommand OnTrue
  std::AddMenu -variable MyMenu -commands { cmdMyCommand
} -path { &MyMenu } -position {after &Tools}
}

Init

std::Output "Done.\n";
```

## Loading the add-in

If you activate the add-in through the "**Tools->Customize**" dialog, an extra menu will show up. Select the "**My Command**" entry in this menu and a message box with the specified text will pop up.

**Note**

*Remember to load a UML project before activating the add-in, since the add-in was defined with "PROJECT" scope!*

**See also**

"User Interface Add-in Specific Commands" on page 2214 in Chapter 77, *Tcl API*

"General Purpose Commands" on page 2188 in Chapter 77, *Tcl API*.

# Profiles

## Application areas for profiles

Profiles are used to extend the UML language and can:

- Add more information to existing UML concepts.
- Introduce new concepts used together with the built-in UML concepts.
- Introduce new graphical symbols used together with the built-in symbols in the UML diagrams.
- Introduce customized behavior to Tau in the form of Agents.

A profile is defined in UML as a «profile» stereotyped package. To create a profile you thus create a usual package and then apply the predefined «profile» stereotype to the package. Inside the «profile» package stereotypes are used to define the new concepts and the attributes of stereotypes are used to define extra information to be stored in the UML models.

**Note**

*All top-level definitions placed in a profile are visible without qualification from all places in the user model. This is because when a profile is loaded a top-level <<access>> dependency is automatically added for it.*

## Creating a profile

The following section shows how to define a profile that adds two pieces of information to each class in the UML models:

- The author of the class (a character string), and
- The creation date of the class (also a character string).

To define this profile you only need to do the following:

1. Create a «profile» package. Put the package in a separate file. This is not strictly needed but it simplifies the application of the profile.
2. Add a class diagram to the «profile» package.
3. Create a new stereotype in the diagram. Call it for example “ClassInfo”.

4. In the Model View: Open the **TTDMetamodel** package that you can find beneath the **Library** node. Find the class called “Class” in TTDMetamodel and drag it into the class diagram, next to the “ClassInfo” stereotype you created above. Notice that “Class” is stereotyped as a «metaclass», more about this later.
5. Define two attributes on “ClassInfo”: “Author” and “CreatedDate”, both typed as Charstring.
6. Create an extension line from “ClassInfo” stereotype to the “Class” class.
7. Add “1” in the text field association with the extension line.
8. Done! The result should look like in Figure 271 on page 1994

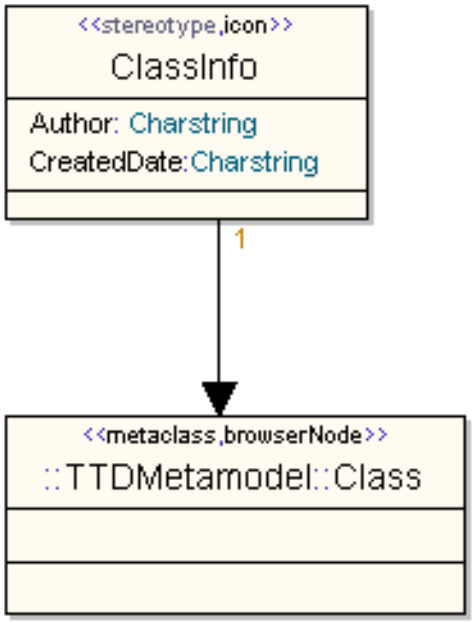



*Figure 271*

You have now created a profile that when applied will extend all classes in a model with information about author and creation date. From a user’s point of view the author and creation date will appear as two entries in a ‘`ClassInfo`’ page among the other property pages for his classes.

## Testing the profile

To test the profile create another class in your diagram and open the Property pages for the class. You should now find your newly created class containing the author and creation date fields in the ClassInfo property page.

## Deploying the profile for use

The next part of the profile design is to make it possible for other users to benefit from your new profile. The recommended way to do this is to use the add-in mechanism. To accomplish this you need to do three things:

- Add the profile in the `Etc` subdirectory in an add-in
- Add a `LoadLibrary` command in one of the Tcl scripts that are part of the add-in. Remember that the Tcl script must be mentioned in the `"listScript"` property.
- Mention the file containing the profile in the `.mod` file of the add-in. This is strictly speaking not necessary, but is recommended for compatibility with future versions of the tool.

Assuming that the profile is stored in a file called "`MyAddinProfile.u2`" in the `MyAddin/Etc` directory the following code can be added to a Tcl script in the add-in to load the profile:

```
set ProfilePath [std::GetUserAddinsDirectory]/MyAddin/Etc/MyAddinProfile.u2
std::Output "Loading MyAddinProfile.\n"
u2::LoadLibrary $ProfilePath
```

As an alternative to using a Tcl script for loading your profile, you may use the "`listEtc`" field of the `.mod` file to specify which `.u2` files to load.

You can also add a "urn" reference to a `.mod` file:

```
[MyAddin/Etc]
"listEtc"="urn:u2useraddins:MyAddin/Etc/MyAddinProfile.u2"
```

### More about profiles

When the add-in is switched on you will find your new profile among the other in the Library node in the Model View.

This is a simple example of how to use a profile to add more information to existing UML classes. It is also possible to use a stereotype as a classification means to refine the semantics of a general UML concept. For example consider to introduce a new concept into UML, "Documentation artifact", and make it possible to mark an artifact as a documentation artifact. To do this you define a profile as above, but call the stereotype "Documentation" and make it extend the "Artifact" Metaclass from TTDMetamodel, instead of the "Class" metaclass. Furthermore write "`0..1`" instead of "`1`" on the extension line. The result should look like in Figure 272 on page 1996.

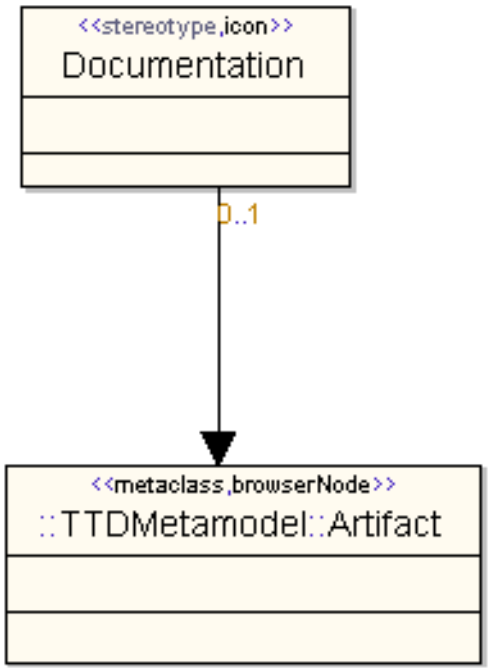



*Figure 272:*

It is also possible to define a special icon that can be used in diagrams to show «Document» artifacts. To accomplish this you need to apply the predefined stereotype **TTDStereotypeDetails::icon** to the «Document» stereotype that you created. When you have done this you can define the path to a file containing a bitmap that will be used for all «Document» icons in the Icon property page for the «Document» stereotype. The path name given in the "Icon File" property is an absolute path, but you can also use `$INSTALLPATH` to refer to the installation directory.

When you have done this a user can show your bitmap instead of the usual artifact symbol in the diagrams, by switching on 'Icon Mode" in the shortcut menu for artifact symbols stereotyped by «Document».

For advanced users it is also possible to customize how the stereotype attributes are shown in the Property pages, essentially defining the editing controls used to display and edit the values. The details on how to accomplish this is defined in a built-in Profile, the TTDExtensionManagement Profile profile. This profile defines a number of stereotypes that can be applied to stereotype definitions in «profile» packages. You can find out more about these possibilities by investigating the `TTDExtensionManagement` profile itself. It is available in the installation. Just open any UML model and you will find it in the **Library** node.

**See also**

"Customizing the Properties Editor" on page 98

"Metamodel" on page 386

# Model Access

## Application areas for model access

The Add-Ins discussed in this section have only changed the appearance of the tool, either by adding menus and dialogs or by introducing stereotypes and graphical icons. However, no real new functionality has been added. Typically useful features will require access to the currently loaded model, either a read-only access or read-write access to modify the model. After all, Tau is a modeling tool and the purpose of the environment is to create UML models of various kinds.

## Adding model access functionality

Tau provides APIs to add new functionality that can access the model loaded in the tool. These APIs are available both through scripting (Tcl API and COM API) and from compiled code (C++ API and COM API). The decision of which to use is a trade-off between the development effort required and of the execution performance required. Scripting can be beneficial during prototype work and when writing small extensions that are executed interactively and that does not require too extensive computations. Some examples are simple report generators and small transformation scripts. When the application is more advanced a compiled tool coded using the C++ API or the COM API is more suitable. In particular since the performance will be substantially better, but also because of the usually better debugging possibilities. A code generator that produces code in a programming language based on a UML model is a typical example.

## Using the Tcl API

The following illustrates how to use the Tcl API, by further extending the script in `MyAddin`. The objective is to change the command in the "`MyAddin`" menu to compute some metrics on how the Author attribute for classes is used. More precisely, count the number of classes in the loaded module and check how many of them has an Author defined.

The following Tcl script will accomplish this:

```
set NoOfClasses 0
set NoOfAuthorClasses 0

proc CheckClass {e} {
  global NoOfClasses
```

```
  global NoOfAuthorClasses

  if { [u2::IsKindOf $e Class] } {
    set NoOfClasses [expr $NoOfClasses + 1]
    set Author [u2::GetTaggedValue $e "ClassInfo (. Author .)"]
    if {$Author != 0} {
      set NoOfAuthorClasses [expr $NoOfAuthorClasses + 1]
    }
  }
}
proc OnMyCommand {cmd} {
   global $std::activeproject
   global NoOfClasses
   global NoOfAuthorClasses
   set ActiveModel [std::GetModels -kind U2 -project
$std::activeproject]
   u2::MetaVisit $ActiveModel CheckClass
   set str "The model contains $NoOfClasses classes, of which
$NoOfAuthorClasses have an author."
   std::MessageDialog -name "Author report" -message $str
 }
```

If you replace the `OnMyCommand` procedure in the `MyAddinScript.tcl` from section "Customizing the User Interface" on page 1991 with the code above, then the metrics dialog will pop up if a user selects the "**My command**" alternative in the "**My Menu**" menu.

The main Tcl commands used in the script are the `u2::MetaVisit` command to simply traverse the model, and the `u2::GetTaggedValue` to extract the Author of the classes you find.

# Adding Semantic Checks

## Application areas for semantic checks

When defining Add-Ins like code generators there is a special kind of extension that is particularly useful: to define specific semantic checks that are performed together with the built-in semantic checks when the user orders a Build or Check command from the Build menu or the tool-bar.

One particularly useful application for this feature is to check that the information specified in profiles is used correctly.

**Example 621: Checking the Author attribute is given a value**

The code below shows how to test if the Author attribute has been given a value for all classes in the current model. This is checked by adding the following Tcl code to the `MyAddinScript.tcl` script introduced in the previous sections:

```
proc SemCheckClass {e} {
  set Author [u2::GetTaggedValue $e  "ClassInfo (.
Author .)"]
  if {$Author == 0} {
    u2::SemMessage werror "No Author" $e
  }
}

u2::CreateSemGroup "/" "MyChecks"

u2::CreateSemRule "/MyChecks" "CheckAuthor" Class 30
"SemCheckClass"
```

This script adds a new group of semantic checks called "`MyChecks`" and then adds a rule "`CheckAuthor`" to this group. The actual semantic check is performed by the procedure `SemCheckClass` that automatically will be called for all classes in the model.

**Note**

*When the add-in gets deactivated, it is important to delete semantic rules and/or groups that were added upon add-in activation. Failure to do so will make the semantic checker attempt to call your Tcl script when it no longer is available. This can lead to a crash. Use the BeforeUnload procedure to delete added rules and/or groups:*

```
proc BeforeUnload {} {
  u2::DeleteSemEntity "/" "MyChecks"
```

```
}
```

It is also possible to introduce new semantic checks using Agents which trigger on Semantic checker events. This makes it possible to implement the checks not only using Tcl scripts, but also using the C++ API or the COM API.

**See also**

"Semantic Checker Commands" on page 2235 in Chapter 77, *Tcl API*.

# Adding Code Generators

## General

A code generator is divided into several components:

- A profile package that defines a build stereotype.
- The actual code generator, commonly in the form of an executable or agents.
- Additional run-time libraries to be used with the generated code.

In most cases these are all bundled into an Add-in.

## Build Stereotype

This section contains an example of a fictional C# code generator that defines two build operations.

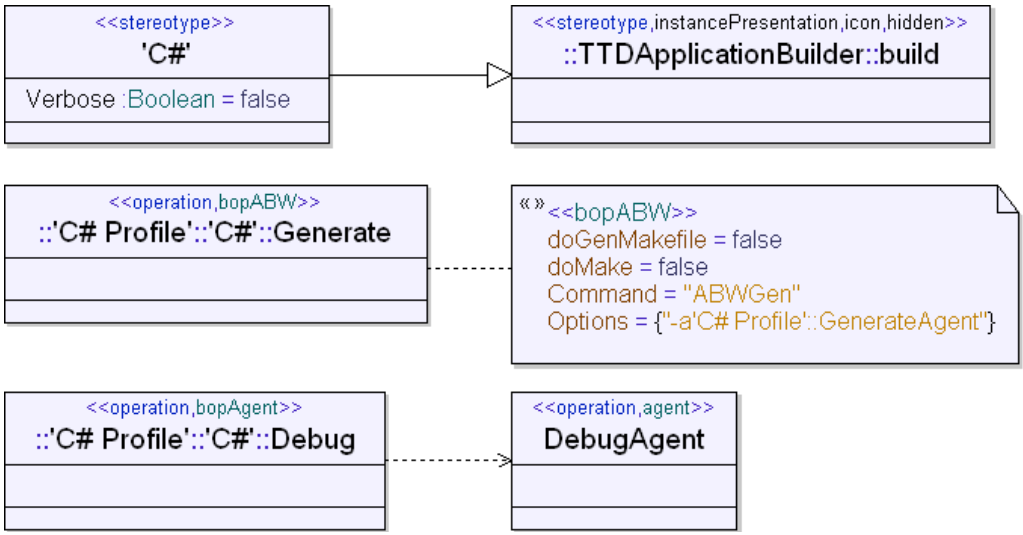



*Figure 273:*

The build stereotype «C#» inherits the predefined «build» stereotype and defines the attribute "Verbose" (which when enabled causes the code generator to emit additional messages). «C#» also defines the two build operations; “Generate” which generates C# code from the model and “Debug” that connects to an external debugger.

A build artifact that is stereotyped by the «C#» stereotype will show the two choices “Generate” and “Debug” in the context menu if the user right-clicks on it.

The operation "‘C#’::Generate" is stereotyped by «bopABW». This specifies that when the operation is selected in the build artifact context menu an executable ("ABWGen.exe") will be launched using the command line argument "-a’C#Profile’::GenerateAgent".

The operation "‘C#’::Debug" is stereotyped using «bopAgent» and has a dependency to the agent "DebugAgent". This specifies that when the operation is selected, the Tau agent is invoked.

## ABWGen

The executable ABWGen is used to execute agents in an external process. It is intended to be used in conjunction with build operations defined using the stereotype «bopABW».

**Note**

*Since ABWGen runs in an external process the agents it invokes are non-interactive agents. Agents implemented in TCL can therefore not be used with ABWGen.*

### Syntax

```
ABWGen [options] [input-file]
```

`input-file`
If input-file is given, that is used for reading the build parameters instead of stdin. This argument is mainly intended to be used in debugging.

`options`

`-v`
Enables verbose mode. Additional messages are printed.

`-p<profile-file-name>`
Name of a u2 file to load as a library. This library can contain agents.

`-E<event-guid>`
Event to emit. This is a tool event which can trigger one or many agents.

`-e<event-name>`
Event to emit. The event is specified using a fully qualified name. This is a tool event which can trigger one or many agents.

`-A<agent-guid>`
Agent to call.

`-a<agent-name>`
Agent to call. The agent is specified using a fully qualified name.

*Except for '-v', each option can be used any number of times.*

*If no event or agent is specified the default event with the GUID "@TTDAB@Generate" is used.*

### Execution

The program executes the agents and events in the following order:

1. Agent parameters are formed:

   ```
   error list : u2::ITtdMessageList
   selection  : u2::ITtdEntity[*]
   ```

   The agent context is always the build artifact.

   The error list can be used by invoked agents to report messages to the Build tab.
   The selection list corresponds to elements that are selected in the Model View when invoking the build operation.
2. "Before processing" is called for each event given using the "-e" and "-E" options in the order they appear on the command line.
3. Each of the agents listed using the "-a" and "-A" options is called in the order they appear on the command line.
4. "After processing" is called for each event given using the "-e" and "-E" options in the reversed order they appear on the command line.

## See also

"Agent Invocation Triggered by a Tool Event" on page 2027

# Adding Importers

An importer is a utility which creates a UML model (or parts of a model) based on some other source of information, which typically is external to Tau. Importers are used for different purposes, for example

- to analyze or visualize information using UML models and diagrams
- to import external software interfaces, to be able to access them from Tau
- to migrate source code or models developed in different tools and languages to Tau

The user-interface in Tau through which importers are run is called the **Import Wizard**. It consists of a set of dialogs which allow you select which importer to run, to specify various options for the selected importer, and finally to perform the actual import.

To open the Import Wizard:

1. Select an entity in the Model View. Some importers may use the selected entity as the model context where new model elements are created. Others may always create new elements at a predetermined location in the model.
2. From the **File** menu select **Import...**.

Tau ships with a set of built-in importers which support importing UML models from a wide variety of sources. Depending on which add-ins that are activated different importers will be available in the Import Wizard.

## Creating a New Importer

You can define your own custom importer, integrated with the Import Wizard, by defining a class which realizes the `ImportWizard` interface.

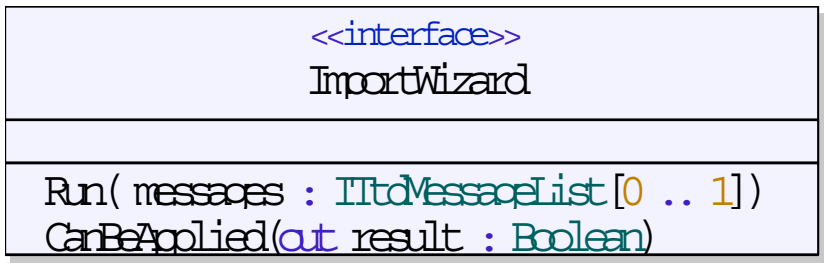



*Figure 274 The ImportWizard interface*

This interface is defined in the `TTDImporters` profile. In that profile you can also find those classes realizing this interface which correspond to the built-in importers in Tau.

The steps for creating a new importer are:

1. Create a class which realizes the `ImportWizard` interface. You typically place the class in a separate package and save it in a separate file, which can be loaded as a library using an add-in (see Creating a profile for more information).
2. Give the class a descriptive name. The name you choose will appear in the list of importers in the Import Wizard.
3. You may apply the <<icon>> stereotype on the class in order to specify an icon for the importer. The icon will appear in the list of importers in the Import Wizard.
4. Give the class a comment which describes what it does. The comment text will be displayed when selecting the importer in the Import Wizard.
5. Implement the `ImportWizard` interface, by defining the operations `Run` and `CanBeApplied` as Agents in your class. See Run and CanBeApplied below for more information about these operations.

### CanBeApplied

The model context of this agent is the context of the importer, i.e. the entity that has been selected when starting the Import Wizard. If no entity was selected, the context will be the top-level model node.

The agent should assign a boolean value to the ‘result’ out parameter which tells the Import Wizard whether or not the importer can be run. If it can be run, it should set the parameter to ‘true’, otherwise to ‘false’.

CanBeApplied makes it possible to implement a dynamic condition for when an importer can be used or not. For example, the importer may require a particular kind of context element to be selected, or it may require the presence of certain libraries.

### Run

The model context of this agent is the context of the importer, i.e. the entity that has been selected when starting the Import Wizard. If no entity was selected, the context will be the top-level model node.

This agent is responsible for implementing the user interface of the importer. It will be called when the importer is selected, and **OK** is pressed in the Import Wizard. The agent can either open a GUI (for example a dialog) inside Tau, or it can launch an external program which provides the user interface.

If the agent needs to report any messages while the user is working with the GUI it can use the 'messages' parameter, if provided. However, often it is better to report such messages directly in the GUI.

### Import

Some importer classes define an `Import` agent which performs the actual import operation. Parameters to this agent then correspond to importer options. Note that `Import` is not part of the `ImportWizard` interface, since different importers have different options.

If you choose to define an `Import` agent for your importer class the implementation of the Run agent typically uses it when performing the import from the user interface.

One benefit from exposing the import operation as a separate agent, instead of embedding it inside Run, is that the importer can be programmatically accessed from the APIs (using the InvokeAgent API method).

## An Example

As an example of how to add a custom importer let's create a simple directory importer using the Tcl API. This importer is a simplified version of the built-in File/Folder Importer.

First we add a class that realizes the `ImportWizard` interface, and we give it a name and a comment. We define the operations `Run`, `CanBeApplied` and `Import` in the class, and turn them into Tcl agents by selecting the command **Utilities - Turn into Tcl agent** in the context menu of each operation.

The result should look something like this:

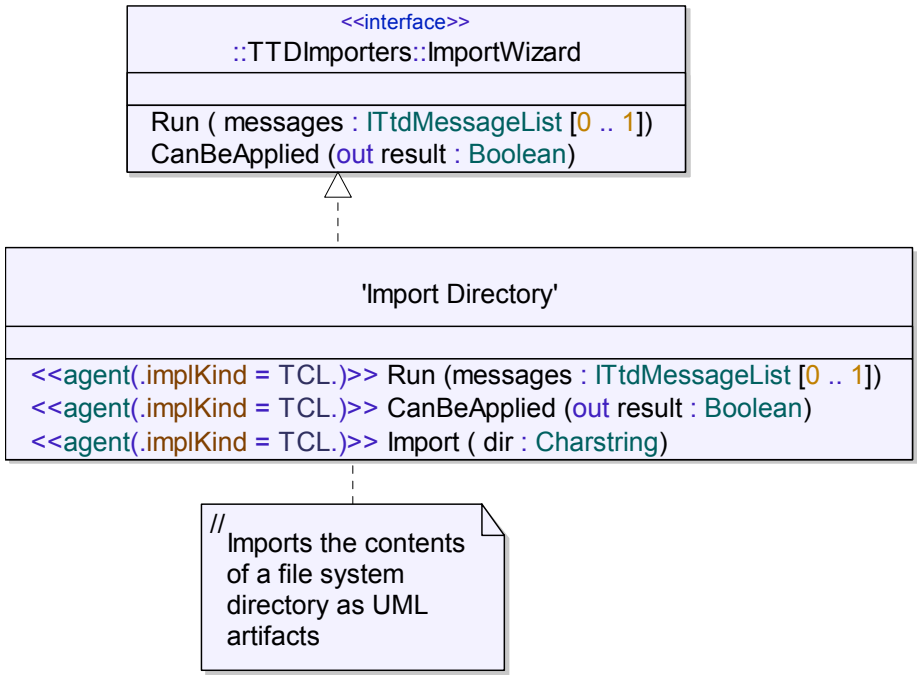



*Figure 275 Definition of a custom importer*

We continue by implementing the `CanBeApplied` agent. This simple importer shall always be available so we just set the 'result' out parameter to true:

```
[[
proc CanBeApplied { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  lset ap 0 1
}
]]
```

Note that 'result' is the first (and only) agent parameter (at index 0) and true is represented as 1 in Tcl.

Now we are ready to open the Import Wizard to make sure we will find our new importer in the list of available importers.

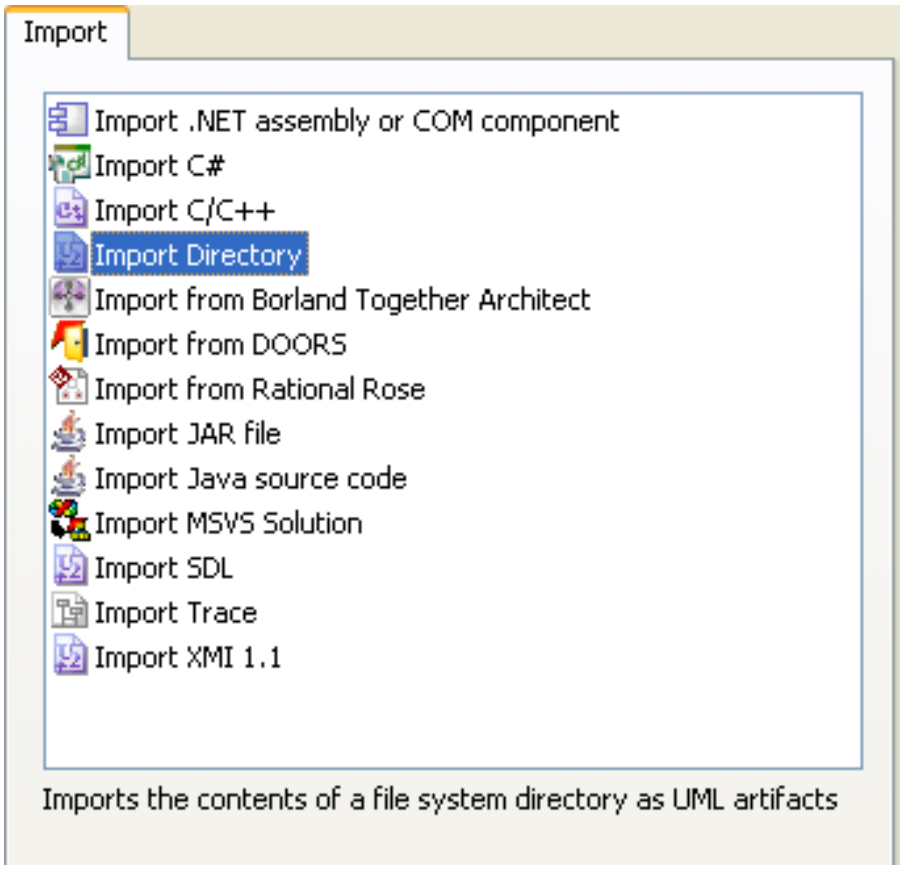



*Figure 276 Import wizard including a custom importer*

As we can see we get a default icon for our importer. We could change this by applying the <<icon>> stereotype if we want to.

If we try to run our importer now nothing will happen. That is because we did not yet implement the `Run` agent. Let's do that now:

```
[[
proc Run { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  set dir [std::DirectoryDialog]
  set model [u2::GetModel $context]
  set importAgent [u2::FindByName $model "'Import
Directory'::Import"]
  set p [list $dir]
  u2::InvokeAgent $model $importAgent $model p
}
]]
```

As you can see the `Run` agent, which implements the GUI of our importer, just opens a standard dialog (using Model Commands) which allows the user to select a directory in the file system. It then invokes the `Import` agent

passing the selected directory path as an agent parameter. Note that we have assumed that our class `'Import Directory'` is placed in a <<profile>> package, so that it is accessible without including the name of the package in the qualifier passed to u2::FindByName. See the note in Application areas for profiles for more information about profile packages, and their impact on the accessibility of contained definitions.

As an alternative we could have used u2::FindByGuid for finding the `Import` agent.

Now, the only thing that remains is to write the implementation of the `Import` agent:

```
[[
proc Import { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  set dir [lindex $ap 0]
  set model [u2::GetModel $context]
  set pkg [u2::Create $model "Package"]
  u2::SetValue $pkg "Name" $dir
  set files [glob -directory $dir "*.*"]
  foreach f $files {
    set a [u2::Create $pkg "Artifact"]
    regsub -all {\\} $f {/} path
    u2::SetValue $a "Name" $path
    set ie [u2::Parse $model "file (. path =
\"$path\".)" -parseAs Expression]
    u2::SetEntity $a "StereotypeInstance" $ie
  }
}
]]
```

As you can see the `Import` agent creates a top-level package and sets its name to the selected directory. It then reads the contents of that directory on the file system using the Tcl `glob` command. For each file that is found a corresponding file artifact is created. Note that we must substitute backslashes in the path to make sure u2::Parse receives a text in valid U2 syntax.

Now we can test our new importer by opening the Import Wizard, selecting the 'Import Directory' importer, and browse for a directory on the file system. Here is an example of what the resulting model could look like:

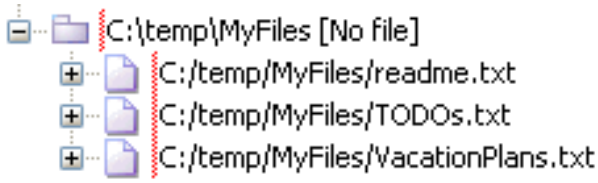



*Figure 277 Result from importing a directory containing 3 text files*

We can open the files in the Tau text editor by double clicking on the file artifacts.

## XML Based Importers

A common kind of importers are those that read XML files as input. In order to facilitate the implementation of such custom XML based importers, Tau has a small XML framework which among other things supports parsing XML into a UML representation. See Importing XML Documents for more information.

## Importers Generating Diagrams

Often an importer provides the possibility to visualize imported elements in UML diagrams. If you want to support this with your custom importer you should do it by means of invoking diagram generators. See Generate Diagram to learn more about diagram generators.

A number of built-in diagram generators are defined in the **TTDDiagramAgents** library. They can be used from an importer in order to generate diagrams. See Invoking Diagram Generators Programmatically for more details.

# Adding Diagram Generators

It is possible to create your own diagram generators in order to generate custom diagrams. Diagram generators are implemented by means of Agents which use the Tau APIs for creating the diagram.

To create a new diagram generator:

1. Select an element of the kind that you want to generate a diagram for. An element of the selected kind will be passed as the context entity for your diagram generator agent.
2. Select **Create new diagram generator...** in the **Generate Diagram** context menu.
3. In the dialog specify a name and description of the new diagram generator, as well as a location in the model where it shall be stored. It can be a good idea to put all diagram generators in a common place, for example in a profile package stored in a separate .u2 file. Thereby you can include and use your saved diagram generators in multiple projects.
4. If you want to use the Tcl API for implementing your diagram generator agent, check the **Generate Tcl stub implementation** checkbox.
5. Press **OK** to create the diagram generator.

A diagram generator is represented in the model as an ordinary agent with the <<diagramGenerator>> stereotype applied.

When you have created a diagram generator in the model, it immediately becomes available in the list of diagram generators applicable on entities of the selected kind. If you want your diagram generator to apply for more kinds of entities, you should create additional <<agent command>> dependencies for the new diagram generator agent. By editing these <<agent command>> stereotype instances in the Properties Editor it is also possible to specify default values for Diagram Generation Parameters.See Agent Commands for more information about the <<agent command>> stereotype.

## Standard Diagram Generator Parameters

In addition to the custom diagram generator parameters which you may define (using ordinary agent parameters) the diagram generator framework will pass some standard parameters which must be correctly handled by the diagram generator agent. In total, a diagram generator receives the following parameters when invoked:

1) inout diagram : ITtdEntity

2) inout synthesizedEntities : ITtdEntity[*]

3...N) <custom diagram generator specific parameters>

If the ‘diagram’ parameter is NULL the diagram generator agent is responsible for creating a new diagram. It can place it wherever it likes, but a common choice is to place it below the context element. Note that ‘diagram’ is an inout parameter, which means that the agent must set it up to refer to the newly created diagram before returning control to the framework.

If the ‘diagram’ parameter is not NULL it will refer to an already generated diagram. The diagram generator agent is then responsible for regenerating that diagram. The diagram will already be empty (that is taken care of by the framework) which simplifies the implementation of the diagram generator agent.

The ‘synthesizedEntities’ parameter is a list of entities which may be created by the diagram generator, in addition to the generated diagram. This allows a diagram generator to visualize information which, at the time of invocation, doesn’t exist explicitly in the model. For example, a diagram generator may compute certain relationships between model entities, and represent these as dependencies in the model. By adding the dependencies to the ‘synthesizedEntities’ list the framework can keep track of such synthesized entities. When a diagram generator agent is invoked in order to regenerate a previously generated diagram ‘synthesizedEntities’ will contain the entities which were created when generating the diagram. They can thus be deleted by the agent, before computing the new information and regenerating the diagram.

A diagram generator which will synthesize additional entities should set the boolean attribute ‘synthesizesAdditionalEntities’ in the <<diagramGenerator>> stereotype to true.

## Implementing a Diagram Generator Agent

A diagram generator agent may use any mechanism available in the Tau APIs for creating and populating the diagram. However, since certain typical tasks are common for many diagram generators, and some of these tasks are non-trivial to implement, Tau provides a dedicated library with utility agents which can be used by a diagram generator agent implementation.

The library is called **TTDDiagramAgents** and contains utility agents which among other things facilitates

- Adding symbols and lines to a class diagram.
- Adding symbols and lines to a sequence diagram.
- Setting an appropriate default size on symbols.
- Automatically route lines in a diagram to make them look nice.
- Automatically layout symbols in a diagram according to different layout algorithms.

For more detailed information about these agents see their documentation in the diagrams available in the **TTDDiagramAgents** library.

### Typical implementation steps

A typical diagram generator agent performs the following steps in its implementation:

1. Create a new diagram of an appropriate kind somewhere in the model. This step is skipped in the case of regeneration of an existing diagram.
2. Collect information to visualize in the diagram. This can for example be done by executing Queries. If the information to visualize is not explicitly represented in the model the agent may compute the information and represent it by means of model entities, for example dependencies.
3. Use utility agents of the **TTDDiagramAgents** library to populate the diagram with symbols and lines, to compute appropriate sizes on the symbols, to route lines and to layout symbols.

## Example

The Tau installation contains an example of a few custom diagram generators implemented in Tcl. To open this example select **File - New - Samples** and choose **umlAgentCallGraph**.

## Invoking Diagram Generators Programmatically

A diagram generator can be invoked programmatically, but should not be directly invoked using the InvokeAgent API method. Instead you should use the agent **DiagramGeneratorFramework** which is the entry point for accessing the diagram generator framework functionality.

**Example 622: Invoking a diagram generator programmatically**

The following Tcl script invokes the InheritanceView diagram generator in order to generate an inheritance diagram for the selected element (which for example can be a class):

```
set curProject [std::GetActiveProject]
set model [std::GetModels -kind U2 -project $curProject]
set a [u2::FindByGuid $model
"@TTDDiagramAgents@DiagramGeneratorFramework"]
set dg [u2::FindByGuid $model
"@TTDDiagramAgents@InheritanceView"]
set p [list $dg "false"]
u2::InvokeAgent $model $a [std::GetSelection] p
```

The first parameter to the DiagramGeneratorFramework agent is the diagram generator agent to invoke. The second parameter is the parameters to pass to the diagram generator.

---

There is also an agent for regenerating a diagram, called **DiagramRegenerationFramework**. It takes the diagram to regenerate as model context, and expects no parameters.

**Example 623: Regenerating a diagram programmatically**

Regenerating the diagram generated in Example 622 on page 2015 (assuming that the generated diagram is selected).

```
set a [u2::FindByGuid $model
"@TTDDiagramAgents@DiagramRegenerationFramework"]
u2::InvokeAgent $model $a [std::GetSelection]
```

---

# Adding Extension Modules for the File/Folder Importer

The File/Folder Importer is a tool for importing files and/or folders to get a model representation of them in Tau. When performing the import it is possible to specify an extension module which can further process imported files/folders in order to augment the model with additional information. For example, an extension module can analyze imported files and create dependencies between the corresponding file artifacts in order to illustrate some logical dependency between the files.

You can define your own custom extension module, integrated with the File/Folder Importer, by defining a class which inherits the `ExtensionModule` class located in the `Import Files/Folders` importer class. You find this class in the `TTDImporters` profile.

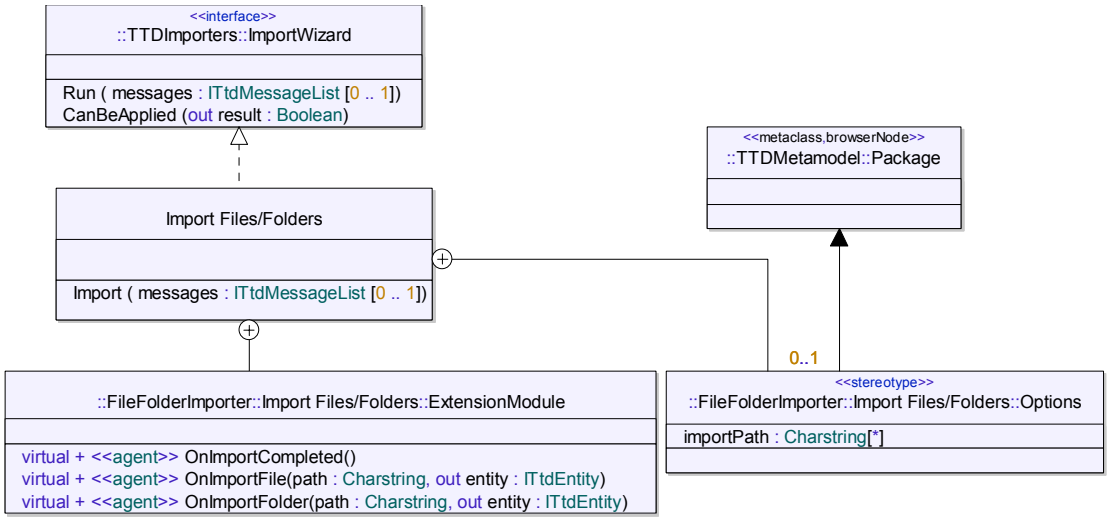



*Figure 278 Classes and stereotypes related to the File/Folder Importer*

As can be seen in the diagram the `ExtensionModule` class defines some virtual agent operations. These agents implement the default behaviour of the File/Folder importer. By overriding some or all of these operations by defining agents in the new class, you can customize this behaviour. See Redefinable Agents for more information.

## Extension Module Options

If your custom extension module needs options these are defined by defining a stereotype which inherits the `Options` class. In order to specify which Option substereotype that belongs to a particular ExtensionModule subclass a dependency is used. The dependency goes from the ExtensionModule subclass to the Option substereotype.

One option is common for all extension modules; the list of paths to files/folders to import. This corresponds precisely to the paths entered in The First Step of the File/Folder Import Wizard.

## Redefinable Agents

The following agent operations are available for implementation in order for an extension module to customize the behaviour of the File/Folder importer.

### OnImportFile

Called when the File/Folder Importer has found a file to import.

**Model context:** The entity into which the importer will place the representation of the file.

**Parameter 'path' (in):** Path to the file.

**Parameter 'entity' (out):** Representation of the file, inserted in the model context if possible. This parameter can be set to NULL in order to skip importing this particular file.

The default implementation creates a file artifact for the file, and inserts it into the context, which by default is a package (either the top-level package created by the importer, or a package representing a folder).

An extension module can implement this agent to use a custom representation of the file in the model. It can also call the inherited `OnImportFile` agent operation if it wants to reuse parts of the default file representation.

### OnImportFolder

Called when the File/Folder Importer has found a folder to import.

**Model context:** The entity into which the importer will place the representation of the folder.

**Parameter 'path' (in):** Path to the folder.

**Parameter 'entity' (out):** Representation of the folder, inserted in the model context if possible. This parameter can be set to NULL in order to skip importing this particular folder.

The default implementation creates a package for the folder, and inserts it into the context, which by default is a package (either the top-level package created by the importer, or a package representing a folder).

An extension module can implement this agent to use a custom representation of the folder in the model. It can also call the inherited `OnImportFolder` agent operation if it wants to reuse parts of the default folder representation.

### OnImportCompleted

Called when the File/Folder Importer has completed the import of all specified files and folders.

**Model context:** The top-level package created by the importer.

The default implementation does nothing.

An extension module can implement this agent in order to perform analysis of imported files or folders. For example, it may add dependencies between file artifacts to represent some kind of relationship between the corresponding files.

## An Example

As an example of how to add a custom extension module for the File/Folder importer let's change the default representation of text files to use a class instead of a file artifact. The contents of the text file will be attached as a comment on the class. For this extension module we will use the Tcl API in order to implement a redefined behaviour of the `OnImportFile` agent.

But before writing any code, we start by defining the extension module class. For the sake of the example, we also add an option stereotype with one boolean option `addComments` controlling if the text of the file should be attached as a comment on the class. Note the dependency from the `Commentizer` class to the `CommentizerOptions` stereotype. This dependency relates the extension module with its options.

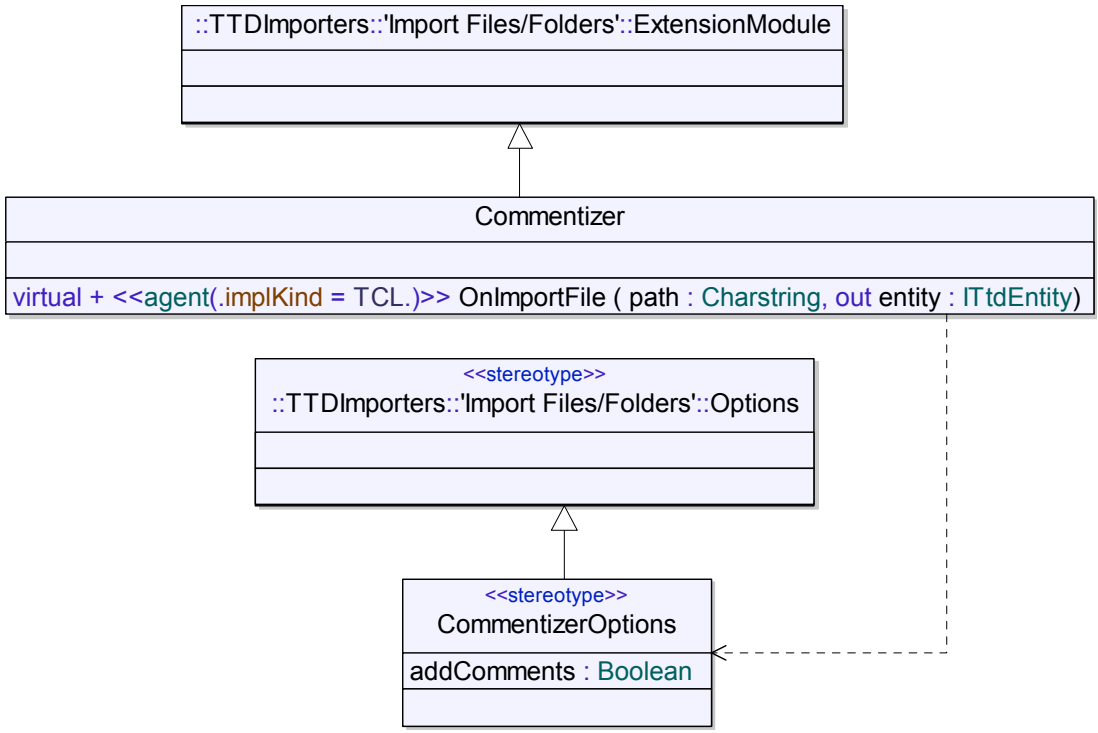



*Figure 279 Example extension module class and options stereotype*

If we now open the File/Folder Import Wizard we will see our Commentizer extension module in the last wizard page. We can also press the **Options...** button to set the `addComments` option.

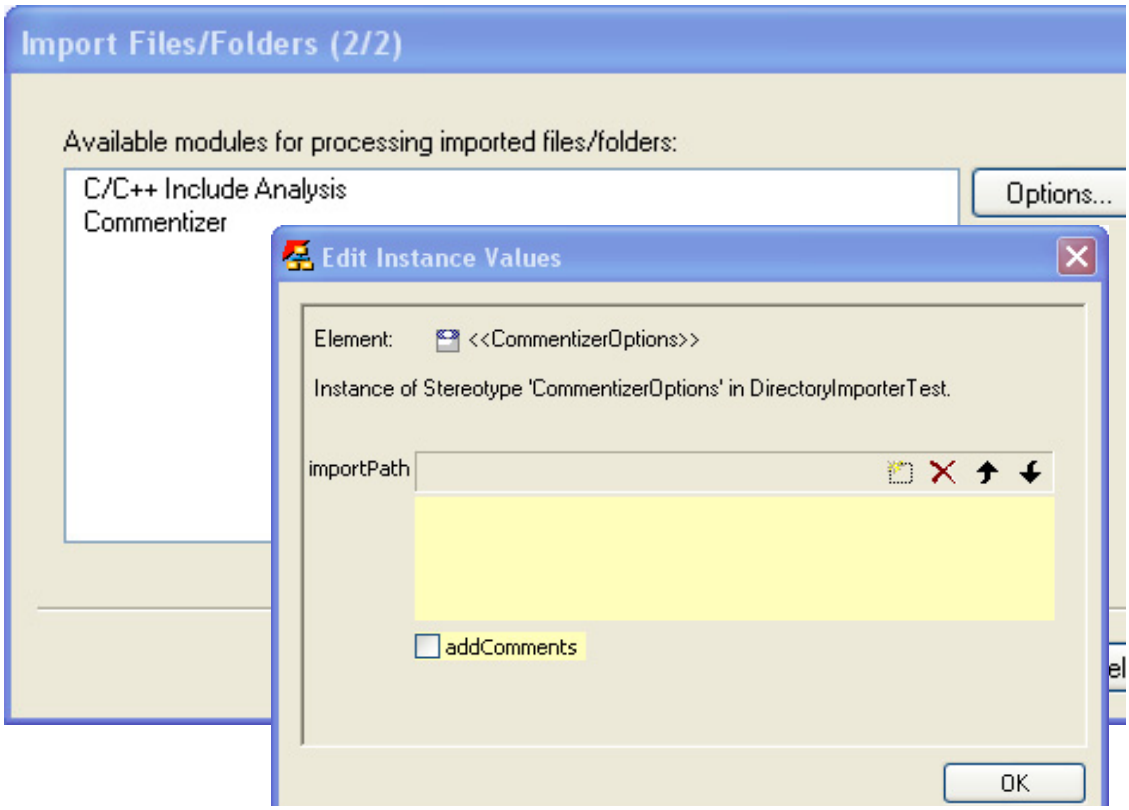



*Figure 280 Second import wizard page customized*

Now we are ready to implement the `OnImportFile` agent. We use the following Tcl script:

```
proc OnImportFile { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap

  set path [lindex $ap 0]
  set entity [lindex $ap 1]

  if {[file extension $path] != ".txt"} {
    ## Only import text files
    return
  }

  set cls [u2::Create $context "Class"]
  u2::SetValue $cls "Name" $path

  lset ap 1 $cls

  ## Find Commentizer options
  set model [u2::GetModel $context]
  set e $context
  while {[u2::GetOwner $e] != $model} {
```

```
    set e [u2::GetOwner $e]
  }
  set ac [u2::GetTaggedValue $e
"CommentizerOptions(.addComments.)"]
  if {$ac != 0} {
    if {[u2::Unparse $ac] == "true"} {
      ## Read file contents into a comment
      set fp [open $path r]
      set contents [read $fp]
      close $fp
      set comment [u2::Create $cls "Comment"]
      u2::SetValue $comment "Text" $contents
    }
  }
}
```

The script first extracts the agent parameters and checks if the ‘path’ specifies a text file. If not, nothing more is done for this file.

If it is a text file a class is created in the context to represent it. Also, the ‘entity’ out parameter is set-up to the created class.

The rest of the script just reads the ‘addComments’ tagged value from the import package. If it is set to true the contents of the file is read into a comment attached to the class.

# 74

# *Predefined Stereotypes and Attributes*

This section contains a reference to the available stereotypes and attributes, listed in alphabetic order.

In this section there is a list containing the model information from the pre-defined profile libraries.

The Complete listing is only available in the on-line help file.

# 75

# Agents

This chapter explains the concept of an agent, and describes how agents may be used in order to customize different parts of Telelogic Tau.

An **agent** is an executable module (a piece of code) that can be hooked into Tau, in order to add new functionality, or customize the existing tool behavior. An agent is defined in UML as an operation, but its implementation is provided in another implementation language, typically using one of the Tau public APIs (COM, C++ or Tcl).

In the UML definition of an agent, it is also possible to specify when it shall be invoked. This is done by specifying a dependency from the agent operation to a **tool event**. A tool event represents an event that may occur within the Tau application. When an event occur within Tau for which there is a corresponding tool event defined, the tool event is said to be **triggered**. When a tool event is triggered all agents that have dependencies to it will also be invoked.

It is also possible to manually invoke an agent using the Tau APIs. This can for example be useful in order to invoke an agent from an add-in and is actually a bridge between the two main techniques for customizing Tau, Add-Ins and agents. This means that an Add-In does not have to be completely implemented as a TCL script, but can use agents implemented in another language. Some benefits with such an approach can for example be improved performance, better means for debugging the Add-In, reuse of software written in another language etc.

When an agent gets invoked it gets a **model context** as an in-parameter from the caller. The model context is typically a model entity that the agent shall process in one way or another. The meaning of "process" here is very loose - it can mean anything from performing customized semantic checks, to producing some reports for external use, or modifying the entity. However, in all cases the agent performs its work from the context of the provided model entity; hence the name model context.

In addition to the model context, an agent may obtain any number of additional actual parameters from the caller. This is true both when the agent is invoked because a tool event was triggered, and when it is explicitly invoked from the APIs. These parameters can be used as in/out parameters and thus allow the agent to pass information back to the caller.

**Note**

*An agent always is a client of the Tau APIs. Everything that is applicable for API clients in general is also applicable for agents. For example, the term interactive agent, refers to an interactive API client, i.e. a client that executes in the same memory space as the Telelogic Tau IDE. An example of a non-interactive agent is an agent that runs in the memory space of another Tau application, such as a batch code generator executable.*

# Defining an Agent

The first step when creating an agent is to define it in UML. This is as simple as creating an operation somewhere in the model, giving it a good name, and applying the «agent» stereotype from the `TTDAgent` profile (a built-in library that is available in all UML models). It does not matter where the agent operation is created, but it is common to place it in a separate package, which could be a profile package.

Figure 281 on page 2027 shows the definition of an agent called CheckClassName. This agent intends to add some additional semantic checks concerning valid names for a class.

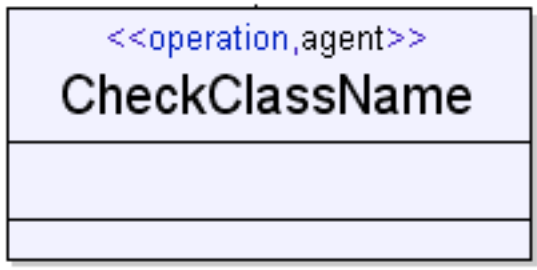



*Figure 281: Definition of an agent*

## Agent Invocation Triggered by a Tool Event

The next step is to think about how and when the agent shall be invoked. The most common is that an agent is invoked when a tool event is triggered.

Just like an agent, a tool event is represented as an operation in UML, but with the «'tool event'>> stereotype applied instead. The `TTDAgent` profile contains all available Tool Events. To specify that an agent shall be invoked every time a certain tool event is triggered, you must create a dependency from the agent to the tool event. The dependency should be stereotyped by one of the following stereotypes from the `TTDAgent` profile:

- «'before processing'>>
  Specifies that the agent shall be invoked before the default processing of the tool event.
- «'after processing'>>
  Specifies that the agent shall be invoked after the default processing of the tool event.

**Example 624: Custom semantic check agents** ———————————————

Consider the addition of two custom semantic checks to be applied on all classes. The first check is to be done **before** all standard checks of the class are performed, and the second check is to be done **after** all standard checks have been performed. Define two agents like Figure 282 on page 2028.

———————————————————————————————

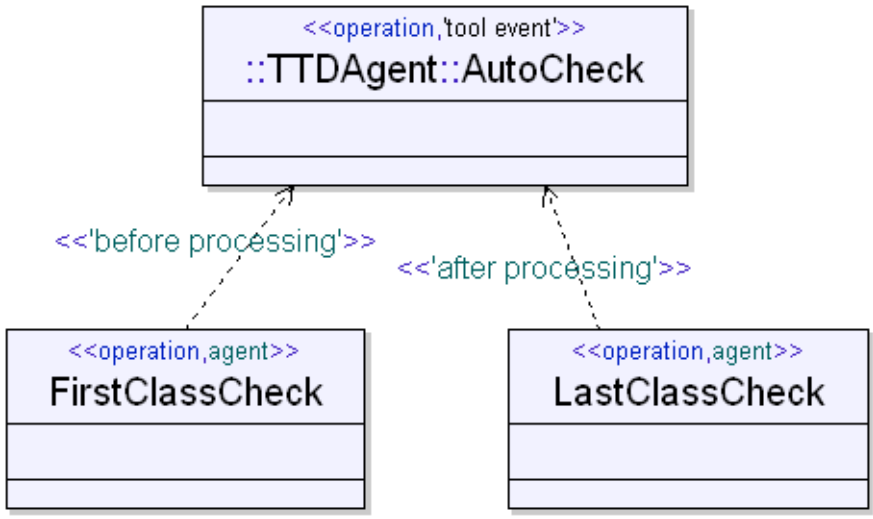



*Figure 282: Definition of an agent*

It is also possible to specify that an agent shall be invoked whenever some other agent is invoked. This is done in exactly the same way as in Example 624 on page 2027, with the only difference being that the supplier of the dependency is another agent instead of a tool event. This makes it possible to create entire "invocation trees" where the triggering of a single tool event may cause several agents to be invoked. The order of invocation is always specified by means of dependencies stereotyped with the <<'before processing'>> and <<'after processing'>> stereotypes. Note the following about such dependencies:

1. If two agents A and B both have the same kind of ordering dependency to a tool event T, it is undefined whether A will be invoked before or after B when T is triggered. This is true even if there is an ordering dependency between A and B.
2. Do not specify circular dependencies, leading to loops in the invocation order. If a loop is detected when an agent A is about to be invoked, the agent framework will refuse to invoke A and will instead continue with the next agent in the invocation order.

Figure 283 on page 2029 illustrates these points.

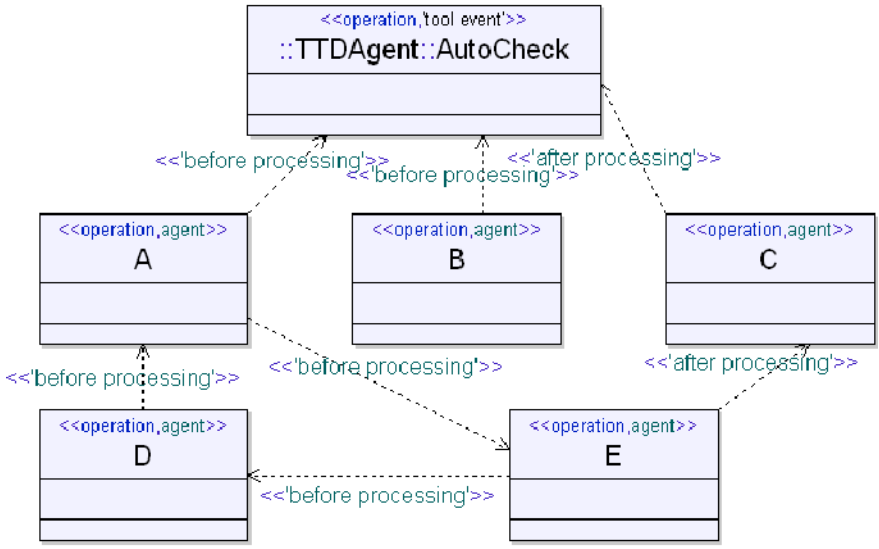



*Figure 283: Ordering dependency*

When the AutoCheck tool event is triggered the agents will be invoked in this order: E D A B C D A E. Or, since the order of invocation between A and B is undefined, B E D A C D A E. The dependency between A and E is ignored since it causes a loop in the invocation order.

## Programmatic Agent Invocation from the APIs

Instead of specifying the invocation of agents statically using ordering dependencies it is possible to invoke an agent programmatically from one of the Tau APIs (COM, C++ or Tcl). The API method to use is `InvokeAgent`, and it lets you invoke a specified agent on a specified model context, providing any number of actual agent parameters.

**Note**

*Ordering dependencies have no meaning when invoking an agent explicitly from the APIs. That is, only one agent will be invoked for each call to* `InvokeAgent`.

# Implementing an Agent

Before an agent can be used it has to have an implementation defining what it shall do when it gets invoked. An agent can currently be implemented in C++ (using the C++ API), in a COM/.NET enabled language (using the

COM API), in Tcl (using the Tcl API), or as a Query expression. Tagged values in the «agent» stereotype instance specifies how an agent is implemented and where to find its implementation.

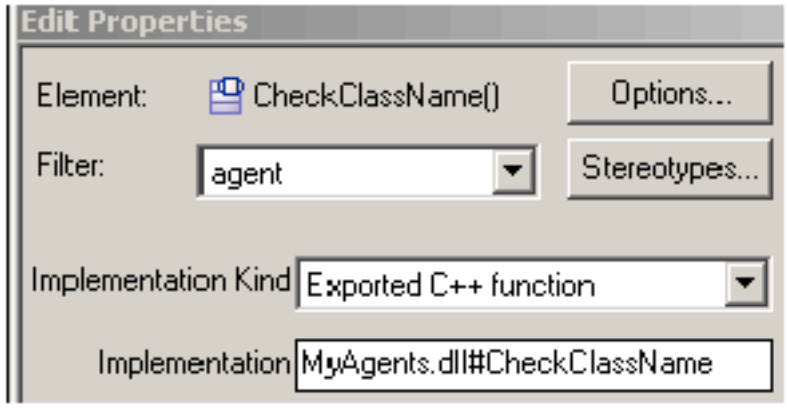



*Figure 284: Tagged values in the «agent» stereotype*

The Implementation Kind is one of the following:

- COM object or `.NET` assembly
  Select this if you want to implement the agent using the COM API.
- Exported C++ function
  Select this if you want to implement the agent using the C++ API.
- TCL
  Select this if you want to implement the agent using the Tcl API.
- Query expression
  Select this if you want to implement the agent as a query expression. This is only applicable if the agent you implement is a query (see Queries for more information about queries).
- Internal
  Reserved for internal use.

The Implementation is a string specifying where to find the implementation of the agent. What to write in this field depends on the selected implementation kind.

- For COM/.NET, the implementation string should specify the programmatic id of the COM/.NET object to invoke.
- For C++, the implementation string should be on the form:

`<path>#<function>`
`<path>` is the path to the dynamic link library or shared object that contains the implementation. The path may either be absolute or relative. In the latter case it will be interpreted against the PATH (on Windows) or LD_LIBRARY (on Unix) environment variables. The path may also contain URNs. If the library cannot be loaded, and a relative path is used, a second attempt to load the library is made, where the relative path is interpreted against the location of the U2 file where the agent is defined.
`<function>` is the name of the C++ function to call.

- For Tcl, the implementation string should be on the form:

  `<path>#<procedure>`
  `<path>` is the path to the Tcl script file that contains the implementation. The path may either be absolute or relative. In the latter case it will be interpreted against the location of the U2 file where the agent is defined. The path may also contain URNs.
  `<procedure>` is the name of the Tcl procedure, within the specified file, that should be called.

  It is also possible to leave the implementation string unspecified. In that case the Tcl script shall be placed as an informal implementation in an operation body for the agent. This allows the agent implementation to be stored inside the UML model, which besides from sometimes being more convenient than referring an external file, also is somewhat more efficient. Note that in this case the Tcl procedure to call must have the same name as the agent.

- For Query expression, the implementation string should specify a file containing the query expression. The file may be specified with either an absolute or relative path. In the latter case it will be interpreted against the location of the U2 file where the agent is defined. The path may also contain URNs.

  It is also possible to leave the implementation string unspecified. In that case the query expression shall be placed as an informal implementation in an operation body for the agent. This allows the agent implementation to be stored inside the UML model, which besides from sometimes being more convenient than referring an external file, also is somewhat more efficient.

### General guidelines for agent implementations

When writing the implementation of an agent, there are a few guidelines that should be followed regardless of implementation language (except when using query expressions, which are not a general purpose programming language):

1. Begin the implementation with tests to see if all conditions are fulfilled for the agent to execute. If any condition is not fulfilled return immediately. This is particularly important for interactive agents that are invoked when "low-level" tool events are triggered. Such agents are typically invoked very frequently, and in order not to decrease the performance of Tau too much, it is important that the agent does not perform unnecessary work.
2. Avoid intrusive error reporting, such as popping up dialog boxes, unless the agent is interactive. The recommended way of reporting errors that occur in the agent is to use the `ITtdModelAccess::WriteMessage` method, which will print the error message in a way that is appropriate for the environment in which the agent executes.
3. An agent may spawn threads for performing tasks asynchronously, but be careful. It is not safe to access the model from multiple threads simultaneously.

### See also

"Implementation using the COM API" on page 2032

"Implementation using the C++ API" on page 2034

"Implementation using the Tcl API" on page 2035

"Implementation using Query Expressions" on page 2037

## Implementation using the COM API

The COM API contains an interface called `ITtdAgent`. This is a callback-interface that all agents must implement. It contains one method `Execute`, which will be called by Tau when the agent shall be invoked.

Build and deploy the COM object, and give it a good programmatic id. This id is used as the "implementation" tagged value of the 'agent' stereotype that is applied on the agent definition. When Tau wants to invoke the COM agent, it will attempt to create an instance of the COM object that has the specified

id. If that succeeds it then does a `QueryInterface` for the `ITtdAgent` interface on the object. And if that also succeeds, the `Execute` method will be called.

**Example 625: A COM agent**

The skeleton C++ code generated by the Visual Studio ATL COM wizard for a COM object implementing the `ITtdAgent` interface will look similar to this:

```
class ATL_NO_VTABLE CMyCOMAgent :
  public CComObjectRootEx<CComSingleThreadModel>,
  public CComCoClass<CMyCOMAgent, &CLSID_MyCOMAgent>,
    public ITtdAgent {
public:
DECLARE_REGISTRY_RESOURCEID(IDR_MYCOMAGENT)
DECLARE_PROTECT_FINAL_CONSTRUCT()
BEGIN_COM_MAP(CMyCOMAgent)
    COM_INTERFACE_ENTRY(ITtdAgent)
END_COM_MAP()
virtual HRESULT __stdcall raw_Execute (
  ITtdEntity* pTriggedBy,
  VARIANT bBeforeProcessing,
  ITtdEntity* pModel,
  IUnknown* pServer,
  SAFEARRAY ** eventProperties ) {
// Implementation...
}
};
```

Instead of implementing the agent with a COM object, it is also possible to use a .NET assembly that is configured for COM interoperability.

**Example 626: A C# agent**

Here is a sample C# implementation of an agent which appends a prefix “Class_” to the name of its model context (a class):

```
using System;
using U2ModelAccessTypeLib;

namespace CSharpAgent
{
  public class C : ITtdAgent
  {
    public void Execute(ITtdEntity triggeredBy,
                        object beforeProcessing,
                        ITtdEntity entity,
                        object server,
```

```
                              System.Array eventProperties)
    {
      if (entity.IsKindOf("Class"))
      {
        string s = "Class_";
        entity.SetValue("Name", s +
entity.GetValue("Name", 0), 0);
      }
    }
  }
}
```

The programmatic id to use as the “Implementation” tagged value will in this case be CSharpAgent.C (i.e. the qualified name of the C# class that implements the agent).

---

## Implementation using the C++ API

The C++ API contains a header file called U2Agent.h. In this header file you can find the following macro definitions:

```
#define AGENT_PARAMETERS const u2::ITtdEntity*
pTriggeredBy, u2::EventTiming timing, u2::ITtdEntity*
pContext, u2::IUnknown* pServer, u2::AgentParameters&
agentParameters, u2dll::Cu2Changer& changer
#define AGENT_SIGNATURE(name) void
name(AGENT_PARAMETERS)

#ifdef _MSC_VER
#define DLLEXPORT __declspec(dllexport)
#else
#define DLLEXPORT
#endif

// Use this macro to define the implementation of an
agent as an exported function.
#define AGENT_IMPL(name) extern "C" DLLEXPORT
AGENT_SIGNATURE(name)
```

The `AGENT_IMPL` macro can be used for a compact definition of an agent implementation in C++. As can be seen above it will expand to an exported function with C linkage.

**Example 627** ———————————————————————————

The C++ implementation of a simple agent that adds a default prefix to the name on a newly created definition can look like this:

```
AGENT_IMPL(SetNameForMemberAttribute) {
  if (!pContext->IsKindOf(_T("Definition")))
    return; // Not a definition - return immediately!
  tstring strName;
  pContext->GetValue(_T("Name"), strName);
  tstring strNewName(_T("MYPREFIX_"));
  strNewName += strName;
  pContext->SetValue(_T("Name"), strNewName, 0, changer);
}
```

It is assumed that the agent is invoked when the 'Create Entity' tool event is triggered (on <<'after processing'>>).

---

**Note**

*If you are implementing the agent on the Windows platform, you should link it with the run-time libraries of Visual Studio 2005 SP1. The run-time libraries that come with other versions of Visual Studio are not compatible with the run-time libraries used by Tau. Using a mix of run-time libraries can cause memory operations to fail, and lead to instability.For the same reason it is important to use the Release version of the run-time libraries, and not the Debug version. See* *"Debug C++ agents in Visual Studio" on page 2280 in Chapter 78, C++ API* *for more information on how to debug an agent.*

## Implementation using the Tcl API

Before implementing an agent using Tcl, note that the Tau Tcl interpreter is only available in the IDE process (vcs.exe). This means that only interactive agents can be implemented by means of the Tcl API.

The most convenient way to implement a Tcl agent is to follow these steps:

- Create a UML operation that defines the agent. Give it a name and define its parameters if any.
- Select the operation in the Model View, and bring up the context menu. Select **Utilities - Turn into Tcl agent**. This command will turn the operation into an agent implemented by a skeleton Tcl script.

The Tcl procedure that is specified as implementation for an agent must have the following signature:

```
proc <name> { triggeredBy timing context server agentParameters }
```

Note that if the Tcl script is put as an inline informal implementation of the agent in the UML model `<name>` must be identical with the name of the agent.

The parameters to the procedure corresponds directly to the parameters of the `ITtdAgent` interface. However, note the following about the Tcl representation of these parameters:

- The `server` parameter is not applicable and will always be NULL (0). This is because Tcl agents always are interactive agents - instead of using the `server` parameter for interacting with the Tau IDE, the entire Tcl API can directly be used.
- The `agentParameters` parameter is a Tcl list of strings representing the agent parameters. These strings are encoded according to the rules described in the documentation of the Tcl command u2::InvokeAgent (see Model Commands). Note also that this is an in/out parameter which means that you have to use the `upvar` command to access it inside the procedure.

**Example 628: A simple Tcl agent**

The Tcl implementation of a simple agent that toggles the activeness of a class is shown below. The agent returns an agent parameter that is false (0) if the class was made passive, and true (1) if the class was made active.

```
proc MyAgent { triggeredBy timing context server
agentParameters } {
upvar 1 $agentParameters ap
  if {[u2::IsKindOf $context "Class"]} {
    set isActive [u2::GetValue $context "isActive"]
    if {$isActive == false} {
      u2::SetValue $context "isActive" true
      set ap [list 1]
    } else {
      u2::SetValue $context "isActive" false
      set ap [list 0]
    }
  }
```

```
}
```

**Note**

*If the Tcl script contains "global" code outside the specified Tcl procedure, that code will execute before the Tcl procedure is called.*

**Example 629: Implementing a query agent using Tcl**

This example shows how to add query result entities to the result list, which always is the first agent parameter. The agent will add the model (Session) of its model context, followed by the model context itself.

```
proc MyQueryAgent { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  set model [u2::GetEntity $context Session]
  set ap [lreplace $ap 0 0 [list $model $context] ]
}
```

## Implementation using Query Expressions

Query expressions are not a general purpose programming language. Therefore, they should only be used for implementing agents that are queries.

The most convenient way to implement an agent by means of a query expression is to use the Save button in the Query dialog. See Saving a query expression as a new query for more information. But of course it is also possible to create a query expression agent manually and enter the query expression either in an external file, or as an informal implementation of the agent. The syntax of the query expression is the same as is used in the query dialog.

**Example 630: Implementing a query agent using a query expression**

Assuming we have a predicate agent `MyPredicateAgent` (i.e. an agent returning true or false based on some condition for its model context), the following query expression can be used as the implementation of a query agent that will find all entities in the model rooted at its model context for which the predicate is fulfilled:

```
GetAllEntities().select(MyPredicateAgent())
```

**Note**

*If an agent is implemented by means of a query expression it cannot have formal parameters.*

# Agent Parameters

Agents are defined as operations in UML so it is natural to allow agents to have formal parameters just like any operation. However, since the call of such an agent operation is a late-bound call (i.e. it is not resolved until at run-time) the use of agent parameters become much more flexible than the use of parameters with ordinary operations. The price for this flexibility is that checks for the expected number and type of agent parameters has to be performed by the called agent.

In general an agent can take any number of actual parameters. Each parameter must have one of the following types:

| Type | C++ Type | COM Type |
|---|---|---|
| Text string | `tstring` | `BSTR` |
| General (unknown) interface on object | `u2::IUnknown` | `IUnknown` |
| List of unknown interfaces | `std::list<u2::ITtdEntity*>` | `ITtdEntities` |
| Integer | `long` | `long` |
| Boolean | `bool` | `VARIANT_BOOL` |
| Untyped pointer | `void*` | `N/A` |

The table also shows which types that are used in the C++ and COM APIs to represent a particular agent parameter type. The Tcl mapping of these types are described in "Model Commands" on page 2221 in Chapter 77, *Tcl API*.

The declaration of an agent in UML should include the specification of which parameters it has, and the type of these parameters. Although this information is currently not used by the agent framework when an agent is invoked, it is valuable information in order to document the agent and make it easier to use by clients.

The implementation of an agent that expects actual parameters should be written in such a way that error cases are handled. The agent must not assume a particular number of actual parameters, nor may it make assumptions about the parameter types. The example below shows a typical implementation of a C++ agent that expects two actual parameters; the first one of boolean type and the second one of string type.

**Example 631**

```
AGENT_IMPL(MyAgent) {
  if (agentParameters.size() != 2) {
    // Error: Too few or too many actual parameters!
    return;
  }

  bool par1;
  tstring par2;
  try {
    par1 = agentParameters.front()->GetBoolean();
    par2 = agentParameters.back()->GetString();
  }
  catch (u2::AgentParameter::ETypeMismatch) {
    // Error: Wrong parameter type!
    return;
  }
}
```

# Tool Events

This section lists all available tool events that can invoke user-defined agents when they are triggered. The UML definition of these tool events can be found in the `TTDAgent` profile unless otherwise stated.

For each tool event it is described

- when it is triggered in Tau.
- whether it is meaningful to use both ‘before processing’ and ‘after processing’ ordering dependencies for the tool event (for some tool events only one of these is meaningful).
- actual parameters that are passed with the tool event and how to use them.

The tool events are categorized according to the functionality or feature in Tau that triggers them.

**Note**

*Some add-ins may define additional tool events not described here. Such tool events are typically internally used by the add-in implementation. In case they are intended to be used in customizations of the functionality provided by the add-in, their documentation can be found in the profile where they are defined.*

## Semantic checker events

### AutoCheck

This event is sent from the Tau IDE when an entity is automatically checked by the semantic checker. The purpose of the event is to allow performing custom semantic checks during auto check.

**Note**

*The AutoCheck event is sent when Tau is idle, on entities that were recently modified. Agents that trigger on this event should therefore restrict themselves to only reading from the model. If changes are made to the model these will not be possible to undo.*

Agents that trigger on 'before processing' will be invoked before all built-in checks of the entity are performed. Agents that trigger 'after processing' will be invoked after all built-in checks of the entity have been performed.

**Timing:**

1. ‘before processing’
2. The semantic checker performs all standard checks on the entity
3. ‘after processing’

**Model context:**

The model entity to check.

**Parameters:**

```
[in] messageList: ITtdMessageList
```

Message list where check messages can be reported.

### Check

This event is sent when an entity is checked for correctness by the semantic checker. This happens in the Tau IDE when selecting the commands “Check” and “Check All”, and also in the Tau code generators before generating code. The purpose of the event is to allow custom semantic checks to be performed.

Agents that trigger on 'before processing' will be invoked before all built-in checks of the entity are performed. Agents that trigger 'after processing' will be invoked after all built-in checks of the entity have been performed.

**Timing:**

1. ‘before processing’
2. The semantic checker performs all standard checks on the entity
3. ‘after processing’

**Model context:**

The model entity to check.

**Parameters:**

```
[in] messageList: ITtdMessageList
```

Message list where check messages can be reported

## Application builder events

### AB AutoSave

This event is sent by the Tau IDE when the Application Builder performs an automatic save of the model before starting to build it.

Agents that trigger on 'before processing' will be invoked before the model is saved. Any changes made by such agents will thus be present in the model that is loaded by the code generator that performs the build. This can be used to make changes in the model that shall affect code generation, and still be present in the model that is loaded in the IDE.

Agents that trigger 'after processing' will be invoked after the model has been saved. Changes made by such agents will thus not be seen by the code generator that performs the build. Therefore, the ‘after processing’ of this event is mostly useful for performing tasks that do not change the model.

**Timing:**

1. ‘before processing’
2. Application Builder saves the model that is about to be built.
3. ‘after processing’

**Model context:**

The Build Artifact that is being built.

**Parameters:**

```
[in] entity1: ITtdEntity
[in] entity2: ITtdEntity
...
[in] entityN: ITtdEntity
```

There is one input parameter for each entity that is about to be built by the Application Builder. If the Build Artifact is selected to be built, these are the entities that are manifested by the build artifact. If the build is started by selecting some other entities, these entities will be passed instead.

### Insert cross reference file

This event is sent by the Tau IDE when the Application Builder detects that a code generator (or more generally, an Application Builder client) has generated a “cross reference” file. Such a file contains a UML representation of all files that were generated by the code generator and a mapping between entities of the original UML model to positions within these generated files. Both the C and the C++ code generators produce cross reference files on a common format.

The UML representation of a cross reference file is a package containing a list of file artifacts, representing generated files. Each file artifact has a list of `<<filePosition>>` stereotype instances to represent those positions within the generated file that corresponds to translated UML entities.

The Application Builder will insert the generated cross reference file into the UML model, in order to facilitate navigation from the model to generated code and vice versa as well as other related features.

**Timing:**

1. Application Builder detects a generated “cross reference” file.
2. ‘before processing’

3. Application Builder loads the generated “cross reference” file into the model.
4. ‘after processing’

**Model context:**

The Build Artifact that has been built.

**Parameters:**

```
[in] fileName : Charstring
```

This is the name of the cross reference file.

```
[in] messageList : ITtdMessageList
```

This is the message list of the Application Builder. Messages added to this list will be printed in the Build tab.

```
[in] resultPackage : ITtdEntity
```

(only on ‘after processing’)
This is the package that results from loading the cross reference file.

### AB Client File Response

This event is a more general version of the “Insert cross reference file” event. It is sent by the Tau IDE when the Application Builder detects that a code generator (or more generally, an Application Builder client) has generated a general “response file” (i.e. a file for which the Application Builder performs no default action). The purpose of the event is to allow an Application Builder client to communicate information from the code generator to the Tau IDE.

Since there is no default task that is performed when this event is triggered, there is no difference between triggering on ‘before processing’ or ‘after processing’.

**Timing:**

1. Application Builder detects a generated “response file”.
2. ‘before processing’
3. ‘after processing’

**Model context:**

The Build Artifact that has been built.

**Parameters:**

```
[in] fileName : Charstring
```

This is the name of the response file.

```
[in] messageList : ITtdMessageList
```

This is the message list of the Application Builder. Messages added to this list will be printed in the Build tab.

### Process BuildArtifact

This event is sent by an Application Builder code generator when it processes a Build Artifact. The event will thus be sent exactly once for each time the code generator is launched. The purpose of the event on 'before processing' is to be able to perform pre-translation transformations of the generated model. On 'after processing' the event can be used as a trigger for generating additional code that is not generated by the standard code generator.

**Timing:**

1. 'before processing'
2. The code generator generates code for the build artifact.
3. 'after processing'

**Model context:**

The build artifact to generate code for.

**Parameters:**

```
[in] messageList : ITtdMessageList
```

This is the message list used by the code generator to report translation messages. Messages added to this list will be printed in the Build tab if the code generation is started from the Tau IDE. If the code generation is started from the TauBatch executable, the messages will be printed to `stdout`.

# Editor events

### OpenDiagram

This event is sent by the Tau IDE when a diagram (of any kind) is opened, or activated. The purpose of the event is, among other things, to support integrations with other tools that may need to be invoked when a diagram is opened.

**Timing:**

1. ‘before processing’
2. The diagram becomes visible in the editor.
3. ‘after processing’

**Model context:**

The diagram being opened.

**Parameters:**

No parameters.

### InsertDiagramElement

This event is sent by the Tau IDE when a diagram element (symbol or line) has been inserted in a diagram. Diagram elements can be inserted in many ways for example by placing new symbols from the tool bar, or pasting symbols or lines from the clipboard. The purpose of the event is to be able to customize newly added diagram elements, for example modifying their size or position, or to automatically add additional diagram elements.

**Note**

*This event is only triggered on <<after processing>>, that is when the diagram already has been added to the diagram.*

**Timing:**

1. The diagram element is inserted to the diagram.
2. ‘after processing’

**Model context:**

The inserted diagram element.

**Parameters:**

No parameters.

## Model interaction events

### Change Entity Property

The ‘Change Entity Property’ event is sent by the Tau IDE when a property (i.e. a metafeature value) of a model entity is changed. The event is sent when changing the value of all metafeatures, except owner and composition links. The purpose of the event is, among other things, to support transformations to take place already when some property of an entity is changed.

**Note**

*The event will not be sent when a property is changed as a consequence of doing an undo or redo. This simplifies the implementation of agents that trigger on this event considerably. Any changes that are performed by such an agent will automatically be part of the same undo/redo step. To trigger on modifications that takes place on Undo / Redo, see the Entity Modified event.*

**Timing:**

1. ‘before processing’
2. The value of the entity is changed.
3. ‘after processing’

**Model context:**

The entity that is modified.

**Parameters:**

```
[in] metaFeatureName : Charstring
```

The name of the metafeature that is modified.

```
[in] value : Charstring
```

A string encoded value. On ‘before processing’ this is the new value that is about to be set, and on ‘after processing’ this is the old value, as it looked before the change took place.

The table shown for "GetValue" on page 2089 in Chapter 76, *COM API* provides more information on how values are string encoded.

If the metafeature is of metaclass type, the new value may be a pointer to an entity. In this case the value parameter will have the type ITtdEntity instead

### Entity Modified

This event is sent by the Tau IDE when an entity in the model is modified, also when the modification takes place as part of an Undo or Redo operation. The purpose of the event is to provide a means for general model change detection, and to allow agents to update custom views of the model.

**Note**

*Since this event is sent also when an entity is modified during Undo / Redo it is not recommended for agents that trigger on this event to modify the model in any way. Triggered agents should restrict themselves to updating views of the model.*

**Timing:**

1. The entity gets modified.
2. 'before processing'
3. If the entity is saved in a file (resource) that file gets marked as modified.
4. The entity is scheduled for semantic checks (AutoCheck).
5. 'after processing'

**Model context:**

The modified entity.

**Parameters:**

No parameters.

### Create Entity

This event is sent by the Tau IDE when a new entity is created in the model. The purpose of the event is, among other things, to support additional entities to be created automatically when an entity is created.

**Timing:**

1. ‘before processing’
2. The entity is created, and inserted into the model.
3. ‘after processing’

**Model context:**

On ‘before processing’ the model context is the entity that will be the owner of the new entity. On ‘after processing’ the model context is the newly created entity.

**Parameters:**

```
[in] metaClassName : Charstring
```

(only on ‘before processing’)

The name of the metaclass of the entity that is about to be created.

**Move Entity**

This event is sent by the Tau IDE when an entity is moved in the model. The purpose of the event is, among other things, to redefine the default 'move' semantics.

**Note**

*If the moved entity contains other entities, these will of course also be moved. However, the ‘Move Entity’ event will only be sent for the top entity.*

**Timing:**

1. ‘before processing’
2. If the `defaultProcessing` flag is true the entity is moved. Otherwise nothing happens.
3. ‘after processing’

**Model context:**

The entity that is moved.

**Parameters:**

```
[in] target : ITtdEntity
```

The target entity to which the entity is moved (i.e. the new owner of the moved entity).

```
[in/out] defaultProcessing : Boolean
```

(only on ‘before processing’)
This flag is set to true by Tau in the call to the agent. If the agent does not change it (i.e. it is still true after the call), the move will be performed in the usual way. Agents that redefine the meaning of a move operation should thus set this flag to false before returning.

# Editor events

### AutoLayout

This event is sent by the Tau IDE when the “Automatic Layout” command is selected in an editor. The purpose of the event is to customize, or override completely, the default autolayout behavior.

**Timing:**

1. ‘before processing’
2. If the `defaultProcessing` flag is true the autolayout will be performed according to the default algorithm. Otherwise nothing happens.
3. ‘after processing’

**Model context:**

The diagram that contains the diagram elements that are being processed.

**Parameters:**

```
[in/out] defaultProcessing : Boolean
```

(only on ‘before processing’)

This flag is set to true by Tau in the call to the agent. If the agent does not change it (i.e. it is still true after the call), the autolayout will be performed in the usual way. Agents that redefine the autolayout algorithm should thus set this flag to false before returning.

```
[in] entity1 : ITtdEntity
[in] entity2 : ITtdEntity
...
[in] entityN : ITtdEntity
```

These are the diagram elements that are being processed.

## Storage Events

### LoadModel

This event is sent by the Tau IDE when a new model is loaded (by loading a project file). The purpose of the event is to be able to perform some tasks just before or just after a new model has been loaded into Tau.

**Timing:**

1. An empty model is created.
2. ‘before processing’
3. The project file is loaded, and the result is inserted into the model. The model will also be bound.
4. ‘after processing’

**Model context:**

The model (ITtdModel).

**Parameters:**

```
[in] messageList : ITtdMessageList
```

This is the message list that are used by Tau when loading a model. Messages added to this list will be printed in the Messages tab.

### LoadResource

This event is sent by the Tau IDE when a resource (typically a `.u2` file) is loaded into the model. The purpose of the event is to be able to perform some tasks just before or just after a resource has been loaded into Tau.

**Timing:**

1. ‘before processing’
2. The resource is loaded, and the result is inserted into the model.
3. ‘after processing’

**Model context:**

On ‘before processing’ the model context is the model (ITtdModel) into which the result of the load shall be inserted. On ‘after processing’ the model context is the resource (ITtdResource) that has been loaded.

**Parameters:**

```
[in] path : Charstring
```

The path (or, more generally, the URI) of the resource that is loaded.

```
[in] messageList : ITtdMessageList
```

This is the message list that are used by Tau when loading a model. Messages added to this list will be printed in the Messages tab.

### SaveResource

This event is sent by the Tau IDE when a resource (typically a `.u2` file) is saved. The purpose of the event is to be able to perform some tasks just before or just after a resource has been saved.

**Timing:**

1. ‘before processing’
2. The resource is saved.
3. ‘after processing’

**Model context:**

The resource (ITtdResource) that is saved.

**Parameters:**

No parameters.

## C++ Application Generator Events

### Print C++ Source File

This event is sent by the C++ Application Generator when it prints the contents of a generated C++ source file. The purpose of the event is to be able to print a custom header or footer to the generated file.

**Timing:**

1. ‘before processing’
2. The default contents of the source file is printed.
3. ‘after processing’

**Model context:**

The file artifact that represents the generated source file in the model. This is a file artifact of the model-to-file mapping (which could have been automatically created by the code generator).

**Parameters:**

```
[in] sourceBuffer : ITtdSourceBuffer
```

This is a representation of the generated file. The source buffer object can be used in order to add text to the generated file.

### Print C++ Definition

This event is sent by the C++ Application Generator when it prints a C++ definition to a generated C++ source file. The purpose of the event is to be able to print something just before or just after the definition.

**Timing:**

1. ‘before processing’
2. The C++ definition is printed to the generated file.
3. ‘after processing’

**Model context:**

The C++ definition that is printed. The C++ definition is represented by an ITtdEntity of metaclass Definition.

**Parameters:**

```
[in] sourceBuffer : ITtdSourceBuffer
```

This is a representation of the generated file. The source buffer object can be used in order to add text to the generated file.

### Entity File Position

This event is sent by the C++ Application Generator when it is printing the generated files. The purpose of the event is to notify interested agents to the file positions that the generated entities were printed. A file position consists of the name of a generated file (full path), and a line and column within that file. The event is sent when printing the following entities:

- Definitions
  The event is sent when the name of the definition is printed.
- Operation bodies
  The event is sent when the opening curly bracket ({) is printed.
- Statements (actions)
  The event is sent when the semicolon (;) that terminates the statement is printed.

This event is used internally by the C++ Application Generator as the implementation of the “Goto source” menu item. It can also be of interest for an agent that wishes to produce some kind of database, index or report of the generated files, and their contents.

**Timing:**

1. ‘before processing’
2. ‘after processing’
3. The generated file is written.

**Model context:**

The entity that is printed.

**Parameters:**

```
[in] path: Charstring
```

The full path of the generated file into which the model context entity is about to be printed.

```
[in] line : Natural
```

The line number where the entity is printed.

```
[in] column : Natural
```

The column number where the entity is printed.

## Java Code Generator Events

The tool events for the Java code generator can be found in the library `TTDJavaModelCodeSync`, in the package `Java tool events`.

### JavaPrintFile

This event is sent by the Java code generator when it prints the contents of a generated Java source file. The purpose of the event is to be able to print a custom header or footer to the generated file.

**Timing:**

1. 'before processing'
2. The default contents of the source file is printed.
3. 'after processing'

**Model context:**

The file artifact that represents the generated source file in the model. This is a file artifact of the model-to-file mapping (which could have been automatically created by the code generator).

**Parameters:**

```
[in] sourceBuffer : ITtdSourceBuffer
```

This is a representation of the generated file. The source buffer object can be used in order to add text to the generated file.

See Example 718 on page 2242 for an example of an agent triggering on this event.

### JavaPrintDefinition

This event is sent by the Java code generator when it prints a Java definition to a generated Java source file. The purpose of the event is to be able to print something just before or just after the definition.

**Timing:**

1. 'before processing'
2. The Java definition is printed to the generated file.
3. 'after processing'

**Model context:**

The Java definition that is printed. The Java definition is represented by an ITtdEntity of metaclass Definition.

**Parameters:**

```
[in] sourceBuffer : ITtdSourceBuffer
```

This is a representation of the generated file. The source buffer object can be used in order to add text to the generated file.

**Transformation**

This event represents the step during Java code generation where UML constructs are transformed into Java constructs.

Agents that trigger on <<before processing>> on this event can implement custom transformations of UML constructs for which no standard transformation to Java is available. For example, a definition can have a special stereotype applied with a domain specific meaning when generating code. An agent can find definitions with that stereotype applied and replace them with other model constructs for which standard transformations to Java exist.

Agents that trigger on <<after processing>> on this event can examine the result of performing the standard transformations, and modify the resulting model in any way it likes. This is a means for customizing the default translation rules from UML to Java.

The Transformation tool event is triggered once for each generated Java source file.

**Timing:**

1. ‘before processing’
2. UML definitions manifested in a Java file artifact are transformed
3. ‘after processing’

**Model context:**

A copy of the Java file artifact which represents the Java source file that is generated.

**Parameters:**

```
[in] messages : ITtdMessageList
```

This is the message list used by the Java code generator. You can use AddMessage to report messages to that list, such as warnings or errors.

```
[in] buildartifact : ITtdEntity
```

This is the Java build artifact used.

```
[in] roots : ITtdEntities
```

These are copies of the entities that are manifested by the Java file artifact and which will be transformed to Java. Note that since they are copies of the original manifested entities an agent can change them freely without modifying the original model. The GUIDs of the entities are, however, the same as the original entities, which makes it possible to find the original entities by using FindByGuid on the 'originalModel' parameter. It is typical that an agent reads information from the original entity and writes information on the copied entity.

```
[in] originalModel : ITtdModel
```

This is the original model. It should be used as the context when finding the original entities to transform. An agent should not change anything in this model. If it does, these changes will show up in the original model.

# Model Verifier Events

### ModelVerifierTextTrace

This event is sent by the Tau IDE when the Model Verifier is about to trace a text message in its console window. It can be used to communicate information from a generated C application to the Tau IDE through the Model Verifier.

**Timing:**

1. 'before processing'
2. If the `shouldPrint` flag is true the text is traced in the console window. Otherwise nothing happens.
3. 'after processing'

**Model context:**

The top-level entity of the model that is being verified with the Model Verifier.

**Parameters:**

```
[in/out] text : Charstring
```

The text message to print in the console window. An agent that wants to change the text to print (only possible on ‘before processing’) should modify this parameter.

```
[in/out] shouldPrint : Boolean
```

(only on ‘before processing’)
This flag is set to true by Tau in the call to the agent. If the agent does not change it (i.e. it is still true after the call), the string in the text parameter will be printed in the console window. If the flag is set to false by the agent, the text will not be printed.

# Agent Commands

An agent command makes it possible to invoke a Tau agent from the Tau user interface. This mechanism makes it easy to invoke common agents that are frequently used. You may define your own agent commands in order to enable invocation of your own agents from the Tau user interface.

## Defining an Agent Command

An agent command is defined by following these steps:

- Create a dependency from the agent to the metaclass in TTDMetaModel that represents the kind of model context expected by the agent. For example, an agent that expects a class as model context should have a dependency to the `TTDMetaModel::Class` metaclass. If an agent can handle more than one kind of model context, multiple dependencies can be created.

- Apply the stereotype `<<agent command>>` on the dependency. Tagged values for this stereotype let you control how the agent command will appear in the Tau user interface:
  - **Target** specifies where in the user interface the command will appear. Currently it is only possible to bind the command to the context menu.
  - **Parameters** specifies actual values for the agent parameters, if any. The values are specified using textual UML expression syntax. For example, an agent that expects a string and a boolean parameter can have the following actual values: `"hello", true`
  - **Name** specifies the name of the command in the user interface. If the name is unspecified the name of the agent will be used as the name of the command.
- You may create a comment on the dependency. This comment describes the agent command and will appear as status bar, or tool tip text in the user interface.
- You may apply the `<<icon>>` stereotype to specify a 16x16 icon for the agent command. This icon will appear in the context menu.

## Using an Agent Command

As soon as you have defined your agent command you can try using it by right-clicking on an element in the Model View that is of the correct kind. Agent commands will appear in the context menu according to the following rules:

- If the agent is a query, the agent command will appear in the **Queries** submenu. When invoked the result of the query will be presented in the Search Result tab, and navigation can be done from that tab as usual.
- For other kinds of agents, the agent command will appear in the **Utilities** submenu.

These submenus will only appear in the context menu if at least one agent command is available for the selected entity.

# Utility Agents

Tau provides numerous utility agents of different kinds exposing important tool functionality to clients. These agents are defined in the libraries some of which are always available, and some of which are loaded when an add-in is activated. The documentation of these agents can also be found in these libraries in the model.

In order to use any of these utility agents (for example when implementing another agent) you should use the `InvokeAgent` API method. Some of the utility agents are also defined to be Agent Commands, which make it possible to use the from the Tau user interface directly.

# 76

# *COM API*

This chapter is the reference documentation of the Tau COM API. Available COM objects and their interfaces are described, as well as the methods and properties of these interfaces.

Intended readers are developers of client applications that use the COM API to access a UML model and/or functionality in the Tau IDE. These client applications could be everything from small interactive Add-Ins to full fledged code generators or import applications. A basic knowledge of COM and C++ is assumed throughout this chapter.

The use of the COM API is only supported for the Windows platforms.

# Introduction

The COM API gives full access to a UML model. It consists of a set of interfaces, each with a number of methods. Some of these methods will only extract information from the model (read API) while others can be used to modify it (write API). It is also possible to build a new model from scratch, and then save it to one or many files.

The COM API also gives access to certain parts of the Tau IDE. This part of the API can for example be used to automate usage of Tau from all environments and technologies supporting COM.

The COM API is can be used by clients running in their own memory space. Such clients are refered to as non-interactive clients since they work on their own private copy of a model, and the information exchange with other applications is therefore typically at file level. However, the COM API can also be used by interactive clients, for example clients that access the model loaded by a running application. This can for example be used to develop an add-in module that accesses the model interactively.

This chapter describes all COM interfaces and their methods. In general there is at least one client usage example for each method. These examples are given in C++, and most of them make use of ATL (Active Template Library) to be as compact as possible.

**Note**

*It should be straight-forward to translate the examples to another COM enabled language, such as C# or Visual Basic.*

The main purpose of the examples is to explain how to use a certain API method. Therefore the examples typically omit error handling (or rather assumes there is an exception handler that deals with them), and are also not always self contained.

**Note**

*For brevity reasons, the type library namespace is also omitted in all references to interfaces, smart pointers etc. For example it is assumed that there is a 'using' clause making the definitions of that namespace accessible without qualifier).*

## Interface overview

The COM API consists of several interfaces each of which provides a set of related functionality. The interfaces are used both by interactive and non-interactive clients.

All interfaces are dual, that is they can be used both from clients that support early binding (C/C++, Java etc.) and from clients that only support late binding (script clients).

There are two main kinds of interfaces in the API:

- Model interfaces
- Utility interfaces

The model interfaces are implemented by classes representing metaclasses in the implementation of the UML Metamodel. Some of the methods in these interfaces hence require knowledge of the metamodel in order to be useful.

The utility interfaces are all other interfaces. They serve as representations for various parts of the Tau toolset that are exposed in the COM API.

## Accessing the API

The way the COM API is accessed depends on whether the client is interactive or non-interactive.

### Accessing the API from non-interactive clients

A non-interactive client accesses the COM API by creating an instance of one of the COM classes that are provided by the API. There are two different COM classes available as shown in the table below:

| COM class | Prog ID | Binary location | Description |
|---|---|---|---|
| `TTD_ModelAccess` | `Telelogic.TauDeveloperModelAPI` | `U2EXTU.DLL` | Instantiate this COM class to access a UML model. This COM class implement the ITtdModelAccess interface. |
| `TTD_StudioAccess` | `Telelogic.TauDeveloperStudioAPI` | `VCS.EXE` | Instantiate this COM class to access features in the Tau IDE (sometimes refered to as Studio). This COM class implements the ITtdStudioAccess interface. |

The programmatic IDs of the COM classes listed in the above table are the version independent ones. There is also a version dependent ID which can be used in case there are more than one version of Tau installed on the machine.

**Note**

*Using a version independent ID on a machine with multiple versions of Tau installed will cause the Tau application that was installed the latest to be used.*

This binary files which contain the COM classes also contain the COM API type libraries.

Another category of non-interactive API clients are Agents that execute in another Tau executable than the Tau IDE (`VCS.EXE`). An agent typically does not need to access the COM API by creating an instance of a COM class. Instead it receives a pointer to an entity within an already loaded model on which it shall operate. It also obtains a server interface which provides services for the agent that are relevant in the executable where it is running. For example, an agent running in the C++ Application Generator obtains the ITtdCppAppGenServer interface which offers some code generation related services to the agent.

### Accessing the API from interactive clients

An interactive COM API client must implement either the ITtdAgent interface (it then becomes an agent) or the ITtdInteractiveClient interface. The former of these interfaces is recommended since it allows the client to be invoked from all other APIs (C++, COM and Tcl), and it also has the advantage of being able to pass actual arguments to the client. See Agents for more information about agents.

The ITtdInteractiveClient interface is only provided for backwards compatibility with clients that were developed before the introduction of the ITtdAgent interface. The ITtdInteractiveClient interface contains a method `OnExecute` which will be called by the Tau application when the interactive client shall be executed. The first argument of the `OnExecute` method is a pointer to an ITtdInteractiveServer interface. This interface represents the Tau application which acts as the server for the interactive client. The second argument of the `OnExecute` method is an ITtdEntities collection of entities. The client can use the COM API on the entities in that collection.

An interactive COM client can be used to implement an Add-In module in a more efficient way than using a Tcl script. However, it is currently not possible to develop the entire Add-In using only a COM client; Tau requires at least a minimal Tcl script to execute. This script can use the Tcl API command `u2::InvokeAgent` (or the older and less general `std::ExecuteCOMClient`) in order to transfer execution to the COM client.

It is sometimes difficult to decide what part of an Add-In to develop in Tcl and what part to let a COM client handle. The following guidelines could be useful when making this decision:

- Use Tcl to hook-up with the user interface of Tau, for example to add new menus. There is currently no COM API available for doing this.
- If the Add-In requires a more sophisticated user interface than a simple dialog, implement it in the COM client.
- If the COM client is developed in a compiled language (for example C++) and runs in the same process as Tau, its performance will be much better than the performance of a corresponding Tcl script.
- Sometimes the COM client must communicate back to Tau during its execution. Examples include writing in an output tab, displaying a diagram etc. This can be done by using the ITtdInteractiveServer::InterpretTclScript method.

# Client restrictions

Although the COM API imposes few restrictions on its clients, there are some important things for a client application to be aware of.

### Bare only

The COM server does not support simultaneous access of the API from multiple threads. Interactive clients will be executed in the main thread of the Tau application.

### Unicode strings only

Strings passed into, or obtained as result from, the methods of the COM API are typed by the BSTR type. BSTR strings are wide (double-byte) Unicode strings on 32-bit Windows platforms. If the COM client does not use Unicode, it must convert strings to Unicode before passing them to the API. In the same way the client must convert the output strings from Unicode to the string encoding used by the client.

# ITtdModelAccess

The ITtdModelAccess interface is the default interface of the TTD_ModelAccess COM class, and is obtained when creating an instance of that class. Contrary to many of the other interfaces of the API, the ITtdModelAccess interface does not correspond directly to a Metaclass of the Metamodel, but is rather used as the entry point of the entire API for all non-interactive clients. Interactive clients normally do not need to use this interface.

ITtdModelAccess contains the following methods:

| | |
|---|---|
| LoadProject | Loads a project (.ttp) into the memory of the client application. Can also be used for creating a new model from scratch. |
| LoadFile | Loads a model file (.u2) into the memory of the client application. Can also be used for creating a new model from scratch. |
| WriteMessage | Writes a message in the environment of the COM client. |
| GetLicense | Requests a run-time license to be taken. |

## LoadProject

Loads a Tau project (extension `ttp`) into the memory of the client application, or creates a new model.

```
HRESULT LoadProject(
          [in] BSTR strProjectPath,
          [out, retval] ITtdModel** ppModel);
```

### Parameters

`[in] strProjectPath`

A string specifying the project file to load. If no such file exists, a new model will be created. If `strProjectPath` is a relative path, it will be interpreted as relative to the current working directory of the client application.

`[out, retval] ppModel`

A pointer to the loaded or created model.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The model of the specified project file is successfully loaded or created. |
| `E_FAIL` | The model of the specified project file could not be loaded or created. This typically happens if a required Telelogic Tau license could not be found, or if there is not enough memory to perform the operation. Use the information provided in the COM error object for more details. |

### Comments

The `LoadProject` method is typically the first method which a non-interactive client calls. It is possible to load or create models for more than one project file by calling `LoadProject` many times. However, be very careful if the same project is loaded more than once. A change in one of the resulting models could overwrite an unsaved change in the other model.

The `LoadProject` method makes an implicit call to ITtdEntity::Bind after the model has been loaded or created, in order to guarantee that all links and references of the model can be navigated directly when the method returns.

**Example 632**

The following example shows how to load a project stored in the project file `MyModel.ttp`, and to create a new model for a non-existing project file `NewModel.ttp`.

```
ITtdModelPtr pITtdModel, pITtdNewModel;
pITtdModel = pITtdModelAccess->LoadProject(_T("MyModel.ttp"));
pITtdNewModel = pITtdModelAccess->LoadProject(_T("NewModel.ttp"));
```

**See also**

"LoadFile" on page 2069

## LoadFile

Loads a UML model data file (extension `u2`) into the memory of the client application. A new model will be created, containing the representation of the contents of the file. If the specified file does not exist, a new model will be created.

```
HRESULT LoadFile(
          [in] BSTR strFileName,
          [in, optional] profile
          [out, retval] ITtdModel** ppModel);
```

### Parameters

`[in] strFileName`

A string specifying the file to load. If this file could not be loaded (for example because it does not exist), a new model will be created. If `strFilePath` is a relative path, it will be interpreted as relative to the current working directory of the client application.

`[in, optional] profile`

Indicates whether the model fragment contained in the file should be loaded as a profile library or not. This parameter is optional and will default to false if omitted.

`[out, retval] ppModel`

A pointer to the loaded or created model.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | The file is successfully loaded. |
| E_OUTOFMEMORY | Not enough memory to create the model representing the contents of the loaded file. |
| E_FAIL | The loading of the file failed, for one reason or another. Use the information provided in the COM error object for more details. |

### Comments

The LoadFile method is an alternative to LoadProject, that can be used to load a model stored in one single file. Similar to LoadProject it can also be used for creating new models. Every call to LoadFile will create a new model representation, which can be accessed through the returned ITtdModel interface pointer.

The model that is returned from LoadFile will contain a Resource representing the file that is loaded (or attempted to be loaded). This Resource can later on be used in order to save changes made in the model to the corresponding file.

The LoadFile method makes an implicit call to ITtdEntity::Bind after the model has been loaded or created, in order to guarantee that all links and references of the model can be navigated directly when the method returns.

**Example 633**

The following example shows how to load a file MyModel.u2.

```
ITtdModelPtr pITtdModel;
pITtdModel = pITtdModelAccess->LoadFile("MyModel.u2");
```

## See also

"LoadProject" on page 2067,

"Bind" on page 2126

"ITtdResource" on page 2142

# WriteMessage

Writes a message to be displayed in the environment of the COM client.

```
HRESULT WriteMessage(
          [in] BSTR strMessage);
```

### Parameters

`[in] strMessage`

The message string. It may contain traditional escape sequences for formatting, such as \n.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The message was successfully written to the environment. |
| `E_FAIL` | Failed to write the message. Use the information provided in the COM error object for more details. |

### Comments

`WriteMessage` can be used by a COM client to write a message in the environment in which the client is executing. For an interactive COM client, the message will be printed in the Messages tab, whereas for a non-interactive COM client, the message will be printed on `stdout`.

**Example 634**

```
pITtdModelAccess->WriteMessage(_T("COM client running!\n"));
```

## GetLicense

Requests a run-time license to be taken. This function is intended to be used by API clients that require a license to be used.

```
HRESULT GetLicense(
          [in] BSTR strFeature);
```

### Parameters

`[in] strFeature`

The name of the license feature that shall be taken.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The license was successfully obtained. |
| `E_FAIL` | Failed to obtain the license. Use the information provided in the COM error object for more details. |

### Comments

`GetLicense` can be used by a COM client which shall have some or all of its functionality available only if one or many license features are available. It can for example be used when developing commercial Add-ins using the COM API for Telelogic Tau.

# ITtdModel

The ITtdModel interface is implemented by the Session class of the Metamodel, which represents the top-level entity of a UML model.

The ITtdModel interface contains methods that do not need a specific model entity context for their execution. Typically these methods operate on the model as a whole, rather than on a particular entity.

**Note**

*Since a Session also is an Entity, a QueryInterface from ITtdModel to ITtdEntity will always succeed.*

ITtdModel contains the following methods:

| | |
|---|---|
| FindByGuid | Finds the entity with the specified GUID. |
| New | Creates a new model entity. An ITtdEntity interface pointer on the created entity is returned.. |
| Parse | Parses a piece of concrete textual UML syntax (U2P) and returns the resulting entities upon success. |
| XMLDecode | Decodes a model fragment encoded as XML and returns the resulting entities upon success.. |
| Save | Saves the model by saving all resources it contains. |
| CreateResource | Creates a new resource in the model. This is the first step in order to save some part of the model in a new file. |
| LoadFile | Loads a UML data file (a .u2 file) into a model. |
| InvokeAgent | Invokes an agent on a specified model context. |

# FindByGuid

Finds the entity with the specified GUID.

```
HRESULT FindByGuid(
          [in] BSTR strGuid,
          [out, retval] ITtdEntity** ppEntity);
```

### Parameters

`[in]  strGuid`

A string containing a GUID.

`[out, retval]  ppEntity`

The entity with the specified GUID, or NULL if no such entity exists in the model.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | Failure due to an internal error. |

### Comments

The `FindByGuid` method can be used to obtain an ITtdEntity pointer on an entity that has a known GUID. Since an entity's GUID remains the same during the entire lifetime of an entity, `FindByGuid` is the appropriate method for locating an entity regardless of its current position in the model.

If the model does not contain an entity with the specified GUID, `ppEntity` will point to NULL after the call.

**Example 635**

The following lines show how to use the `FindByGuid` method in order to access the predefined Boolean datatype:

```
ITtdModelPtr pITtdModel;
pITtdModel = pITtdModelAccess->LoadProject("MyModel.ttp");
ITtdEntityPtr pEntity;
pEntity = pITtdModel->FindByGuid("@Predefined@Boolean");
```

### See also

"GUID" on page 78

## New

Creates a new model entity. An ITtdEntity interface pointer on the created entity is returned.

```
HRESULT New(
          [in] BSTR strMetaClass,
          [out, retval] ITtdEntity** ppEntity);
```

### Parameters

`[in] strMetaClass`

The name of a Metaclass in the Metamodel. The string should specify an existing metaclass, spelled with the correct case. If an incorrect metaclass is specified the method will fail.

`[out, retval] ppEntity`

The created entity.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_OUTOFMEMORY` | Too little memory available to allocate the new entity. |
| `E_FAIL` | The creation failed. Use the information provided in the COM error object for more details. |

### Comments

The `New` method creates an instance of the specified Metaclass, which must exist and be non-abstract.

## Note

*It is the responsibility of the client to take care of the returned entity. It should either be inserted into the model, or deleted, to avoid a memory leak.*

**Example 636**

This example shows how to create a Package as a top-level model entity using `New`, followed by an insertion of the created entity using `ITtdEntity::SetEntity`:

```
ITtdModelPtr pITtdModel;
ITtdEntityPtr pSession = pITtdModel;
pITtdModel = pITtdModelAccess->LoadProject("MyModel.ttp");
ITtdEntityPtr pPackage;
pPackage = pITtdModel->New("Package");
pPackage->SetEntity("Namespace", pSession);
```

### See also

"Create" on page 2117

## Parse

Parses a piece of concrete textual UML syntax (U2P) and returns the resulting entities upon success.

```
HRESULT Parse(
          [in] BSTR strConcreteSyntax,
          [in, optional] VARIANT parseAs,
          [out, retval] ITtdEntities** ppEntities);
```

### Parameters

`[in] strConcreteSyntax`

A string containing a piece of U2P syntax.

`[in, optional] parseAs`

A hint to the parser how it should attempt to parse the provided string. The string should specify one of the metaclasses Definition, Expression, or Action. The default is to try to parse as Definition.

`[out, retval] ppEntities`

The entities resulting from a successful parse.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | The parse failed. Use the information provided in the COM error object for more details. |

### Comments

The `Parse` method is useful for creating model fragments from textual syntax descriptions. This is often more convenient than creating the entities one by one (using `ITtdModel::New` or `ITtdEntity::Create`) and connecting them (using `ITtdEntity::SetEntity`). Also it requires much less Metamodel knowledge, since the parser takes care of creating the model.

## Note

*It is the responsibility of the client to take care of the returned entities. They should either be inserted into the model, or deleted, to avoid a memory leak.*

**Example 637**

This example shows how to create a Class with an Attribute in the Package created in Example 636 on page 2076 using the `Parse` method. The parser is told to parse the provided string as a Definition, which is the default behavior.

```
ITtdEntitiesPtr pParseResult;
pParseResult = pITtdModel->Parse("class C { public Integer a; };");
ITtdEntityPtr pClass;
pClass = pParseResult->GetItem(1);
pPackage->SetEntity("OwnedMember", pClass);
```

## See also

"Unparse" on page 2109

# XMLDecode

Decodes a model fragment encoded as XML and returns the resulting entities upon success.

```
HRESULT XMLDecode(
          [in] BSTR strXMLEncoding,
          [out, retval] ITtdEntities** ppEntities);
```

### Parameters

`[in] strXMLEncoding`

A string containing a model fragment encoded in XML.

`[out, retval] ppEntities`

The entities resulting from a successful XML decoding.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | The decoding failed. Use the information provided in the COM error object for more details. |

### Comments

The `XMLDecode` method can be used to create a model fragment from an XML encoding. The XML encoding is typically obtained as the result of calling `ITtdEntity::XMLEncode`, but could also be obtained by other means, for example by reading the content of a `.u2` file.

**Note**

*It is the responsibility of the client to take care of the returned entities. They should either be inserted into the model, or deleted, to avoid a memory leak.*

**Example 638**

This example shows how to encode the Package created in Example 636 on page 2076 as XML using the `ITtdEntity::XMLEncode` method, and then how to decode that XML back to a Package using `XMLDecode`:

```
CComBSTR bstrXMLEncoding((TCHAR*) pPackage->XMLEncode());
ITtdEntitiesPtr pRoots;
pRoots = pITtdModel->XMLDecode((BSTR) bstrXMLEncoding);
```

## Save

Saves the model by saving all resources it contains.

```
HRESULT Save();
```

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The model was successfully saved. |
| `E_FAIL` | At least one resource in the model could not be saved. Use the information provided in the COM error object for more details. |

### Comments

Use the `Save` method to save all changes that have been made to the model. This method is really nothing more than a convenience method, since the same thing can be done by iterating over all resources in the model, and calling ITtdResource::Save on each of them.

An example of how to use the `Save` method is in Example 639 on page 2080.

## CreateResource

Creates a new resource in the model. This is the first step in order to save some part of the model in a new file.

```
HRESULT CreateResource(
          [in] BSTR strFileName,
          [out, retval] ITtdResource** ppResource);
```

### Parameters

`[in] strFileName`

The name of the file for the resource.

`[out, retval] ppResource`

The created resource.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The resource was successfully created. |
| `E_OUTOFMEMORY` | Too little memory available to allocate the new resource. |
| `E_FAIL` | Failure due to an internal error. |

### Comments

The `CreateResource` method is the recommended way to create a new resource in the model. Another way is to use the more general (but in this case less appropriate) `ITtdEntity::Create` with the Resource Metaclass as argument. The drawback with that approach is that it will display a modal dialog prompting for the resource filename. This is not always the wanted behavior, since the client may have obtained the filename already by some other means.

**Example 639**

This example uses `CreateResource` to save the Package created in Example 636 on page 2076 in a file of its own:

```
ITtdEntityPtr pResource;
pResource = pITtdModel->CreateResource("D:\\temp\\COMtest.u2");

// Insert the created package pPackage as a root of pResource
pResource->SetEntity("Root", pPackage);
pITtdModel->Save(); // Saves all resources in the model
```

---

# LoadFile

Loads a UML data file (a `.u2` file) into a model.

```
HRESULT LoadFile(
          [in] BSTR strFileName,
          [in, optional] VARIANT profile,
          [out, retval] ITtdResource** ppResource);
```

### Parameters

`[in] strFileName`

A string specifying the file to load. If this file could not be loaded (for example because it does not exist), the error code E_FAIL is returned. If `strFilePath` is a relative path, it will be interpreted as relative to the current working directory of the client application.

`[in, optional] profile`

Indicates whether the model fragment contained in the file should be treated as a profile library or not. This parameter is optional and will default to false if omitted.

`[out, retval] ppResource`

A pointer to the resource of the loaded file. If `profile` above is true, this pointer will be set to NULL.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | The file is successfully loaded. |
| E_OUTOFMEMORY | Not enough memory to create the resource representing the contents of the loaded file. |
| E_FAIL | The loading of the file failed, for one reason or another. Use the information provided in the COM error object for more details. |

### Comments

The LoadFile method can be used to load additional model fragments, stored in single files, into a model. A common usage of this method is when a profile stored in a separate file, and used by the model in question, needs to be loaded. However, also regular model segments can be loaded. Use the argument profile to control whether the model fragment to load should be treated as a profile library or not.

LoadFile returns a Resource representing the file that was loaded. This Resource can later on be used in order to save changes made in the model to the corresponding file.

The LoadFile method makes an implicit call to ITtdEntity::Bind after the file has been loaded, in order to guarantee that all links and references of the model can be navigated directly when the method returns.

**Example 640**

The following example shows how to load a project and a profile definition, stored in the project file MyProj.ttp and file MyProfile.u2 respectively. Note that the profile in this case will be loaded as an ordinary U2 file (i.e. it will not be loaded into the Libraries section of the model).

```
ITtdModelPtr pITtdModel;
ITtdResourcePtr pITtdResource;
pITtdModel = pITtdModelAccess->LoadProject(_T("MyProj.ttp"));
pITtdResource = pITtdModel->LoadFile(_T("MyProfile.u2"));
```

## See also

"LoadProject" on page 2067

"ITtdResource" on page 2142

## InvokeAgent

Invokes an agent on a specified model context.

```
HRESULT InvokeAgent(
          [in] ITtdEntity* agent,
          [in] ITtdEntity* modelContext,
          [in, out, optional] VARIANT* agentParameters);
```

### Parameters

`[in] agent`

The definition of the agent to invoke.

`[in] modelContext`

The entity that is the model context of the agent invocation. The agent will perform its work in the context of this entity.

`[in, out, optional] agentParameters`

An optional list (safe array) of actual agent parameters.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | The agent is successfully invoked. |
| E_FAIL | The agent could not be invoked, for one reason or another. Use the information provided in the COM error object for more details. |

### Comments

The `InvokeAgent` method can be used to invoke an agent programmatically as an alternative to specifying statically how it shall be invoked by means of ordering dependencies to tool events. The method has many similarities with the `ExecuteCOMClient` Tcl command, but is more general since the invoked agent does not need to be implemented in COM.

If the invoked agent expects actual arguments, these should be specified as a safe array of variants. The variants may have the following types:

- A COM interface pointer (`IUnknown`). Either `ITtdEntity` or `ITtdEntities` must be obtainable from the passed interface.
- A string (BSTR)
- An integer (long)
- A boolean (VARIANT_BOOL)

The agent parameters list is an in/out parameter. Thus it is possible for the agent to modify the agent parameters in order to pass information back to the caller. The caller should not assume anything about the actual parameters after the agent has been invoked. The agent may have changed both the number of actual arguments, and their types.

**Example 641**

The following example shows how to execute an agent whose definition has a known GUID. The model context is any entity from the model. No actual arguments are passed to the agent.

```
ITtdEntityPtr pAgent;
pAgent = pITtdModel->FindByGuid(_T("MyAgentGuid"));
pITtdModel->InvokeAgent(pAgent, pModelContext);
```

### See also

"Agents" on page 2025 in Chapter 75, *Agents*

ITtdAgent::Execute

# ITtdEntity

The ITtdEntity interface is implemented by the Entity class of the Meta-model, which represents a general entity of a UML model.

The ITtdEntity interface contains methods that need a specific model entity context for their execution. Typically these methods operate on the entity on which they are called.

| Method | Description |
|---|---|
| ApplyStereotype | Instantiates the given stereotype and applies it on an entity. |
| Bind | Binds all references in a model fragment, or a single reference of an entity. |
| Clone | Creates a clone of an entity. |
| Create | Creates a new entity in the context of an entity, that is adds a new direct or indirect child to an entity. |
| CreateInstance | Creates an instance of a Signature. |
| Delete | Deletes an entity from the model. |
| FindByName | Performs a name-lookup from the context of an entity to find another entity by its (qualified) name. |
| GetContainerMetaFeature | Returns the name of the metafeature in which an entity is contained. |
| GetDescriptiveName | Returns a description of an entity. |
| GetEntities | Returns the value of a metafeature for an entity as a collection of entities. |
| GetEntity | Returns the value of a metafeature for an entity as an entity. |
| GetMetaClassName | Returns the name of the entity's Metaclass. |
| GetModel | Returns the model to which an entity belongs. |
| GetOwner | Returns the composition owner of an entity. |
| GetReference | Returns the identifier of a metafeature representing a reference. |
| GetReferringEntities | Returns a collection of entities that refer to an entity through a particular metafeature (or through any metafeature). |

**ITtdEntitiy methods**

| Method | Description |
|---|---|
| GetTaggedValue | Returns the specified property (tagged value) of an element. Can also be used to obtain an arbitrary value from an instance representation. |
| GetValue | Returns the value of a metafeature for an entity as a string. |
| HasAppliedStereotype | Determines if a certain stereotype is applied on an element. |
| IsKindOf | Determines if an entity is of a particular metaclass kind. |
| MetaVisit | Traverses a model fragment and calls a method in a callback interface for each entity it contains. |
| MetaVisitEx | Extended version of MetaVisit which allows the traversal to include references. |
| Move | Moves an entity from its current location in the model to another owner. |
| Replace | Replaces an entity with another entity. |
| SetEntity | Sets the value of a metafeature for an entity as an entity. |
| SetTaggedValue | Sets a property (tagged value) on an element. Can also be used to set an arbitrary value of an instance representation. |
| SetValue | Sets the value of a metafeature for an entity as a string. |
| UnlinkFromOwner | Unlinks an entity from it current owner in the model. |
| Unparse | Unparse of an entity into a concrete syntax representation. |
| XMLEncode | Encodes an entity into an XML representation. |

**ITtdEntitiy methods**

## ApplyStereotype

Instantiates the given stereotype and applies it on an entity. The qualifier in the reference to the stereotype is calculated based on the `referenceKind` (see below). If `pInsertElement` is given the stereotype is logically instantiated on the host entity, but physically instantiated on the `pInsertElement`. In this case the stereotype instance will point to the host entity. In this way, stereotype instances may be “applied” to an entity but not modify the entity itself.

```
HRESULT ApplyStereotype(
          [in] ITtdEntity* stereotype,
          [in] TtdReferenceKind referenceKind,
          [in, optional] VARIANT insertElement,
          [out, retval] ITtdEntity** result);
```

### Parameters

`[in]` `stereotype`

The stereotype to instantiate on the entity.

`[in]` `referenceKind`

The qualifier in the reference to the stereotype is calculated based on the `referenceKind`.

| TtdReferenceKind | Description |
|---|---|
| TTD_RK_GUID | The reference will only contain a GUID reference. |
| TTD_RK_NO_QUALIFIER | The reference will not be qualified. That is, it will only contain the name of the stereotype. |
| TTD_RK_FULL_QUALIFIER | The reference will contain a full qualifier. |
| TTD_RK_MINIMAL_QUALIFIER | The reference will contain the minimum qualifier needed to reference the stereotype. This option may at most return the same qualifier as TTD_RK_RELATIVE_QUALIFIER. Note: the presence of <<access>>/<<import>> dependencies may make it shorter. |
| TTD_RK_RELATIVE_QUALIFIER | The reference will be a relative qualifier to the stereotype. If there are no common upper scopes of the stereotype and the host entity, a full qualifier is calculated, otherwise a shorter qualifier starting from the nearest common scope is calculated. |

For Tcl users, there is a shorter version of the `referenceKind` values. These shorter values must be used with the Tcl API. The following table lists the values used in the Tcl API.

| COM | Tcl |
|---|---|
| TTD_RK_GUID | GUID |
| TTD_RK_NO_QUALIFIER | noQualifier |
| TTD_RK_FULL_QUALIFIER | fullQualifier |
| TTD_RK_MINIMAL_QUALIFIER | minimalQualifier |
| TTD_RK_RELATIVE_QUALIFIER | relativeQualifier |

**Note**

*The value* `minimalQualifier` *is recommended for most uses.*

`[in, optional]` `insertElement`

If `insertElement` is given, the stereotype is logically instantiated on the host entity, but physically instantiated on the `insertElement`. In this case the stereotype instance will point to the host entity. In this way, stereotype instances may be “applied” to an entity but not modify the entity itself. This technique is known as “stereotype injection”.

`[out, retval]` `result`

The instantiated stereotype (i.e a stereotype instance).

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | Failure due to an internal error. |

## GetValue

Returns the value of a metafeature for the entity on which the method is called. The value is represented as a string.

```
HRESULT GetValue(
          [in] BSTR strMetaFeature,
          [in, optional] VARIANT index,
          [out, retval] BSTR* strValue);
```

### Parameters

`[in]` `strMetaFeature`

A string specifying the metafeature from which the value shall be extracted. The string should specify an existing metafeature for the entity on which the method is called, spelled with the correct case. If an incorrect metafeature is specified the method will fail.

`[in, optional]` `index`

The index of the metafeature from which the value shall be extracted. Indexes start at 1 and the index 0 can be used to specify the last index of the metafeature. If this optional parameter is omitted it will default to 0. If an index is specified it must be within a valid range, otherwise the method will fail.

`[out, retval]` `strValue`

The obtained value represented as a string. Details on the format of the string can be found in the table in the **Comments** section.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | The specified value could not be obtained for one reason or another. Use the information provided in the COM error object for more details. |

### Comments

The `GetValue` method can be used on all metafeatures of an entity that can have their values encoded as a string. This is the case for all metafeatures except derived features of Metaclass type, owner links and composition links.

The format of the string returned by `GetValue` depends on the type of the specified metafeature. The table below shows the different possibilities.

| Metafeature type | String encoding | Example |
|---|---|---|
| Enumeration (datatype with literals) | The name of the literal | `"true"`<br>`"VkVirtual"` |
| Charstring or identifier (`CeIdent`) | The string or identifier | `"MyClass"` |
| Numeric type (Integer, Real, Natural etc.) | The numeric value as a string | `"14"`<br>`"217.5"` |
| GUID (`CeGuid`) | The GUID | `"63Lkm3hRE7K*uRE"` |
| Position or size type (`SPoint` or `SSize`) | The X and Y coordinates separated with a space | `"1240 1380"` |
| List of positions (`PointVector`) | The encoding of each position separated with a space | `"1940 1020 1940 1120"` |

| Metafeature type | String encoding | Example |
|---|---|---|
| Metaclass | If the metafeature has no assigned value: an empty string | `""` |
| | If the metafeature is set up by GUID or directly by pointer: the string `"uid:"` followed by the GUID of the target entity. | `"uid:63Lkm3hRE7K*uRE"` |
| | If the metafeature is set up by name: the string `"ref:"` followed by the name of the target entity (possibly qualified). | `"ref:MyClass::MyAttr"` |

**Example 642**

The following example uses the `GetValue` method in order to access the name of the predefined Boolean datatype:

```
ITtdModelPtr pITtdModel;
pITtdModel = pITtdModelAccess->LoadProject("MyModel.ttp");
ITtdEntityPtr pEntity;
pEntity = pITtdModel->FindByGuid("@Predefined@Boolean");
CComBSTR bstrName((TCHAR*) pEntity->GetValue("Name"));
```

## See also

"GetEntity" on page 2093,

"GetTaggedValue" on page 2103

"Metamodel Classes" on page 391.

# GetEntity

Returns the value of a metafeature for the entity on which the method is called. The value is represented as an ITtdEntity pointer.

```
HRESULT GetEntity(
          [in] BSTR strMetaFeature,
          [in, optional] VARIANT index,
          [out, retval] ITtdEntity** ppEntity);
```

### Parameters

`[in] strMetaFeature`

A string specifying the metafeature from which the value shall be extracted. The string should specify an existing metafeature for the entity on which the method is called, spelled with the correct case. If an incorrect metafeature is specified the method will fail.

`[in, optional] index`

The index of the metafeature from which the value shall be extracted. Indexes start at 1 and the index 0 can be used to specify the last index of the metafeature. If this optional parameter is omitted it will default to 0. If an index is specified it must be within a valid range, otherwise the method will fail.

`[out, retval] ppEntity`

The obtained value represented as an ITtdEntity pointer. If no value exists, this pointer will be NULL.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | Success. |
| E_FAIL | The specified value could not be obtained for one reason or another. Use the information provided in the COM error object for more details. |

### Comments

The GetEntity method can be used on all metafeatures of an entity that have Metaclass type. For these metafeatures this method is often more convenient to use than GetValue since the obtained ITtdEntity pointer can be used directly in subsequent method calls. However, if the entity pointer obtained from GetEntity is NULL, it could be interesting to make a corresponding call to GetValue in order to know if this means that the metafeature could be bound at a later point in time, or if no value has ever been given to the metafeature. In the latter case GetValue would return an empty string, while in the former case the returned string would be the encoded metafeature value, although the metafeature is (currently) unbound.

### Note

*If the metafeature represents a reference in the Metamodel, GetReference could be a better alternative than* GetValue.

**Example 643**

The following example uses the GetEntity method in order to access the type of an attribute pointed to by pAttribute. If its type metafeature is bound to a Definition, the name of that Definition is extracted:

```
ITtdEntityPtr pType;
pType = pAttribute->GetEntity("Type");
if (pType != 0 && pType->IsKindOf("Definition"))
  CComBSTR bstrName((TCHAR*) pType->GetValue("Name"));
```

### See also

"GetValue" on page 2089,

"GetEntities" on page 2095,

"GetReference" on page 2096

## GetEntities

Returns the value of a metafeature for the entity on which the method is called. The value is represented as a collection of entities, that is as an ITtdEntities pointer.

```
HRESULT GetEntities(
          [in] BSTR strMetaFeature,
          [out, retval] ITtdEntities** ppEntities);
```

### Parameters

`[in] strMetaFeature`

A string specifying the metafeature from which the value shall be extracted. The string should specify an existing metafeature for the entity on which the method is called, spelled with the correct case. If an incorrect metafeature is specified the method will fail.

`[out, retval] ppEntities`

A pointer to an ITtdEntities collection containing the entities being the value of the metafeature.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | The specified value could not be obtained for one reason or another. Use the information provided in the COM error object for more details. |

### Comments

The `GetEntities` method can be used on all metafeatures of an entity that have Metaclass type. It is often a more convenient alternative to making several consecutive calls to `GetEntity` with different indexes, in order to obtain

all entities of a metafeature with non-single multiplicity. However, it can also be used on metafeatures with single multiplicity, in which case the obtained collection only will contain one or zero entities.

The returned collection may include NULL pointers, if the metafeature is unbound at the corresponding index. Use GetValue or GetReference to find out more information about such unbound metafeatures.

**Example 644**

The following example uses the `GetEntities` method in order to access the classes (and other signatures) contained in a package pointed to by `pPackage`:

```
ITtdEntitiesPtr pSignatures;
pSignatures = pPackage->GetEntities("Signature");
long lCount = pSignatures->Count;
ITtdEntityPtr pSignature;
// N.B. Index are 1-based
for (int i = 1; i <= lCount; i++){
  pSignature = pSignatures->GetItem(i);
  // Do something with pSignature...
}
```

**See also**

"GetEntity" on page 2093

## GetReference

Returns the reference of a metafeature for the entity on which the method is called. The metafeature should represent a (non-derived) reference in the Metamodel.

```
HRESULT GetReference(
          [in] BSTR strMetaFeature,
          [in, optional] VARIANT index,
          [out, retval] ITtdEntity** ppEntity);
```

### Parameters

`[in]  strMetaFeature`

A string specifying the metafeature from which the reference shall be extracted. The string should specify an existing metafeature for the entity on which the method is called, spelled with the correct case, and the metafeature should represent a (non-derived) reference in the metamodel. If an incorrect metafeature is specified the method will fail.

`[in, optional] index`

The index of the metafeature from which the reference shall be extracted. Indexes start at 1 and the index 0 can be used to specify the last index of the metafeature. If this optional parameter is omitted it will default to 0. If an index is specified it must be within a valid range, otherwise the method will fail.

`[out, retval] ppEntity`

An ITtdEntity pointer to the obtained reference. The reference is represented by an identifier (an instance of the `Ident` Metaclass). If no reference exists, this pointer will be NULL. This happens when the reference is not set up by identifier (but by GUID or by pointer), or when it has never been set up at all.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | The reference could not be obtained for one reason or another. Use the information provided in the COM error object for more details. |

### Comments

The `GetReference` method can be used on all metafeatures that represent non-derived references in the Metamodel. This is the case for all metafeatures of metaclass type, except owner and composition links.

The returned reference is represented as an identifier in the model. This identifier has a name, and possibly also additional information that is needed in order to bind it (for example a scope qualifier).

Since a reference can be fairly complex in the general case, `GetReference` is often a better alternative than `GetValue` for metafeatures that represent references. The reason is that the result is a model representation of the identifier, rather than a string representation.

**Example 645**

The following example uses the `GetReference` method in order to access the reference of the type metafeature of an attribute pointed to by `pAttribute`. The returned reference is then unparsed, using the `Unparse` method.

```
ITtdEntityPtr pRef;
pRef = pAttribute->GetReference("Type");
if (pRef != NULL){
    CComBSTR bstrText((TCHAR*) pRef->Unparse());
}
```

**See also**

"GetValue" on page 2089

## GetOwner

Returns the composition owner of the entity on which the method is called.

```
HRESULT GetOwner(
          [out, retval] ITtdEntity** ppEntity);
```

### Parameters

`[out, retval]  ppEntity`

A pointer to the entity that is the direct owner of the entity on which the method is called, or NULL if the entity has no owner.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |

### Comments

`GetOwner` is the recommended method to use in order to find the owner of an entity. An alternative technique for finding the owner is to call `GetEntity` for the owner metafeature. However, if an entity has more than one owner metafeature, it is easier to call `GetOwner` than to make one `GetEntity` call for each owner metafeature.

**Example 646**

This example shows how `GetOwner` can be used to find the Session of an entity pointed to by `pEntity`. Remember that the Session is the outermost entity of a model:

```
ITtdEntityPtr pOwner = pEntity;
ITtdEntityPtr pSession;
while (pOwner != 0){
    pSession = pOwner;
    pOwner = pOwner->GetOwner();
}
// pSession now points at the Session of pEntity (provided pEntity
// is part of the model of course)
```

It should be noted that a much easier way to get to the Session, is to use GetEntity on the derived metafeature “Session” available on all entities, or to use GetModel.

---

### See also

“GetEntity” on page 2093

## GetMetaClassName

Returns the name of the Metaclass of the entity on which the method is called.

```
HRESULT GetMetaClassName(
          [out, retval] BSTR* strName);
```

### Parameters

`[out, retval]  strName`

The name of the entity's metaclass.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |

### Comments

`GetMetaClassName` is useful for finding out what kind of entity an ITtdEntity pointer is referring to. Often it is enough to use `IsKindOf` to find out if an entity is of a particular metaclass kind, but sometimes it is needed to find out the precise metaclass of an entity.

**Example 647**

This example illustrates the difference in how `GetMetaClassName` and `IsKindOf` can be used to distinguish what kind of entity an ITtdEntity pointer is pointing to:

```
CComBSTR bstrMetaClass((TCHAR*) pEntity->GetMetaClassName());
if (bstrMetaClass == "Operation"){
  // Will only get here if pEntity's metaclass is Operation,
  // that is  not if pEntity is a StateMachine
}
if (pEntity->IsKindOf("Operation")){
  // Will get here if pEntity is an Operation (including
  // StateMachine since the StateMachine metaclass inherits
  // the Operation metaclass)
}
```

**See also**

"IsKindOf" on page 2108

## GetReferringEntities

This is the same function as `GetReferingEntities`. The spelling of the function as `GetReferingEntities` is deprecated, and is present in the API to ensure backward compatibility.

Returns a collection of entities that refer to the entity on which the method is called, through a particular metafeature.

```
HRESULT GetReferringEntities(
          [in] BSTR strMetaFeature,
          [out, retval] ITtdEntities** ppEntities);
```

### Parameters

`[in] strMetaFeature`

A string specifying the metafeature through which the entity is referenced. This is a metafeature on the referring entity, not on the entity on which the method is called (which is the referred-to entity). The string should specify an existing metafeature spelled with the correct case. If an incorrect metafeature is specified, an empty collection will be the result.

The string may be empty to find all referring entities, regardless of through which metafeature they refer to the entity.

`[out, retval] ppEntities`

A pointer to an ITtdEntities collection containing the entities that refer to the entity through the specified metafeature.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | Success. |

### Comments

The purpose of the GetReferringEntities method is to find so called reverse references. When a reference corresponding to a certain metafeature in the model is bound, there will also be a reverse reference established from the referred-to entity to the referring entity. The GetReferringEntities gives access to the target entities of these reverse references, that is the referring entities.

Figure 285 on page 2102 illustrates the concept of a reverse reference. It shows an Attribute typed by a Class. To navigate from the Attribute to its type, use the GetEntity method for the "type" metafeature. However, to navigate from the type to the Attribute, GetReferringEntities must be used instead.

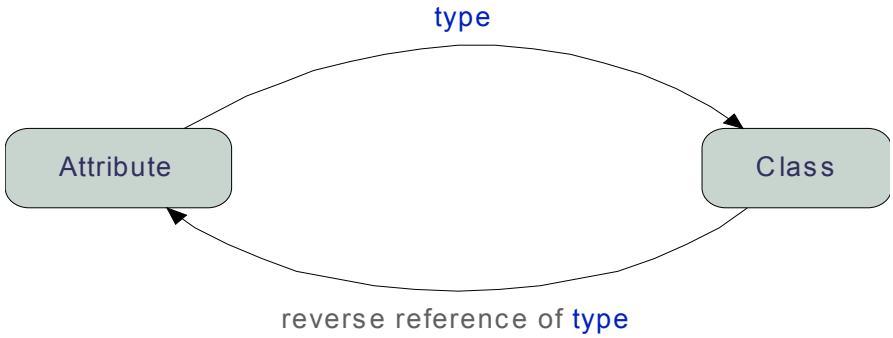



*Figure 285: A reverse reference*

**Example 648**

The following example uses the GetReferringEntities method in order to find all entities typed by the predefined Boolean type:

```
ITtdModelPtr pITtdModel;
pITtdModel = pITtdModelAccess->LoadProject("MyModel.ttp");
ITtdEntityPtr pEntity;
pEntity = pITtdModel->FindByGuid("@Predefined@Boolean");

ITtdEntitiesPtr pReferringEntities;
pReferringEntities = pEntity->GetReferringEntities("Type");
// pReferringEntities now contains all entities typed by Boolean
```

## GetTaggedValue

Returns a property (tagged value) of an element, or in general any value from an instance representation.

```
HRESULT GetTaggedValue(
          [in] BSTR strSelector,
          [in, optional] VARIANT interpretIdentsAsGuids,
          [out, retval] ITtdEntity** ppValue);
```

### Parameters

`[in] strSelector`

A string containing a selector pattern that specifies which (tagged) value to retrieve. The format of this pattern is described below.

`[in, optional] interpretIdentsAsGuids`

Indicates whether identifiers in the selector pattern should be interpreted as GUIDs rather than as ordinary names. This parameter is optional and will default to false if omitted.

`[out, retval] ppValue`

The (tagged) value selected by the selector pattern, or NULL if no such value exists (either because the specified value does not exist, or that the selector pattern is not well-formed). The value is an Expression.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | Failure due to an internal error. |

### Comments

`GetTaggedValue` is a convenience method for extracting properties (tagged values) from an element. Since properties are represented as ordinary values in the model, the method can also be used on any instance, represented in the model with the InstanceExpr Metaclass.

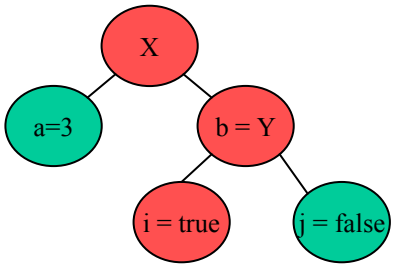
If the entity on which the method is called is not an `InstanceExpr` nor an entity that can have stereotypes applied, NULL will be returned.

Of course, properties can be extracted just by using the `GetEntity` method (and by knowing exactly how values are represented in the model). However, `GetTaggedValue` offers several benefits to that approach:

- Patterns can select values at any depth in the "tree of values" represented by an `InstanceExpr`.
- One call to `GetTaggedValue` can replace several calls to `GetEntity`, making the client code much more compact and readable.
- Patterns support Signature inheritance, that is it is possible to match instances of several inherited Signatures using a pattern that specify a common parent Signature.
- If the selected value is missing in the `InstanceExpr`, but the corresponding Attribute has a default value, that value will be returned.
- The method also handles values from stereotypes that are automatically applied due to the presence of an non-optional extension for a matching metaclass.
- Some special cases in the model representation of values are taken care of by the `GetTaggedValue` implementation, relieving the client from having to take these into consideration.

The parameter `interpretIdentsAsGuids` should be used to avoid the risk that a pattern selects a property (tagged value) in the wrong stereotype instance. It is typically useful for a client that extracts properties for stereotypes in a profile package which the client is responsible for. If the client knows the GUID of the stereotype it wants to access properties from, it can use a selector pattern with GUIDs rather than plain names. Example 649 on page 2105 shows an example.

The syntax of the selector pattern is the textual UML syntax (U2P) for instance expressions, but restricted so that only identifiers, instance expressions, and assignments are allowed. Also, there must be at most one assignment in each instance expression. The result is a pattern that specifies a path down through the “tree of values”, selecting the value at the end of the path.




X(. b = Y(. i .) .)

*Figure 286: Selector pattern example*

Figure 286 on page 2105 shows how a pattern selects the property (tagged value) of Y::i in an instance of some stereotype X.

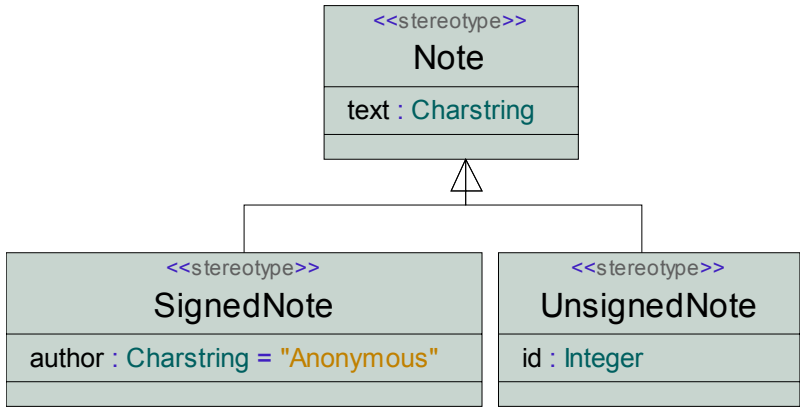



*Figure 287: Stereotypes describing notes*

**Example 649: Usages of GetTaggedValue.**

The model shown in Figure 287 on page 2105 is used.

Assume there is an element pointed to by `pElement`. The following lines will extract the SignedNote::author property of that element:

```
ITtdEntityPtr pTaggedValue;
pTaggedValue = pElement->GetTaggedValue("SignedNote (. author .)");
```

Since SignedNote::author has a default value (“Anonymous”), `pTaggedValue` above will point to a CharstringValue although no explicit value is provided for the author attribute in the SignedNote instance.

In the following code the pattern “`Note (. text .)`” is used to extract the tagged value “`text`” on both SignedNotes and UnsignedNotes:

```
ITtdEntityPtr pTaggedValue;
pTaggedValue = pElement->GetTaggedValue("Note (. text .)");
```

The following lines contain a simple pattern for determining if the UnsignedNote stereotype is applied on the element:

```
ITtdEntityPtr pTaggedValue;
pTaggedValue = pElement->GetTaggedValue("UnsignedNote (. .)");
```

If UnsignedNote is applied on the element, `pTaggedValue` will point to the InstanceExpr representing the stereotype instance. If not, it will be NULL. However, please read the note below concerning the HasAppliedStereotype method which is the best method to use for testing if a certain stereotype is applied.

If the GUID of the UnsignedNote stereotype is known to be `@UnsignedNote`, the `interpretIdentsAsGuids` parameter can be used to avoid getting an instance of another stereotype with the same name. The GUID has to be within apostrophes to make the pattern syntactically correct. If UnsignedNote is applied on the element, it is then possible to make another call to `GetTaggedValue` on the returned stereotype instance. In that call the pattern does not need to contain GUIDs since you now know that you are working with the correct InstanceExpr.

```
ITtdEntityPtr pTagValue;
pTagValue = pElement->GetTaggedValue("'@UnsignedNote' (. .)", true
/*idents as GUIDs*/);
if (pTaggedValue != NULL){
  ITtdEntityPtr pIdValue;
  pIdValue = pTagValue->GetTaggedValue("UnsignedNote (. id .)");
}
```

---

**Note**

*Although a selector pattern on the form "*`X (. .)`*" can be used to test if a stereotype X is applied on an entity, it is better to use HasAppliedStereotype for this purpose. The reason is that a stereotype that is automatically applied (due to a non-optional extension on a matching metaclass) will not be instantiated until at least one of its attributes get a tagged value that differs from the default value of the attribute.GetTaggedValue thus has no stereotype instance to return for the particular case. However, when used with a selector pattern that selects a tagged value, GetTaggedValue will also consider such automatically applied stereotypes.*

**See also**

"GetEntity" on page 2093,

"SetTaggedValue" on page 2114,

"HasAppliedStereotype" on page 2107

## HasAppliedStereotype

Determines if an element has a certain stereotype applied or not.

```
HRESULT HasAppliedStereotype(
          [in] BSTR strStereotype,
          [in, optional] VARIANT guid,
          [out, retval] VARIANT_BOOL* result);
```

### Parameters

`[in] strStereotype`

A string specifying which stereotype to look for among the applied stereotypes of the entity. If guid is false this string should be the name of the stereotype. Otherwise it should be the GUID of the stereotype.

`[in, optional] guid`

Indicates whether strStereotype should be interpreted as the GUID rather than as the name of the stereotype to look for. This parameter is optional and will default to false if omitted.

`[out, retval] result`

True if the specified stereotype is applied on the element. False otherwise.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | Failure due to an internal error. |

### Comments

`HasStereotypeApplied` is the recommended method for checking for applied stereotypes on an entity. It will consider both explicitly applied stereotypes, and stereotypes that are automatically applied due to non-optional extensions from a metaclass that matches the metaclass of the entity

### See also

"GetTaggedValue" on page 2103

## IsKindOf

Determines if the entity on which the method is called is of a particular Metaclass kind.

```
HRESULT IsKindOf(
          [in] BSTR strMetaClass,
          [out, retval] VARIANT_BOOL* isKindOf);
```

### Parameters

`[in] strMetaClass`

The name of a metaclass in the Metamodel.

`[out, retval] isKindOf`

True if the entity on which the method is called is of the specified metaclass kind, false otherwise.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | Failure due to an internal error. |

### Comments

The `IsKindOf` method returns true if one of the following conditions are true:

1. The entity's metaclass is the metaclass specified in the strMetaClass parameter.
2. The entity's metaclass inherits the metaclass (directly or indirectly) specified in the strMetaClass parameter.

Note the difference with the GetMetaClassName method which can be used to check if only the first of these conditions apply. Example 647 on page 2100 shows an example using `IsKindOf`.

### See also

"GetMetaClassName" on page 2099

## Unparse

Unparse of the entity on which the method is called into a concrete syntax representation.

```
HRESULT Unparse(
          [out, retval] BSTR* strConcreteSyntax);
```

### Parameters

`[out, retval]  strConcreteSyntax`

The concrete syntax representation of the entity on which the method is called.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | Unparse failed. Use the information provided in the COM error object for more details. |

### Comments

The `Unparse` method can unparse most models and model fragments. In particular it can unparse all model entities obtained from calling Parse. There are, however, some kinds of entities that cannot be unparsed on their own since they require a model context in order to be unambiguously unparsed. If `Unparse` is called on such an entity, it will fail.

**Example 650**

This example shows how to unparse the Class created using Parse in Example 637 on page 2077:

```
CComBSTR bstrText((TCHAR*) pClass->Unparse());
// bstrText will become "class C { public Integer a; };" (perhaps
// formatted slightly differently)
```

## See also

“Parse” on page 2076

## SetValue

Sets the value of a metafeature for the entity on which the method is called. The value is represented as a string.

```
HRESULT SetValue(
          [in] BSTR strMetaFeature,
          [in] BSTR strValue,
          [in, optional] VARIANT index);
```

### Parameters

`[in] strMetaFeature`

A string specifying the metafeature which should have its value set. The string should specify an existing metafeature for the entity on which the method is called, spelled with the correct case. If an incorrect metafeature is specified the method will fail.

`[in] strValue`

The value to set, represented as a string. The format of the string is described in the documentation of the GetValue method.

`[in, optional] index`

The index of the metafeature which should have its value set. The value will be inserted before the value that is currently at that index. Indexes start at 1 and the index 0 can be used to specify the last index of the metafeature. If this optional parameter is omitted it will default to 0. If the index is outside a valid range, the value will be inserted at the last position in the metafeature.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | Success. |
| E_FAIL | The value could not be set for one reason or another. Use the information provided in the COM error object for more details. |

### Comments

The SetValue method can be used on all writable metafeatures of an entity that can have their values encoded as a string. This is the case for all metafeatures except derived features (which are read-only), owner links and composition links.

The format of the string is described in the documentation of the GetValue method.

**Example 651**

This example shows how to create an Attribute in the context of pClass and then give it the name "CreatedAttribute" using the SetValue method:

```
ITtdEntityPtr pCreatedEntity;
pCreatedEntity = pClass->Create(_T("Attribute"));
if (pCreatedEntity){
    pCreatedEntity->SetValue("Name", "CreatedAttribute");
}
```

A slightly more advanced use of SetValue is to set up a reference in the model by an identifier. In the example below this is done for the type of the attribute created above. In order to have the reference bound, so that GetEntity can be used to navigate to the type of the created attribute, call the Bind method on the attribute.

```
pCreatedEntity->SetValue("Type", "ref:Integer");
```

## See also

"SetEntity" on page 2113,

"GetValue" on page 2089,

"GetEntity" on page 2093

# SetEntity

Sets the value of a metafeature for the entity on which the method is called. The value is represented as an entity.

```
HRESULT SetEntity(
          [in] BSTR strMetaFeature,
          [in] ITtdEntity* pEntity,
          [in, optional] VARIANT nPos);
```

### Parameters

`[in] strMetaFeature`

A string specifying the metafeature which should have its value set. The string should specify an existing metafeature for the entity on which the method is called, spelled with the correct case. If an incorrect metafeature is specified the method will fail.

`[in] pEntity`

The value to set, represented as an ITtdEntity pointer. If pEntity is NULL, the value of the specified metafeature will be reset.

`[in, optional] nPos`

The index of the metafeature which should have its value set. The entity will be inserted before the entity that is currently at that index. Indexes start at 1 and the index 0 can be used to specify the last index of the metafeature. If this optional parameter is omitted it will default to 0. If the index is outside a valid range, the entity will be inserted at the last position in the metafeature.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | Success. |
| E_FAIL | The value could not be set for one reason or another. Use the information provided in the COM error object for more details. |

### Comments

The SetEntity method can be used on all writable (non-derived) metafeatures of an entity that have Metaclass type. If the specified metafeature is a reference, SetEntity will set it by pointer to refer to the given entity. In order to set a reference by identifier or by GUID, use SetValue instead.

**Example 652**

The following example shows how to set up a reference by pointer, using the SetEntity method. Compare it with Example 651 on page 2112 where the same reference is set up by identifier instead:

```
// First find the Integer datatype using FindByGuid
ITtdEntityPtr pType;
pType = pITtdModel->FindByGuid("@Predefined@Integer");
pCreatedEntity->SetEntity("Type", pType);
```

### See also

"SetValue" on page 2111

"GetValue" on page 2089

"GetEntity" on page 2093

## SetTaggedValue

Sets a property (tagged value) of an element, or in general any value in an instance representation.

```
HRESULT SetTaggedValue(
          [in] BSTR strSelector,
          [in] BSTR strValue,
          [in, optional] VARIANT overwrite /* = true */);
```

### Parameters

`[in] strSelector`

A string containing a selector pattern that specifies which (tagged) value to set. The format of this pattern is described in the documentation of the GetTaggedValue method. If the pattern is ill-formed, or does not match, the method will fail.

`[in] strValue`

The value to set, encoded as a string. This string should be a valid expression. It will be parsed by the expression parser and the resulting expression will be inserted in the "tree of values" at the position determined by the selector pattern. If this value string cannot be parsed as an expression, the method will fail.

`[in, optional] overwrite`

If this parameter is true, the new value will overwrite any existing values selected by the selector pattern. This parameter is by default true since it typically is not desired to have more than one property (tagged value) for a particular attribute.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | The property (tagged value) could not be set for one reason or another. Use the information provided in the COM error object for more details. |

### Comments

`SetTaggedValue` can be called on all elements which can have stereotypes applied, or it can be called on an instance expression (metaclass InstanceExpr). If it is called on an element, the property (tagged value) will be set in the first applied stereotype instance, available on the element, for which the selector pattern matches. If it is called on an instance expression, the property (tagged value) will be set in that instance expression (provided that the selector pattern matches it of course).

In order to be able to overwrite one or many existing values (using the `overwrite` parameter) the instance expression that the selector pattern is matched against must be bound to the Signature of which it is an instance. Use the Bind method to make sure it is properly bound.

### Note

*When* `SetTaggedValue` *overwrites existing values (using the* `overwrite` *parameter), the previous values will be deleted. Any pointers on these deleted values will thus be invalid afterwards and should not be used. Compare with the* `Delete` *method for how to handle deleted entities in the client code.*

**Example 653**

This example uses `SetTaggedValue` to set a property (tagged value) on a Class created in the context of a Package pointed to by `pPackage`. The stereotype used is the SignedNote stereotype from Example 649 on page 2105. Before a property (tagged value) can be set, the stereotype must be applied (here using the CreateInstance method followed by a SetEntity call).

```
ITtdEntityPtr pClass;
pClass = pPackage->Create("Class");

ITtdEntityPtr pStereotypeInstance;
// Assuming that pStereotype points to the SignedNote stereotype...
pStereotypeInstance->CreateInstance(pStereotype);
pClass->SetEntity("StereotypeInstance", pStereotypeInstance);

pClass->SetTaggedValue("SignedNote (. author .)", "\"Elvis\"");
```

You must quote the property (tagged value) to set to make it a legal Charstring expression. If the quotes were omitted it would have been interpreted as an identifier.

**See also**

GetTaggedValue

## Create

Creates a new entity in the context of the entity on which the method is called. This means that a new (direct or indirect) child will be added to the entity.

```
HRESULT Create(
          [in] BSTR strMetaClass,
          [in, optional] VARIANT buildModelForPresentations,
          [in, optional] VARIANT strMetaFeature,
          [out, retval] ITtdEntity** ppCreatedEntity);
```

### Parameters

`[in] strMetaClass`

The name of a Metaclass in the Metamodel. The string should specify an existing metaclass, spelled with the correct case. If an incorrect metaclass is specified the method will fail.

`[in, optional] buildModelForPresentations`

Specifies whether created presentation elements should have a model representation built automatically. This optional parameter is by default true. It could be useful to set this parameter to false if the created presentation element should be set up to reference an existing model element rather than a new one.

`[in, optional] strMetaFeature`

A string specifying the metafeature in which the new entity should be created. This parameter is optional, and need only be given when it is ambiguous where to insert the created entity. In most cases an entity of the specified metaclass can only fit into one particular metafeature, and then this parameter is ignored. If the parameter is given, it should

specify an existing metafeature for the entity on which the method is called, spelled with the correct case. If an incorrect metafeature is specified the method will fail.

`[out, retval]  ppCreatedEntity`

The created entity.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | The creation failed. Use the information provided in the COM error object for more details. |

### Comments

`Create` is the recommended method for creating an entity as a direct or indirect child of the entity on which the method is called. It is similar to `ITtdModel::`New, but has a few benefits which usually makes it the better choice of the two:

- It will create the new entity and link it to the model as one atomic operation, thereby reducing the risk of orphan entities.
- It provides several creation "shortcuts", that is it often allows an entity to be created although it cannot be inserted as a direct child of the entity on which the method is called. `Create` will then automatically create the intermediate entities required in order to insert the new entity. Compare with how the "New" menu works in the Model View of the tool.
- Model representations for created presentation elements will be built automatically (unless the `buildModelForPresentations` parameter is set to false).

**Example 654**

This example shows how to use `Create` in order to create a package in the top-level model entity (the Session). Compare with Example 636 on page 2076 which does the same, using the less sophisticated `ITtdModel::`New method.

```
ITtdModelPtr pITtdModel;
pITtdModel = pITtdModelAccess->LoadProject("MyModel.ttp");
ITtdEntityPtr pSession = pITtdModel;
ITtdEntityPtr pPackage;
pPackage = pSession->Create("Package");
```

Now you can continue with creating a sequence diagram in the package. This is possible although the model does not allow a sequence diagram to be inserted as a direct child of a package. `Create` will automatically create a use case in the Package, an Interaction in the use case, and Interaction Implementation in the Interaction, and finally the sequence diagram in that Interaction Implementation.

```
ITtdEntityPtr pSQDiagram;
pSQDiagram = pPackage->Create("SequenceDiagram");
```

In the calls to `Create` above no metafeature has to be specified, since it is unambiguous where to create the entity. But in the example below, where the left and right operand shall be created in a binary expression pointed to by `pBinaryExpr`, a metafeature must be specified since there are two possible metafeatures of a BinaryExpr where an Expression could fit (namely LeftOperand and RightOperand):

```
// Create an assignment expression: myVar = 4
ITtdEntityPtr pIdentifier;
ITtdEntityPtr pValue;
pIdentifier = pBinaryExpr->Create("Ident", "LeftOperand");
pValue = pBinaryExpr->Create("IntegerValue", "RightOperand");
pIdentifier->SetValue("Name", "myVar");
pValue->SetValue("Value", "4");
```

**See also**

"New" on page 2074

## CreateInstance

Creates an instance of the entity on which the method is called. That entity should thus be a Signature that is possible to instantiate.

```
HRESULT CreateInstance(
          [out, retval] ITtdEntity** ppInstance);
```

### Parameters

`[out, retval]` `ppInstance`

The created instance.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | The creation failed. Use the information provided in the COM error object for more details. |

### Comments

`CreateInstance` implements the UML semantics of creating an instance of a Signature. The instance is represented in the model with the InstanceExpr Metaclass.

**Note**

*It is the responsibility of the client to take care of the returned instance. It should either be inserted into the model, or deleted, to avoid a memory leak.*

**Example 655**

Although `CreateInstance` can be called on any Signature, a particular case is when it is called on a Stereotype in order to apply it to an element. This example shows how to apply the predefined stereotype “usecase” on an operation pointed to by pOperation:

```
ITtdEntityPtr pStereotype;
pStereotype = pITtdModel->FindByGuid("@Predefined@usecase");
ITtdEntityPtr pInstance;
pInstance = pStereotype->CreateInstance();
pOperation->SetEntity("StereotypeInstance", pInstance);
```

## Delete

Deletes the entity on which the method is called.

```
HRESULT Delete();
```

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | Success. |
| E_FAIL | Failure due to an internal error. |

### Comments

Use `Delete` to remove an entity from the model, and delete the memory it occupies. If the entity owns other entities (directly or indirectly) these entities will also be deleted according to the semantics of the composition relationship.

**Note**

*Be careful so that pointers on deleted entities are not used after the deletion, as they then will be invalid.*

**Example 656**

This example uses `Delete` to delete the Package created in Example 654 on page 2119. `Detach` is called on the smart pointer after the deletion, in order to detach it from its interface pointer, which will be invalid after the deletion. `pPackage` can then be assigned to a new entity.

```
pPackage->Delete();
pPackage->Detach(); // Important since the pointer is invalid!
```

`Detach()` is part of the Microsoft API for handling C++ smart pointer classes (`_com_ptr_t::Detach`).

# XMLEncode

Encodes the entity on which the method is called into an XML representation.

```
HRESULT XMLEncode(
          [out, retval] BSTR* strXMLEncoding);
```

### Parameters

`[out, retval]  strXMLEncoding`

A string containing the XML encoding of the entity on which the method is called.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | The encoding to XML failed. Use the information provided in the COM error object for more details. |

### Comments

`XMLEncode` is typically used together with `ITtdModel::`XMLDecode. Together these methods provide a means for serializing model fragments to and from a string representation. Example 638 on page 2079 shows an example of how to use `XMLEncode`.

## See also

ITtdModel::XMLDecode

## MetaVisit

Performs a traversal of the model fragment that is rooted at the entity on which the method is called. For each entity that is encountered during this traversal, including the entity on which the method is called, methods are called through a callback interface. These callbacks thus allow the caller to do something for all or some of the visited entities.

```
HRESULT MetaVisit(
          [in] ITtdMetaVisitCallback* pMetaVisitCallback,
          [in, optional] VARIANT visitAll /* false */);
```

### Parameters

`[in] pMetaVisitCallback`

A pointer to the interface through which the callbacks will be performed during the model traversal. If this parameter is NULL the method will fail.

`[in, optional] visitAll`

A flag controlling whether library packages and the predefined package shall be traversed. If this optional parameter is omitted, it will default to false, meaning that such packages will not be traversed. The reason why library packages and the predefined package by default is not traversed is that it improves performance (in particular if the method is called from a script client).

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | Success. |
| E_FAIL | Failure due to an internal error. |
| E_POINTER | Invalid pointer argument. The pMetaVisitCallback pointer must not be NULL. |

### Comments

MetaVisit is a convenient method for traversing a model, or a subset of a model, that does not require any knowledge of the UML Metamodel. All direct and indirect composition children of an entity will be traversed, no matter in which metafeatures they reside.

By implementing the callback interface ITtdMetaVisitCallback, a client can do whatever it wants when an entity is visited. The MetaVisit method can thus be used for a great variety of purposes.

**Example 657**

This example shows how to use MetaVisit in order to find all definitions in a loaded model.

```
ITtdModelPtr pITtdModel;
pITtdModel = pITtdModelAccess->LoadProject("MyModel.ttp");
ITtdEntityPtr pSession = pITtdModel;
pSession->MetaVisit(pCallbackHandler);
```

pCallbackHandler is here assumed to point to a class that implements the ITtdMetaVisitCallback interface. For example, assume that the purpose of the traversal is to print the names of all definitions in the model. pCallbackHandler could then point to an instance of the class below:

```
class CDefinitionPrinter : public ITtdMetaVisitCallback {
public:

  virtual HRESULT __stdcall raw_OnVisitedEntity (ITtdEntity*
pVisitedEntity){
    if (pVisitedEntity->IsKindOf(_T("Definition"))){
      CComBSTR bstrValue((TCHAR*) pVisitedEntity-
>GetValue("Name"));
    }
    return S_OK;
  }

  virtual HRESULT __stdcall raw_OnAfterVisitedEntity (ITtdEntity*
pVisitedEntity){
    return S_OK;
}
```

```
  virtual HRESULT STDMETHODCALLTYPE QueryInterface(REFIID iid, void
** ppvObject){
    *ppvObject = NULL;
    if (iid == __uuidof(ITtdMetaVisitCallback)) {
      *ppvObject = static_cast<ITtdMetaVisitCallback*>(this);
      AddRef();
      return S_OK;
    }
  return E_FAIL;
    }

  virtual ULONG STDMETHODCALLTYPE AddRef(){return 0;}; // No ref
counting

  virtual ULONG STDMETHODCALLTYPE Release(){return 0;}; // No ref
counting
};
```

The ITtdMetaVisitCallback interface has one method called `OnVisitedEntity`, which is called for each visited entity. The visited entity is passed in as a parameter to that method. The class above also has to implement the `IUnknown` interface, since ITtdMetaVisitCallback inherits `IUnknown` (like all other COM interfaces).

There is also a second callback method called `OnAfterVisitedEntity`. It is also called once for each visited entity, but contrary to `OnVisitedEntity` it is called when the composition children of the entity already has been visited. It thus allows the caller to do something on the "back recursion" of the model traversal.

---

**See also**

"New" on page 2074

"OnVisitedEntity" on page 2157

"OnAfterVisitedEntity" on page 2158

## MetaVisitEx

This is an extended version of the MetaVisit method which allows the model traversal to include also reference representations.

```
HRESULT MetaVisitEx(
          [in] ITtdMetaVisitCallback* pMetaVisitCallback,
          [in, optional] VARIANT visitAll /* false */);
          [in, optional] VARIANT visitRefs /* false */);
```

### Parameters

MetaVisitEx adds one optional parameter to the MetaVisit signature:

```
[in,              visitRefs
optional]
```

A flag controlling whether references should be traversed or not. If this optional parameter is omitted, it will default to false, meaning that the model for references will not be traversed.

### Return value

See MetaVisit.

## Bind

Attempts to bind all references in the model fragment that is rooted at the entity on which the method is called. The method can also be used for binding only one particular reference on that entity.

```
HRESULT Bind(
          [in, optional] VARIANT strMetaFeature);
```

### Parameters

```
[in,              strMetaFeature
optional]
```

If this optional parameter is given, it should be a string specifying the metafeature which should be bound. The string should specify an existing metafeature for the entity on which the method is called, spelled with the correct case. If an incorrect metafeature is specified the method will fail.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | An error occurred. Use the information provided in the COM error object for more details. |

### Comments

Use `Bind` to attempt to bind all references in a model. There is an implicit call to `Bind` from LoadProject and LoadFile, to guarantee that a newly loaded model can be navigated on all its links and references.

A common case when `Bind` should be called, is when a reference has been set up by name or GUID using the `SetValue` method. In that case the metafeature could be passed as parameter to the `Bind` method in order to only bind that particular metafeature.

**Example 658**

The following example shows how to load a file `MyModel.u2` using the LoadFile method.

```
ITtdModelPtr pITtdModel;
pITtdModel = pITtdModelAccess->LoadFile("MyModel.u2");
```

Since LoadFile calls `Bind` implicitly when the model has been loaded, all links and references that could be bound will be bound, and could thus be navigated using the GetEntity or GetEntities methods.

Create a package in pITtdSession and create an attribute in that package. The type of the attribute is set up by name to the predefined datatype Integer, and `Bind` is then called on the attribute's Type metafeature in order to be able to navigate to the type using the GetEntity method.

```
ITtdEntityPtr pPackage, pAttribute;
pPackage = pITtdSession->Create(_T("Package"));
pAttribute = pPackage->Create(_T("Attribute"));
pAttribute->SetValue("Type", "ref:Integer");
pAttribute->Bind("Type");

ITtdEntityPtr pIntegerDatatype;
```

```
pIntegerDatatype = pAttribute->GetEntity("Type");
```

## Clone

Creates a clone of the entity. By default the clone will be unbound and have new unique GUIDs (i.e. the copy of the entity itself and the copy of all contained entities will get new unique GUIDs).

```
HRESULT Clone(
          [in, optional] VARIANT preserveBindings,
          [in, optional] VARIANT preserveGuids,
          [out, retval] ITtdEntity** result);
```

### Parameters

`[in, optional]` `preserveBindings`

If this optional parameter is given, it should be a boolean specifying whether bindings of the original entity should be preserved in the clone. By default bindings will not be preserved.

`[in, optional]` `preserveGuids`

If this optional parameter is given, it should be a boolean specifying whether GUIDs of the original entity (and its children) should be preserved in the clone. By default GUIDs will not be preserved.

`[out, retval]` `result`

The resulting clone.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | An error occurred. Use the information provided in the COM error object for more details. |

### Comments

Use `Clone` to create an identical copy of an entity. A “deep” cloning is performed, i.e. the clone will contain copies of all children of the original entity recursively.

**Important!**

*Be careful when cloning an entity without changing GUIDs. Such a clone should not be inserted into the same model as the original entity, or GUID conflicts will arise. If a model with GUID conflicts is saved, it might not be possible to load again.*

In order to be able to preserve bindings of the clone, the original entity must belong to a model (i.e. GetModel called on the original entity must not return NULL).

**Note**

*It is the responsibility of the client to take care of the returned entity. It should either be inserted into the model, or deleted, to avoid a memory leak.*

## Move

Moves the entity from its current location in the model into the context of a new owner.

```
HRESULT Move(
          [in] ITtdEntity* newOwner,
          [in, optional] VARIANT metafeature,
          [in, optional] VARIANT index);
```

### Parameters

```
[in]              newOwner
```

The new owner into which the entity shall be moved.

`[in, optional] metafeature`

The metafeature of the new owner into which the entity shall be moved. This parameter is optional, and need only be given when it is ambiguous where to move the entity. In most cases the moved entity can only fit into one particular metafeature, and then this parameter is ignored. If the parameter is given, it should specify an existing metafeature for the `newOwner` entity on which the method is called, spelled with the correct case. If an incorrect metafeature is specified the method will fail.

`[in, optional] index`

This optional parameter can be used to specify the position at which to insert the moved entity in a target metafeature that has non-single multiplicity.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | An error occurred. Use the information provided in the COM error object for more details. |

### Comments

Use `Move` to move an entity from one owner to another, without loosing its identify (its GUID). If the entity is orphan (does not have an owner) it is better to use `SetEntity` for inserting it into a model.

## See also

"SetEntity" on page 2113

## GetModel

Returns the model to which the entity belongs.

```
HRESULT GetModel(
          [out, retval] ITtdModel** result);
```

### Parameters

`[out, retval]` `result`

The model to which the entity belongs, or NULL if the entity does not belong to a model.

### Return value

The method always succeeds (i.e. returns S_OK).

### Comments

Use `GetModel` to obtain an `ITtdModel` interface on the top-level entity in the model to which an entity belongs.This method is often the most convenient way to get an ITtdModel interface from the context of an ITtdEntity interface.

## UnlinkFromOwner

Unlinks the entity from its current owner in the model.

```
HRESULT UnlinkFromOwner();
```

### Return value

The method always succeeds (i.e. returns S_OK).

### Comments

Use `UnlinkFromOwner` to unlink an entity from its current owner in the model. After the call the entity will be orphan, i.e. a call to GetOwner or GetModel will both return NULL.

### Note

*The entity will not be deleted, so it is the responsibility of the client to take care of it after the call. It should either be inserted into the model again, or deleted, to avoid a memory leak.*

### See also

"Delete" on page 2121

"Move" on page 2129

## Replace

Replaces an entity with another entity. The original entity will not be deleted.

```
HRESULT Replace(
          [in] ITtdEntity* replacementEntity);
```

### Parameters

`[in]` `replacementEntity`

The entity which shall replace the entity on which the method is called. If the replacement entity cannot be inserted where the original entity is, the method will fail.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The entity was successfully replaced. |
| `E_FAIL` | An error occurred. Use the information provided in the COM error object for more details. |

### Comments

Use `Replace` to replace an entity with another entity. The replacement entity will be inserted at exactly the same position in the model as the entity on which the method is called.

**Note**

*The original entity will not be deleted, so it is the responsibility of the client to take care of it after the call. It should either be inserted into the model again, or deleted, to avoid a memory leak.*

**Note**

*If the entity on which the method is called is an identifier representing a reference it will be replaced with a clone of* `replacementEntity`*, rather than* `replacementEntity` *itself. In that case it is the responsibility of the client to take care of the replacement entity itself after the call.*

## GetContainerMetaFeature

Obtains the name of the metafeature in which the entity is contained.

```
HRESULT GetContainerMetaFeature(
          [out, retval] BSTR* result);
```

### Parameters

`[out, retval]` `result`

The name of the container metafeature. This string will be empty if the entity is orphan.

### Return value

The method always succeeds (i.e. returns S_OK).

### Comments

Use `GetContainerMetaFeature` to find out the name of the metafeature in which an entity is contained. This metafeature name can then be used in calls to for example SetEntity, GetEntity or GetEntities.

## FindByName

Performs a name-lookup from the context of the entity on which the method is called, in order to find an entity with a certain name (possibly qualified).

```
HRESULT FindByName(
          [in] BSTR strName,
          [out, retval] ITtdEntity* result);
```

### Parameters

`[in]` `strName`

The name of the entity to find. The name may be qualified. It should be a valid UML identifier.

`[out, retval]` `result`

The found entity, or NULL is no matching entity was found.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | An error occurred. Use the information provided in the COM error object for more details. |

### Comments

Use `FindByName` to find a definition by the name through which it is referrable from the context of the entity on which the method is called. This is an alternative to using FindByGuid for finding an entity in the model, whose name rather than GUID is known.

### See also

"FindByGuid" on page 2073

## GetDescriptiveName

Obtains a textual description of an entity.

```
HRESULT GetDescriptiveName(
          [out, retval] BSTR* result);
```

### Parameters

`[out, retval]` `result`

The textual description of the entity.

### Return value

The method always succeeds (i.e. returns S_OK).

### Comments

Use `GetDescriptiveName` to get a string describing the entity on which the method is called. The description includes the Metaclass of the entity, its name (full signature for event classes) and its location in the model.

# ITtdEntities

The ITtdEntities interface represents a collection of entities. The interface specifies a collection that can be both read and written, although a client in most cases only has to read from the collection (that is traverse and access the entities of the collection). There is no specific Metaclass in the UML Metamodel corresponding to the ITtdEntities interface.

The methods of the ITtdEntities interface is the usual methods available on a read/write COM collection interface:

| | |
|---|---|
| `_NewEnum` | Returns an enumeration interface pointer for the collection, to allow iteration over the contained entities. |
| `Item` | Returns the entity at the specified index in the collection. |

| | |
|---|---|
| Count | Returns the number of entities in the collection. |
| Add | Adds an entity to the collection. |
| Remove | Removes an entity from the collection. |

# _NewEnum

Returns an enumeration interface pointer for the entity collection on which the method is called. The enumeration interface is the standard Automation enumeration interface, and it can be used to step through the entities in the collection.

_NewEnum is defined as a read-only property.

```
HRESULT _NewEnum(
          [out, retval] IUnknown** ppUnk);
```

### Parameters

[out, retval] ppUnk

A pointer to an enumeration interface for the collection.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | Success. |
| E_FAIL | An error occurred. Use the information provided in the COM error object for more details. |

### Comments

Using the _NewEnum method is one way to iterate over the entities in an ITtdEntities collection. The other way is to loop over the indexes of the collection and to call the Item method to get the entity at a certain index. _NewEnum is typically the more efficient alternative.

**Example 659**

The following example shows how to iterate over the attributes in a class (pointed to by pClass) using the `_NewEnum` method:

```
ITtdEntitiesPtr pAttributes = pClass->GetEntities(_T("Attribute"));
// Iterate through the collection using the enumerator...
// Get the VARIANT enumerator from the collection
IEnumVARIANTPtr spEnum = pAttributes->Get_NewEnum();
// nBatchSize is the number of items requested in each call to
IEnumVARIANT::Next.
// The actual number of items returned may not equal nBatchSize.
const ULONG nBatchSize = 5;
// nReturned will store the number of items returned by a call to
// IEnumVARIANT::Next
ULONG nReturned = 0;
// arrVariant is the array used to hold the returned items
VARIANT arrVariant[nBatchSize] = {0};

HRESULT hr = E_UNEXPECTED;
do {
  hr = spEnum->Next(nBatchSize, &arrVariant[0], &nReturned);
  if (FAILED(hr))
    return hr;

  ITtdEntityPtr pAttribute;
  for (ULONG i = 0; i < nReturned; ++i){
    _variant_t vt(arrVariant[i]);
    pAttribute = vt;
    if (pAttribute){
      // Do something with pAttribute
    }
    ::VariantClear(&arrVariant[i]);
  }
} while (hr != S_FALSE); // S_FALSE indicates end of collection
```

`_NewEnum` is defined as a read-only property of the ITtdEntities interface, and is thus called using the prefix “Get”.

---

## See also

Item

# Item

Returns the entity at the specified index in the entity collection on which the method is called.

`Item` is defined as a read-only property.

```
HRESULT Item(
          [in] long Index,
          [out, retval] ITtdEntity** ppEntity);
```

### Parameters

`[in] Index`

The index of the collection where to extract an entity. Index should be a valid index between 1 (the first entity in the collection) and the result from calling `Count` (the number of entities in the collection).

`[out, retval] ppEntity`

The entity at the specified index in the collection, or NULL if no such entity exists.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | An error occurred. Use the information provided in the COM error object for more details. |

### Comments

Use the `Item` method to access an arbitrary entity in an ITtdEntities collection. `Item` can also be used to iterate over the entities in the collection, but this is typically less efficient than using an enumeration interface obtained from the `_NewEnum` method.

**Example 660**

The following example shows how to iterate over the attributes in a class (pointed to by pClass) using the `Item` method. Using the `_NewEnum` method instead is described in Example 659 on page 2137.

```
ITtdEntitiesPtr pAttributes = pClass->GetEntities(_T("Attribute"));
// Iterate through the collection using the Item method...

long lCount = 0;
lCount = pAttributes->Count;

ITtdEntityPtr pAttribute;
```

```
// N.B. Index are 1-based
for (long i = 1; i <= lCount; i++){
  pAttribute = pAttributes->GetItem(i);
  if (pParameter != 0){
    // Do something with pAttribute
  }
}
```

`Item` is defined as a read-only property of the ITtdEntities interface, and is thus called using the prefix “Get”.

---

**See also**

“_NewEnum” on page 2136

“Count” on page 2139

## Count

Returns the number of entities in the entity collection on which the method is called.

`Count` is defined as a read-only property.

```
HRESULT Count(
          [out, retval] long* pVal);
```

### Parameters

`[out, retval]  pVal`

The number of entities in the collection.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | Success. |
| E_FAIL | An error occurred. Use the information provided in the COM error object for more details. |

### Comments

Use the Count method to determine the number of entities in an ITtdEntities collection. All items in the collection are counted, including NULL pointers. Example 660 on page 2138 has an example of how to use Count.

## Add

Adds an entity to the entity collection on which the method is called.

```
HRESULT Add(
          [in] ITtdEntity* pEntity,
          [in, optional] VARIANT nIndex);
```

### Parameters

[in] pEntity

The entity that is to be added to the collection.

[in, optional] nIndex

An index specifying where in the collection the entity shall be inserted. Indexes start at 1 and the index 0 can be used to specify that the entity shall be added to the end of the collection. If this optional parameter is omitted it will default to 0. If an index is specified it must be within a valid range, otherwise the method will fail.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | Success. |
| E_FAIL | An error occurred. Use the information provided in the COM error object for more details. |

### Comments

Use the Add method to add an entity to an ITtdEntities collection. An ITtdEntities collection may include NULL pointers, and may also contain the same entity more than once.

**Example 661**

The following example uses GetEntities to obtain a collection of entities. The first entity of the collection is then added once more to the end of the collection. Finally the Remove method is used to remove all occurrences of that entity in the collection.

```
ITtdEntitiesPtr pAttributes = pClass->GetEntities(_T("Attribute"));
ITtdEntityPtr pEntity = pAttributes->GetItem(1);
pAttributes->Add(pEntity); // Add once more last in collection
pAttributes->Remove(pEntity); // Removes all occurrences of pEntity
```

## Remove

Removes all occurrences of an entity from the entity collection on which the method is called.

```
HRESULT Remove(
          [in] ITtdEntity* pEntity);
```

### Parameters

[in] pEntity

The entity that is to be removed from the collection. This parameter must not be NULL.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | Success. |
| E_POINTER | A NULL pointer is given as the pEntity parameter. |

### Comments

Use the Remove method to remove all occurrences of an entity from an ITtdEntities collection. Example 661 on page 2141 contains an example on usage of the Remove method.

# ITtdResource

The ITtdResource interface is implemented by the Resource class of the Metamodel, which represents a resource where a UML model could be persistently stored. Typically a Resource corresponds to a .u2 file.

**Note**

*Since a Resource also is an Entity, a QueryInterface from ITtdResource to ITtdEntity will always succeed.*

ITtdResource contains the following methods:

| | |
|---|---|
| Save | Saves the model entities associated with the resource (typically saves the corresponding .u2 file). |

## Save

Saves the model entities that are associated with the resource on which the method is called. For the common case when the resource represents a .u2 file, this means that the file will be saved.

```
HRESULT Save();
```

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | The entities associated with the resource were successfully saved in the resource. |
| E_FAIL | An error occurred while saving. This happens for example when the file of the resource is read-only, or when there is not enough space on the disk. Use the information provided in the COM error object for more details. |

### Comments

Use the Save method to save the entities associated with a resource (that is its roots) in the resource. If the entire model shall be saved, it is more convenient to call ITtdModel::Save.

**Example 662**

This example uses Save to save the Package created in Example 636 on page 2076 in a file of its own:

```
ITtdEntityPtr pResource;
pResource = pITtdModel->CreateResource("D:\\temp\\COMtest.u2");

// Insert the created package pPackage as a root of pResource
pResource->SetEntity("Root", pPackage);

pITtdResource->Save(); // Saves all resources in the model
```

### See also

"Save" on page 2079

# ITtdPresentationElement

The ITtdPresentationElement interface is implemented by the PresentationElement class of the Metamodel. A presentation element is an element with a graphical appearance, for example a diagram, symbol or a line.

**Note**

*Since a PresentationElement also is an Entity, a QueryInterface from ITtd-PresentationElement to ITtdEntity will always succeed.*

ITtdPresentationElement contains the following methods:

| | |
|---|---|
| GenerateEMF | Generates an EMF file (Enhanced Meta File) for a presentation element. (deprecated) |
| GenerateEMFEx | Generates an EMF file (Enhanced Meta File) for a presentation element with support for scaling the image. |
| GenerateImage | Generates an image file for a presentation element. |

# GenerateEMF

Generates an EMF file (Enhanced Meta File) for the graphical appearance of a presentation element. This is a deprecated function. Use GenerateEMFEx instead. The presentation element will have the same appearance in this EMF file as when shown in the tool's editors.

```
HRESULT GenerateEMF(
          [in] BSTR strFileName,
          [in, optional] VARIANT maxWidth,
          [in, optional] VARIANT maxHeight,
          [in, optional] VARIANT optimizeForVectorGraphics,
          [in, optional] VARIANT includeFrame);
```

## Parameters

`[in] strFileName`

The file name of the EMF file to generate. If `strFileName` is a relative path, it will be interpreted as relative to the current working directory of the client application.

`[in, optional] maxWidth`

`[in, optional] maxHeight`

These optional parameters can be used to specify the maximum size of the generated image. The image will be scaled to fit into the specified size. If the parameters are omitted, the generated image will be the same size as shown in the tool's editors. The unit of the height and width numbers is 1/10:th of a millimeter. These parameters are currently only considered if the presentation element on which the method is called is a diagram.

`[in, optional]` `optimizeForVectorGraphics`

If this optional parameter is set to true, the EMF generation will be optimized for vector graphics. The default behavior is to not do this optimization. This parameter is currently only considered if the presentation element on which the method is called is a diagram. This parameter is considered deprecated and exists for backward compatibility purposes.

`[in, optional]` `includeFrame`

This optional parameter specifies whether the frame symbol of a diagram should be included in the EMF generation or not. By default it will be included. This parameter is only considered if the presentation element on which the method is called is a diagram.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The EMF file was successfully generated. |
| `E_FAIL` | An error occurred. This happens for example when the EMF file cannot be saved at the specified location, or if a required Tau license could not be obtained. Use the information provided in the COM error object for more details. |

### Comments

The `GenerateEMF` method can for example be used by clients that produce reports from a UML model, as a means for visualizing presentation elements. It works on all kinds of presentation elements, and if the presentation element contains other presentation element (as is the case with for example a diagram), the EMF file will include these contained presentation elements as well.

## GenerateEMFEx

Generates an EMF file (Enhanced Meta File) for the graphical appearance of a presentation element. The presentation element will have the same appearance in this EMF file as when shown in the tool's editors. This is the recommended method to call in order to generate an EMF file for a presentation element.

```
HRESULT GenerateEMFEx(
          [in] BSTR strFileName,
          [in, optional] VARIANT maxWidth,
          [in, optional] VARIANT maxHeight,
          [in, optional] VARIANT optimizeForVectorGraphics,
          [in, optional] VARIANT includeFrame);
          [in, optional] VARIANT scaleFactor);
```

### Parameters

`[in] strFileName`

The file name of the EMF file(s) to generate. If `strFileName` is a relative path, it will be interpreted as relative to the current working directory of the client application.

`[in, optional] maxWidth`

`[in, optional] maxHeight`

These optional parameters can be used to specify the maximum size of the generated image. The image will be scaled to fit into the specified size if no specific scaleFactor is given. If the parameters are omitted, the generated image will be the same size as shown in the editors. The unit

of the height and width numbers is 1/10:th of a millimeter. These parameters are currently only considered if the presentation element on which the method is called is a diagram.

`[in, optional]` `optimizeForVectorGraphics`

If this optional parameter is set to true, the EMF generation will be optimized for vector graphics. The default behavior is to not do this optimization. This parameter is currently only considered if the presentation element on which the method is called is a diagram. This parameter is considered deprecated and exists for backward compatibility purposes.

`[in, optional]` `includeFrame`

This optional parameter specifies whether the frame symbol of a diagram should be included in the EMF generation or not. By default it will be included. This parameter is currently only considered if the presentation element on which the method is called is a diagram.

`[in, optional]` `scaleFactor`

If this optional parameter is given the original diagram is scaled before generating any images. The scale factor should be given as an integer and is interpreted as percent of the original diagram size. If both the `scaleFactor` and `maxWidth`/`maxHeight` are given as arguments then more than one image may be generated as a result of the operation. If the `scaleFactor` parameter is given then the name of the generated files is the same as the `strFileName` parameter but with a number added before the file extension.

**Return value**

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | The EMF file was successfully generated. |
| E_FAIL | An error occurred. This happens for example when the EMF file cannot be saved at the specified location, or if a required Tau license could not be obtained. Use the information provided in the COM error object for more details. |

### Comments

The GenerateEMFEx method can for example be used by clients that produce reports from a UML model, as a means for visualizing presentation elements. It works on all kinds of presentation elements, and if the presentation element contains other presentation element (as is the case with for example a diagram), the EMF file will include these contained presentation elements as well.

**Example 663**

This example shows the GenerateEMFEx method can be used to create an EMF file for each diagram in a package pointed to by pPackage:

```
ITtdEntitiesPtr pDiagrams;
pDiagrams = pPackage->GetEntities(_T("Diagram"));

long lCount = pDiagrams->Count;

ITtdPresentationElementPtr pDiagram;
// N.B. Index is 1-based
for (long i = 1; i <= lCount; i++){
  pDiagram = pDiagrams->GetItem(i);
  if (pDiagram != 0){
    // A diagram is also an entity...
    ITtdEntityPtr pE = pDiagram; // ... so this is OK!
    pDiagram->GenerateEMFEx(pE->GetValue(_T("Name")));
  }
}
```

## GenerateImage

Generates an image file of a specified kind for the graphical appearance of a presentation element. The presentation element will have the same appearance in this image file as when shown in the tool's editors.

```
HRESULT GenerateImage(
          [in] ImageKind imgKind,
          [in] BSTR strFileName);
```

### Parameters

`[in] imgKind`

The kind of image file to generate. Valid values for this parameter are shown in the table below:

| ImageKind | Description |
|---|---|
| IK_JPEG | Generate a JPEG image file. |
| IK_BMP | Generate a BMP image file. |
| IK_GIF | Generate a GIF image file. |
| IK_TIFF | Generate a TIFF image file. |
| IK_TARGA | Generate a TGA (Targa) image file. |
| IK_DIB | Generate a device independent bitmap file. |
| IK_PCX | Generate a PCX image file. |

`[in] strFileName`

The file name of the image file to generate. If `strFileName` is a relative path, it will be interpreted as relative to the current working directory of the client application.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | The image file was successfully generated. |
| E_FAIL | An error occurred. This happens for example when the image file cannot be saved at the specified location, or if a required Tau license could not be obtained. Use the information provided in the COM error object for more details. |

### Comments

The GenerateImage method can for example be used by clients that produce reports from a UML model, as a means for visualizing presentation elements. It works on all kinds of presentation elements, and if the presentation element contains other presentation element (as is the case with for example a diagram), the image file will include these contained presentation elements as well.

# ITtdSymbol

The ITtdSymbol interface is implemented by the Symbol class of the Metamodel. A symbol is a presentation element whose graphical appearance occupies a two-dimensional region in a diagram. It has thus both a size and a position.

**Note**

*Since a Symbol also is a PresentationElement and an Entity, a QueryInterface from ITtdSymbol to ITtdPresentationElement or ITtdEntity will always succeed.*

ITtdSymbol contains the following methods:

| | |
|---|---|
| SetSize | Sets the size of the symbol. |
| SetPosition | Sets the position of the symbol. |

## SetSize

Sets the size of a symbol. This is the recommended method for changing the size of a symbol.

```
HRESULT SetSize(
          [in] long width,
          [in] long height);
```

### Parameters

`[in] width`

The width of the symbol. The unit of this number is 1/10:th of a millimeter. It should be a positive number.

`[in] height`

The height of the symbol. The unit of this number is 1/10:th of a millimeter. It should be a positive number.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The symbol was successfully resized. |
| `E_FAIL` | An error occurred. Use the information provided in the COM error object for more details. |

### Comments

For many symbols the `SetSize` method simply acts as a wrapper for setting a new size using ITtdEntity::SetValue on the “size” metafeature. However, for some symbols the size has a semantic significance. Resizing such a symbol can thus lead to some modifications in the object model. It could also be the case that the resizing of one symbol should trigger another symbol to

be resized. These “side-effects” of a symbol resize will only happen if `SetSize` is used. That is the reason why it is the recommended method for changing the size of a symbol.

In order to read the size of a symbol use ITtdEntity::GetValue on the “size” metafeature.

## SetPosition

Sets the position of a symbol. This is the recommended method for changing the position of a symbol.

```
HRESULT SetPosition(
          [in] long x,
          [in] long y);
```

### Parameters

`[in]  x`

The horizontal position of the symbol. The unit of this number is 1/10:th of a millimeter. It should be a positive number.

`[in]  y`

The vertical position of the symbol. The unit of this number is 1/10:th of a millimeter. It should be a positive number.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The symbol was successfully repositioned. |
| `E_FAIL` | An error occurred. Use the information provided in the COM error object for more details. |

**Comments**

For many symbols the `SetPosition` method simply acts as a wrapper for setting a new position using ITtdEntity::SetValue on the "position" metafeature. However, for some symbols the position has a semantic significance. Repositioning such a symbol can thus lead to some modifications in the object model. It could also be the case that when one symbol is moved, some other symbols should be moved to. These "side-effects" of a symbol repositioning will only happen if `SetPosition` is used. That is the reason why it is the recommended method for changing the position of a symbol.

The position of a symbol is defined as the coordinate of the upper left corner of its bounding rectangle.

In order to read the position of a symbol use ITtdEntity::GetValue on the "position" metafeature.

# ITtdExpression

The ITtdExpression interface is implemented by the Expression class of the Metamodel. Expressions may appear at various places in a model.

**Note**

*Since an Expression also is an Entity, a QueryInterface from ITtdExpression to ITtdEntity will always succeed.*

ITtdExpression contains the following methods:

| | |
|---|---|
| `GetType` | Computes the type of the expression. |
| `EvaluateConstantIntegralExpression` | Evaluates the integral value of a constant expression. |
| `GetInstanceChildExpression` | Finds a child expression of an instance. |

## GetType

Computes and returns the type of the expression. If the type cannot be computed (for example because the expression contains unbound references) NULL is returned.

```
HRESULT GetType(
          [out, retval] ITtdEntity** result);
```

### Parameters

```
[out, retval]   result
```

The type of the expression.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | Success. |
| E_FAIL | Failure due to an internal error. |

### Comments

Use GetType to find out the type of an expression. All expressions have a type, but it is not guaranteed that the type is a definition located in the model (i.e. GetModel may return NULL when called on the returned entity). This happens when the type is implicitly defined, and in those cases the returned entity will be a temporary entity representing the type. Make sure not to store any pointers or delete such a temporary object.

## EvaluateConstantIntegralExpression

Evaluates the value of an expression, which is expected to be a constant integral expression.

```
HRESULT EvaluateConstantIntegralExpression(
          [out, retval] long* value);
```

### Parameters

```
[out, retval]   value
```

The evaluation of the expression as an integer.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The expression was successfully evaluated. |
| `E_FAIL` | An error occurred. This happens for example when the expression cannot be evaluated because it contains unbound identifiers. Use the information provided in the COM error object for more details. |

### Comments

Use `EvaluateConstantIntegralExpression` to find out the constant integer number to which to expression evaluates. This method is useful for clients that operate on expressions, and want to be independent on how the expression is structured (only its value is interesting).

## GetInstanceChildExpression

Obtains the right-hand side of an assignment contained in an instance, where the left-hand side of the assignment is a certain identifier.

```
HRESULT GetInstanceChildExpression(
          [in] BSTR strName,
          [out, retval] ITtdExpression** result);
```

### Parameters

`[in]  strName`

The name of the identifier which the left-hand side of the assignment must match. This name must be a valid identifier, and may include a qualifier.

`[out, retval]  result`

The matching child expression.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | An error occurred. Use the information provided in the COM error object for more details. |

### Comments

Use `GetInstanceChildExpression` on an instance expression (for example a stereotype instance) to obtain the right-hand side of a contained assignment, where the left-hand side of the assignment is an identifier matching `strName` (which thus should be a valid identifier).

If no matching child expression is found, NULL is returned.

Although `GetInstanceChildExpression` can be used to extract tagged values from a stereotype instance, be aware that it doesn't cover all cases supported by GetTaggedValue.

# ITtdMetaVisitCallback

The ITtdMetaVisitCallback interface is a callback interface that clients using the ITtdEntity::MetaVisit method must implement.

ITtdMetaVisitCallback contains the following methods:

| | |
|---|---|
| `OnVisitedEntity` | Called for each entity that is visited during the model traversal. |
| `OnAfterVisitedEntity` | As above, but the call is made after all composition children of the entity already have been visited. |

## OnVisitedEntity

Called by the ITtdEntity::MetaVisit implementation when an entity is encountered (visited) during the traversal of a model.

```
HRESULT OnVisitedEntity(
          [in] ITtdEntity* pVisitedEntity);
```

### Parameters

`[in] pVisitedEntity`

The entity currently being visited during the model traversal.

### Return value

The value returned by `OnVisitedEntity` is used by the `MetaVisit` implementation to determine how to proceed with the traversal. The returned HRESULT should be one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Entity successfully visited. Proceed with the traversal of its composition children. |
| `S_FALSE` | Entity successfully visited, but do not traverse its composition children. This return value can be used to skip the traversal of parts of the model that are of no interest for the client. |
| `E_FAIL` | Abort the model traversal. This return value can for example be used when `MetaVisit` is used for finding a particular entity. If `pVisitedEntity` turns out to be the entity that is looked for, `OnVisitedEntity` could return `E_FAIL` to effectively abort the traversal. |
| `any other return value` | Will be treated as `E_FAIL`, but will also cause the `MetaVisit` call to fail (that is return `E_FAIL`). This can be used when an error occurs while visiting `pVisitedEntity` that should be propagated to the calling client code. |

### Comments

The `OnVisitedEntity` callback method is called for all visited entities, including the entity on which the MetaVisit method is called. The client should do whatever it likes to do with the visited entity, and then return a return value according to the table above to specify how to proceed with the model traversal.

In Example 657 on page 2124 there is an example of how to use `OnVisitedEntity` for implementing a Metamodel driven traversal of a model.

## OnAfterVisitedEntity

Called by the ITtdEntity::MetaVisit implementation when an entity is encountered (visited) during the traversal of a model. The call is made when all composition children of the entity already have been visited.

```
HRESULT OnAfterVisitedEntity(
          [in] ITtdEntity* pVisitedEntity);
```

### Parameters

`[in]  pVisitedEntity`

The entity currently being visited during the model traversal.

### Return value

The value returned by `OnAfterVisitedEntity` is used by the `MetaVisit` implementation to determine how to proceed with the traversal. The returned HRESULT should be one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Entity successfully visited. Proceed with the traversal. |
| `E_FAIL` | Abort the model traversal. This return value can for example be used when `MetaVisit` is used for finding a particular entity. If `pVisitedEntity` turns out to be the entity that is looked for, `OnAfterVisitedEntity` could return `E_FAIL` to effectively abort the traversal. |
| `any other return value` | Will be treated as `E_FAIL`, but will also cause the `MetaVisit` call to fail (that is return `E_FAIL`). This can be used when an error occurs while visiting `pVisitedEntity` that should be propagated to the calling client code. |

**Comments**

The `OnAfterVisitedEntity` callback method is called for all visited entities, including the entity on which the `MetaVisit` method is called. The call takes place when all (direct and indirect) composition children of the entity already has been visited, and is thus a means to do something on the "back recursion" of the model traversal. The client should do whatever it likes to do with the visited entity, and then return a return value according to the table above to specify how to proceed with the model traversal.

In Example 657 on page 2124 there is an example of how to use `OnAfterVisitedEntity` for implementing a Metamodel driven traversal of a model.

# ITtdInteractiveClient

The ITtdInteractiveClient interface is an interface that all interactive COM clients of the COM API must implement (except for agents which instead must implement ITtdAgent). An interactive client is a client that is invoked by the Tau application, and that may access a model loaded in that application. ITtdInteractiveClient defines the interface through which Tau communicates with the client.

ITtdInteractiveClient contains the following methods:

| | |
|---|---|
| OnExecute | Called by Tau when the client shall start to execute. |

# OnExecute

This method is called on the client, when Tau is ready to let it execute.

```
HRESULT OnExecute(
          [in] ITtdInteractiveServer* pServer,
          [in] ITtdEntities* pEntities);
```

### Parameters

`[in] pServer`

A pointer to a server object implementing the ITtdInteractiveServer interface. The client may communicate with the server, the Tau application, through that interface.

`[in] pEntities`

A collection of entities that is provided to the client by the server. These entities constitute the context on which the client can start working.

### Return value

The HRESULT value returned by `OnExecute` should be one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The client completed its execution successfully. |
| `E_FAIL` | An error occurred in the client during its execution. |

### Comments

The `OnExecute` method is called synchronously by Tau to let the interactive client execute. The client should perform all its actions in the implementation of this method.

In order to make Tau call the method on the client, a Tcl command called std::ExecuteCOMClient is used:

This Tcl command can for example be part of the Tcl script specified for a Tau add-in module. If the `OnExecute` method fails, the Tcl command will also fail.

**Note**

*The ITtdInteractiveClient interface is only provided for backwards compatibility with clients that were developed before the introduction of the ITtdAgent interface. New API clients should use the ITtdAgent interface since it allows the client to be invoked from all APIs (C++, COM and Tcl), and it also has the advantage of being able to pass actual arguments to the client. See Agents for more information about agents.*

# ITtdInteractiveServer

The ITtdInteractiveServer interface is implemented by the Tau module that launches an interactive client. It defines the interface through which the client communicates with Tau during its execution.

**Note**

*It is possible to obtain the ITtdModelAccess interface from the ITtdInteractiveServer interface. This can be useful for interactive COM clients that wants to utilize the methods provided by that interface.*

ITtdInteractiveServer contains the following methods:

| | |
|---|---|
| CreateEntityCollection | Creates an empty collection of entities. |
| InterpretTclScript | Interprets a Tcl script on the server. |

## CreateEntityCollection

Creates an empty collection of entities, and returns it to the caller.

```
HRESULT CreateEntityCollection(
          [out, retval] ITtdEntities** pEntityCollection);
```

### Parameters

`[out, retval] pEntityCollection`

The created empty collection of entities.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Entity collection successfully created. |
| `E_OUTOFMEMORY` | Failed to create a new collection due to insufficient memory. |

### Comments

`CreateEntityCollection` is a helper method that can be used by the client in order to get a container for entities. The client can use this container internally, or it can use it to be able to call other methods of the ITtdInteractiveServer interface that requires an ITtdEntities collection as argument.

The created collection should not be deleted by the client. It will delete itself when no one holds a reference to it anymore (i.e. when the last user calls `Release` on it).

Example 664 on page 2164 contains an example of how to use `CreateEntityCollection`.

## InterpretTclScript

Interprets a Tcl script on the server, and returns the result to the caller. This method is a means for an interactive client to communicate with the server.

```
HRESULT InterpretTclScript(
          [in] BSTR strScript,
          [in, optional] VARIANT pEntities,
          [out, retval] BSTR* strResult);
```

### Parameters

`[in] strScript`

A string containing the Tcl script to interpret. The string may contain "substitution markers" that will be substituted with the Tcl ids of the corresponding entities in the `pEntities` collection.

`[in, optional] pEntities`

An ITtdEntities collection of entities that are arguments to the Tcl script in `strScript`. This parameter is optional and need only be given when the Tcl script contains "substitution markers".

`[out, retval] strResult`

The result of interpretation of the Tcl script.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The script is successfully interpreted. |
| `E_FAIL` | An error occurred during argument substitution or script interpretation. Use the information provided in the COM error object for more details. |

### Comments

Currently `InterpretTclScript` is the only means for an interactive client to communicate with Tau during its execution. Although it could feel a bit awkward to use a Tcl script, rather than calling COM methods, it gives the client access to the entire Tcl API supported by Tau.

The Tcl script may include “substitution markers” on the form `#n`, where `n` is an index of an entity in the entity collection. The server will preprocess the script to replace all markers with the Tcl id of the corresponding entity in the provided entity collection. As usual, indexes start at 1.

**Example 664**

This example demonstrates how an interactive client may communicate with the server using the `InterpretTclScript` method. The code below is part of the client’s `OnExecute` method, and `pServer` is the ITtdInteractiveServer interface pointer which the client obtains as argument in that method.

```
// First create an empty collection of entities that are arguments
// to the Tcl script.
ITtdEntitiesPtr pEntities = pServer->CreateEntityCollection();

// Add an entity to the collection
pEntities->Add(pEntity);

// Specify the Tcl script using a substitution marker for the first
// entity of the collection
CComBSTR strScript(_T("output #1"));

// Interpret the Tcl script and save the result in strResult
CComBSTR strResult((BSTR) pServer->InterpretTclScript((BSTR)
strScript, pEntities));
```

# ITtdSourceBuffer

The ITtdSourceBuffer interface represents a buffer of text (typically source code) used primarily during code generation as a representation of a generated file. It can for example be used by agents of the C++ Application Generator that wish to add some additional text to a generated file.

ITtdSourceBuffer contains the following methods:

| | |
|---|---|
| AddText | Add some text to the source buffer. |

## AddText

Adds a piece of text to the source buffer.

```
HRESULT AddText(
          [in] BSTR text);
```

### Parameters

`[in] text`

The text string to add to the source buffer.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The text is successfully added to the source buffer. |
| `E_FAIL` | An error occurred. Use the information provided in the COM error object for more details. |

### Comments

The `AddText` method will write a piece of text to the source buffer. Source buffers are written sequentially, and the text will be added at the current position in the source buffer.

A typical use of this method occurs when a code generator agent wants to write additional text to a generated file. The file is represented as a source buffer which is filled by the code generator. The agent may listen for certain tool events that are sent by the code generator when certain entities are printed to the buffer. Additional text can thus be added to the source buffer just before or just after the printing of such an entity.

# ITtdMessageList

The ITtdMessageList interface represents a list of messages. Such a list is for example used when reporting errors from semantic analysis or code generation, and the interface may be used by agents in order to add custom messages based on for example a custom semantic check.

ITtdMessageList contains the following methods:

| | |
|---|---|
| AddMessage | Add a new message to the message list. |

## AddMessage

Adds a new message to the end of the message list.

```
HRESULT AddMessage(
          [in] BSTR text,
          [in] MessageSeverity severity,
          [in, optional] VARIANT subject);
```

### Parameters

`[in] text`

The text of the message to add.

`[in] severity`

The severity of the message. The following literals are possible:

- MS_INFORMATION
  Use this for information messages.
- MS_WARNING
  Use this for warning messages.
- MS_ERROR
  Use this for error messages.
- MS_FATAL
  Use this for fatal error messages (errors that are unrecoverable).

`[in, optional] subject`

This parameter is an optional subject entity attached with the message. Subject entities are located to when a message is located (double-clicked) in the Tau IDE.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | The message is successfully added to the message list. |
| E_FAIL | An error occurred. Use the information provided in the COM error object for more details. |

**Comments**

The AddMessage method will add a new message to the end of the message list.

A typical use of this method occurs when a code generator or semantic checker agent wants to report a message in the Check or Build tab. The agent will then receive the ITtdMessageList interface from the code generator or semantic checker (which triggers the tool event that the agent has registered for).

# ITtdAgent

The ITtdAgent interface is used when implementing Agents using the COM API. A COM object that is the implementation of an agent must implement the ITtdAgent interface.

ITtdAgent contains the following methods:

| | |
|---|---|
| Execute | Called by Tau when the agent is invoked. |

## Execute

This method is called on the agent, when Tau is ready to let it execute.

```
HRESULT Execute(
          [in] ITtdEntity* triggeredBy,
          [in] VARIANT beforeProcessing,
          [in] ITtdEntity* modelContext,
          [in] IUnknown* server,
          [in, out] SAFEARRAY(VARIANT)* agentParameters);
```

### Parameters

`[in] triggeredBy`

A pointer to the UML definition of an operation that caused the agent to be triggered. The operation is either a tool event or an agent definition. The agent can need this information if it may be invoked by more than one tool event or agent. If the agent is invoked programmatically (using InvokeAgent) this parameter will be NULL.

`[in] beforeProcessing`

This parameter will be true if the agent is invoked through an ordering dependency stereotyped by the 'before processing' stereotype, otherwise it will be false. An agent can need this information if it can be triggered through more than one ordering dependency. This parameter is not applicable if the agent is invoked programmatically (using InvokeAgent).

`[in] modelContext`

The entity that is the model context for the agent invocation.

`[in] server`

A pointer to a server object through which the agent may communicate with the Tau application that invoked it. An interactive agent receives a server object that implements the ITtdInteractiveServer interface. For a non-interactive agent the server object may realize another interface, which provides services that are relevant for that particular application. See for example ITtdCppAppGenServer which provides services for agents running in the C++ Application Generator executable.

For all kinds of agents the ITtdModelAccess interface can also be obtained from this parameter.

`[in, out] agentParameters`

This parameter is a list (safe array) of Agent Parameters that are carried by the Tool Events that triggered the invocation of the agent (directly or indirectly). Which parameters that are passed depend on the tool event, and sometimes even on the ordering dependency that is involved in the invocation.

If the agent is invoked programmatically (using InvokeAgent) this list will contain the actual arguments that was passed to the agent by the caller.

The agent may modify the parameter list in order to communicate information back to the caller.

### Return value

The HRESULT value returned by `Execute` should be one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The agent completed its execution successfully. |
| `E_FAIL` | An error occurred in the agent during its execution. In the case of an interactive agent an error message will be printed in the Message tab. For non-interactive agents the error message is typically printed on the `stderr`. |

### Comments

The `Execute` method is called synchronously by Tau to let the agent execute. The agent should perform all its actions in the implementation of this method.

Example 625 on page 2033 contains an example on how to realize the `ITtdAgent` interface and implement the `Execute` method.

# ITtdCppAppGenServer

The `ITtdCppAppGen` interface is implemented by the C++ Application Generator. It is used when customizing C++ code generation using Agents implemented as COM objects. The agent can obtain this interface from the server parameter in the call to its Execute method.

ITtdCppAppGenServer contains the following methods:

| | |
|---|---|
| ScheduleForDeletion | Use this method to delete an entity in the model. |

## ScheduleForDeletion

If an agent wants to delete an entity, this is the method to use. The C++ Application Generator will immediately unlink the entity to be deleted from the model, so it will not be seen by the code generator afterwards. However, the entity is not actually deleted until the code generator finds it appropriate to do so.

```
HRESULT ScheduleForDeletion(
          [in] ITtdEntity* entity);
```

### Parameters

```
[in]   entity
```

The entity to be deleted.

### Return value

The HRESULT value returned by `ScheduleForDeletion` is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The entity was successfully scheduled for deletion. |
| `E_FAIL` | An error occurred. Use the information provided in the COM error object for more details. |

### Comments

It is important that the agent uses ScheduleForDeletion, and not the Delete method. The latter method will immediately delete the entity, which might give the C++ Application Generator, or other active customization agents problems. For example, they may have stored pointers to the deleted entity that would become invalid.

# ITtdStudioAccess

The ITtdStudioAccess interface is the default interface of the TTD_StudioAccess COM class, and is obtained when creating an instance of that class. It is the main entry point for accessing functionality in the Tau IDE that are not specific to UML modelling.

ITtdStudioAccess contains the following methods:

| | |
|---|---|
| OpenWorkspace | Open a workspace (.ttw file) |
| NewWorkspace | Create a new workspace |
| OpenProject | Open a project (.ttp file) |
| GetWorkspace | Get a reference to the currently loaded workspace. |
| InterpretTclScript | Interpret a Tcl script. |
| GetApplicationName | Get the full name of the running Tau application. |
| GetApplicationPID | Get the PID (process ID) of the running Tau application. |
| GetApplicationVersion | Get the version of the running Tau application. |
| GetApplicationUserName | Get the name of the user of the running Tau application. |

## OpenWorkspace

Opens a Tau workspace file (extension `ttw`). All projects contained in the workspace will be loaded.

```
HRESULT OpenWorkspace(
          [in] BSTR strPath,
          [out, retval] ITtdWorkspace** workspace);
```

### Parameters

`[in]  strPath`

A string specifying the workspace file to load. If the string specifies a relative path, it will be interpreted as relative to the current working directory (typically the installation `bin` directory of Tau). The string may contain URN references.

`[out, retval]  workspace`

A pointer to the loaded workspace.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The workspace was successfully opened. |
| `E_FAIL` | Failed to open the specified workspace. Use the information provided in the COM error object for more details. |

### Comments

The `OpenWorkspace` method opens an existing workspace by reading a Tau workspace file (.ttw). This method is equivalent of opening the workspace using the **File - Open Workspace** menu.

## NewWorkspace

Create a new Tau workspace and associate it with a workspace file (extension `ttw`).

```
HRESULT NewWorkspace(
          [in] BSTR strPath,
          [out, retval] ITtdWorkspace** workspace);
```

### Parameters

`[in]  strPath`

A string specifying where to save the workspace file of the workspace. If the string specifies a relative path, it will be interpreted as relative to the current working directory (typically the installation `bin` directory of Tau).

```
[out, retval]  workspace
```

A pointer to the new workspace.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The workspace was successfully created. |
| `E_FAIL` | Failed to create the specified workspace. Use the information provided in the COM error object for more details. |

### Comments

The `NewWorkspace` method creates a new workspace and saves it in the specified workspace file (.ttw). If an existing workspace was open it will first be closed. This method is equivalent of creating a new blank workspace using the **File - New** menu.

## OpenProject

Opens a Tau project file (extension `ttp`). The model associated with the project will be loaded.

```
HRESULT OpenProject(
          [in] BSTR strPath,
          [out, retval] ITtdProject** project);
```

### Parameters

```
[in]  strPath
```

A string specifying the project file to load. If the string specifies a relative path, it will be interpreted as relative to the current working directory (typically the installation `bin` directory of Tau). The string may contain URN references.

`[out, retval]  project`

A pointer to the loaded project.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The project was successfully opened. |
| `E_FAIL` | Failed to open the specified project. Use the information provided in the COM error object for more details. |

### Comments

The `OpenProject` method opens an existing project by reading a Tau project file (.ttp). This method is equivalent of opening the project using the **File - Open** menu.

The opened project will be inserted into the current workspace. If there is no workspace available, the method will fail. Use GetWorkspace to check if a workspace is available.

**Note**

*The opened project will also be set as active as a side-effect of calling this method.*

## GetWorkspace

Returns the current workspace.

```
HRESULT GetWorkspace(
          [out, retval] ITtdWorkspace** workspace);
```

### Parameters

`[out, retval] workspace`

A pointer to the current workspace, or NULL if no workspace is open.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The workspace was successfully obtained. |
| `E_FAIL` | Failed to access the workspace. Use the information provided in the COM error object for more details. |

### Comments

The `GetWorkspace` method returns the workspace that is currently open in Tau. It can also be used for checking if a workspace is currently available, since it returns NULL in case no workspace is available.

## InterpretTclScript

Interpret a Tcl script in Tau.

```
HRESULT InterpretTclScript(
          [in] BSTR strScript,
          [out, retval] BSTR* result);
```

### Parameters

`[in] strScript`

A string containing a Tcl script to interpret.

`[out, retval] result`

The result of interpretation of the Tcl script.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | The script was successfully interpreted. |
| E_FAIL | An error occurred while interpreting the script. Use the information provided in the COM error object for more details. |

### Comments

The ínterpretTclScript method allows any Tcl script to be interpreted by Tau. Which Tcl commands that are available for use in the script depends on what is loaded in Tau. As a general rule all Tcl commands prefixed with std are always available, while those that are prefixed with u2 only are available when a UML model is loaded.

## GetApplicationName

Returns the Tau application product name.

```
HRESULT GetApplicationName(
          [out, retval] BSTR* name);
```

### Parameters

```
[out, retval]  name
```

The application product name.

### Return value

This method always succeeds (i.e. returns S_OK).

### Comments

The `GetApplicationName` method returns a string which is the product name of the running Tau application. Different Tau products support different features, and this method is thus a means for a client to know which Tau features it can utilize.

## GetApplicationPID

Returns the PID (process id) of the Tau application.

```
HRESULT GetApplicationPID(
          [out, retval] BSTR* pid);
```

### Parameters

`[out, retval]  pid`

The PID (process id) of the Tau application.

### Return value

This method always succeeds (i.e. returns S_OK).

### Comments

The `GetApplicationPID` method returns the PID (process id) of the running Tau application. This method can for example be useful when there are multiple instances of Tau running on a machine, and a client wants to communicate with one particular of these instances. The PID returned by this method is then a means for distinguishing the different Tau instances.

## GetApplicationVersion

Returns the version of the Tau application.

```
HRESULT GetApplicationVersion(
          [out, retval] BSTR* version);
```

### Parameters

`[out, retval]  version`

The version of the Tau application.

### Return value

This method always succeeds (i.e. returns S_OK).

### Comments

The `GetApplicationVersion` method returns the version number (as a string) of the running Tau application. Different Tau versions support different features, and this method is thus a means for a client to know which Tau features it can utilize.

## GetApplicationUserName

Returns the name of the user that is running the Tau application.

```
HRESULT GetApplicationUserName(
          [out, retval] BSTR* name);
```

### Parameters

`[out, retval]  name`

The user name for the user who is running the Tau application.

### Return value

This method always succeeds (i.e. returns S_OK).

### Comments

The `GetApplicationUserName` method returns the user name of the user who is running the Tau application. This method can for example be useful when there are multiple instances of Tau running on a machine with more than one user logged onto it. Using this method a client can know if a certain Tau application instance runs under the same user as the client, or under some other user.

# ITtdWorkspace

The ITtdWorkspace interface represents a Tau workspace. It contains methods that operate on a workspace.

ITtdWorkspace contains the following methods:

| | |
|---|---|
| GetPath | Get the path of a workspace. |
| GetProject | Get a project contained in a workspace. |
| GetActiveProject | Get the active project in a workspace. |
| SetActiveProject | Set the active project in a workspace. |

## GetPath

Returns the full path of the workspace file where this workspace is stored.

```
HRESULT GetPath(
          [out, retval] BSTR* path);
```

### Parameters

`[out, retval]  path`

The workspace path.

### Return value

This method always succeeds (i.e. returns S_OK).

### Comments

The `GetPath` method returns the full path of the workspace, i.e. the path of the workspace file.

## GetProject

Returns a project with a specified path that is contained in this workspace.

```
HRESULT GetProject(
          [in] BSTR strPath,
          [out, retval] ITtdProject** project);
```

### Parameters

`[in] strPath`

The project file path to search for.

`[out, retval] project`

The found project with the specified path.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | A matching project was found. |
| `E_FAIL` | No matching project was found. Use the information provided in the COM error object for more details. |

### Comments

The `GetProject` searches the workspace for a contained project which is stored in a project file with a specified path. Note that project path comparisons are done by just comparing the project file paths of the workspace with strPath, without normalizing the paths or expanding URNs.

## GetActiveProject

Returns the currently active project of the workspace.

```
HRESULT GetActiveProject(
          [out, retval] ITtdProject** project);
```

### Parameters

`[out, retval] project`

The active project.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | Success. |
| `E_FAIL` | No active project was found. Use the information provided in the COM error object for more details. |

### Comments

A Tau workspace contains one or more projects of which one is always active. `GetActiveProject` returns a reference to the active project.

## SetActiveProject

Sets a project as active in this workspace.

```
HRESULT SetActiveProject(
          [in] ITtdProject* project);
```

### Parameters

`[in] project`

The project to set as active in the workspace.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| S_OK | The project was successfully set as active. |
| E_FAIL | Failed to set the project as active. Use the information provided in the COM error object for more details. |

### Comments

The SetActiveProject sets a project of a workspace as active. A workspace can at most have one active project, so the previously active project will thus not be active anymore after the call.

# ITtdProject

The ITtdProject interface represents a Tau project. It contains methods that operate on a project.

ITtdProject contains the following methods:

| | |
|---|---|
| GetPath | Get the path of a project. |
| GetName | Get the name of a project. |
| GetModel | Get the UML model of a project. |

## GetPath

Returns the full path of the project file where this project is stored.

```
HRESULT GetPath(
          [out, retval] BSTR* path);
```

### Parameters

[out, retval] path

The project path.

### Return value

This method always succeeds (i.e. returns S_OK).

### Comments

The `GetPath` method returns the full path of the project, i.e. the path of the project file.

## GetName

Returns the name of this project.

```
HRESULT GetName(
          [out, retval] BSTR* name);
```

### Parameters

`[out, retval]  name`

The project name.

### Return value

This method always succeeds (i.e. returns S_OK).

### Comments

The `GetName` method returns the name of the project. This is usually the name of the project file without path.

## GetModel

Returns the UML model of this project.

```
HRESULT GetModel(
          [out, retval] IDispatch** model);
```

### Parameters

`[out, retval] model`

The model of the project.

### Return value

The return value obtained from the returned HRESULT is one of the following:

| Return value | Meaning |
|---|---|
| `S_OK` | The model of the project was successfully obtained. |
| `E_FAIL` | No model could be obtained for the project. Use the information provided in the COM error object for more details. |

### Comments

`GetModel` returns a reference to the model of the project. This method acts as a bridge between the `TTD_StudioAccess` COM class and the `TTD_ModelAccess` COM class. The type libraries for these COM classes are separated, and the model parameter is therefore typed by IDispatch. However, by using QueryInterface on the returned pointer you may obtain an ITtdModel pointer from it.

# 77

# Tcl API

This chapter is the reference documentation for the Tau Tcl API. Available Tcl commands and how to use them are described.

Intended readers are developers of client applications that use the Tcl API for accessing the UML model or for adding dialogs and menus to Tau. Client applications could be everything from small interactive Add-Ins to full fledged code generators or import applications. A basic knowledge of Tcl is assumed throughout this chapter.

# Introduction

The Tcl API is divided into the following groups of commands:

- The General Purpose Commands provide functionary like accessing the current selection, loading documents and producing output to report panes.
- The User Interface Add-in Specific Commands make it possible to add menus, dialogs, toolbars and to define the actions that will be performed upon invocation. The user interface commands are only available from Tcl scripts executed via Add-Ins.
- The Model Commands provide a number of operations applicable on the currently loaded UML model.
- The Entity Commands provide read and write access to the details of the currently loaded UML model. These commands thus makes it possible to write scripts that changes a model in an interactive fashion.
- The Resource Commands provide the necessary commands to save a model in a physical storage unit, most commonly a file in the file system.
- The Presentation Element Commands provide functionality to create a representation of the presentation elements like diagrams in a picture format similar to what is shown on the screen.
- The Library Handling Commands provide functionality for loading and unloading UML libraries.
- The Semantic Checker Commands provide means for constructing and adding user-defined semantic checks to the semantic checker.
- The Utility Interface Commands provide miscellaneous functionality corresponding to functions in interfaces of the public Tau APIs, which are not covered by any other group of commands.

### Mapping of COM to Tcl commands

The COM API and the Tcl API are very similar for the following sections:

- Model Commands
- Entity Commands
- Resource Commands
- Presentation Element Commands
- Symbol Commands

- Expression Commands

In general the Tcl commands in these sections have the same names as the COM methods, with an additional prefix denoting that they are part of the 'u2' Tcl namespace. The command arguments are also very similar in Tcl, except from a mandatory extra argument and a different handling of optional arguments. To begin with, the extra argument is always stated first when calling a model command. This argument corresponds to the object that the COM method operates on. The first Tcl argument for all commands in this section is thus a reference to a model. When it comes to the optional COM parameters, these are in Tcl stated as options according to the following syntax:

```
-parameter value
```

Here, *parameter* is the name of the COM parameter and *value* is the parameter value. The kind of value to use, for example string, integer or boolean, is sometimes indicated by the first character in the parameter name. It could also sometimes be apparent from the name of the value.

It could also be worth pointing out that a shortcut exists for boolean options which value is to be set to true. In this case, stating the option is enough and the accompanying value can be left out. The following two examples of setting a boolean option to true are thus identical:

```
-bOverwrite true
-bOverwrite
```

Compare with the respective COM command documentation for more details on arguments and options.

**Hint**

*To find out the synopsis of a Tcl command corresponding to a COM method, invoke the command* `u2::<COM method name>`*. This will print a message describing how the command should be used.*

Error handling is also different in Tcl compared to COM. In the COM case, all methods return an error value, while the Tcl commands throw an exception if an error is encountered. This implies that return values described in the COM API documentation does not apply to the corresponding Tcl commands. Instead, the return value for a Tcl command is shown as a parameter of kind '`retval`' in the COM API.

**Example 665:**

Mapping a COM method to a Tcl command is usually straight forward. For example, consider the FindByGuid command that is defined as follows in the COM API documentation:

```
HRESULT FindByGuid(
          [in] BSTR strGuid,
          [out, retval] ITtdEntity** ppEntity);
```

The corresponding Tcl command looks as follows:

**u2::FindByGuid** *modelRef guid*

This command returns a reference to a model, taking two parameters, a model reference, *modelRef*, and a string containing a GUID, *guid*. It could be called as follows:

```
set myObj [u2::FindByGuid $model "@Predefined@integer"]
```

To make sure that any errors during the execution are handled properly, the call can also be included in a catch clause:

```
if [catch {
 set myObj [u2::FindByGuid $model "@Predefined@integer"]
} ret] {
 std::Output "Error during execution of Tcl
script\n$ret\n"
}
```

# General Purpose Commands

The general purpose part contains the following commands:

| Command | Description |
|---|---|
| std::BrowserReport | Adds a new object called <childobject> in the tree. |
| std::BrowserReportInit | Activates a tree tab inside the Browser view. If needed the tab is created. |
| std::Button | |

| | |
|---|---|
| std::ComboBox | |
| std::Dialog | Creates a dialog box and immediately builds and displays it. |
| std::DirectoryDialog | Display directory selection dialog. |
| std::ExecuteCOMClient | Invoke interactive COM client. |
| std::FileOpenDialog | Display a file selection dialog. |
| std::FileSaveDialog | Display a file save dialog. |
| std::Frame show-window | Hide, Show Tau frame window |
| std::GetActiveProject | Get active project in current workspace. |
| std::GetInstallationDirectory | Get path to Tau installation directory. |
| std::GetKind | Get a string identifying the general kind of a Tcl object. |
| std::GetModels | Get list of loaded models. |
| std::GetProject | Get list of projects. |
| std::GetProjectPath | Returns the absolute path to the given project. |
| std::GetSelection | Get list of currently selected entities. |
| std::GetUserAddinsDirectory | Get path to user Add-Ins directory. |
| std::GetUserDirectory | Get path to user information directory. |
| std::GetWebServerPort | Get the TCP/IP port currently used by the Tau Web Server. |
| std::HtmlReport | Open html file. |
| std::IsModified | Check if project is modified. |
| std::Label | |
| std::Locate | This function locates an object described by <locatestring>. This string must be understood by a DataServer. |
| std::MessageDialog | Display a modal message dialog. |
| std::OpenDocument | Open document. |

| | |
|---|---|
| std::Output | Print message in Tcl output tab. |
| std::OutputTab | Clears the content of an output tab or activates an output tab. |
| std::Report | This command is used for adding new lines in a report tab. |
| std::Quit | Equivalent to the “close window” button. |
| std::ReportInit | This command is used to create a new report tab in the output window. |
| std::SaveAll | Save workspace or project. |
| std::TextReport | Open a file and locate a position. |
| std::View | The std::View command allows you to control the windows canvas area. |

# std::BrowserReport

## Synopsis

```
std::BrowserReport [-expanded] [-userdata <userdata>] <childobject>
[<parentobject>]
```

## Description

Adds a new object called <childobject> in the tree.

If -expanded is present and if the object has children, then the node will be expanded, otherwise it will be collapsed.

If <parentobject> is present the node is created as a child of the node corresponding to <parentobject>.

If -userdata is present, then <userdata> will be associated to this node on the tree.

**Example 666:**

```
std::BrowserReportInit MyBrowser
std::BrowserReport -expanded [std::GetActiveProject]
[std::GetSelection]
```

# std::BrowserReportInit

## Synopsis

```
std::BrowserReportInit <tabname> [<iconfile>] [-keep] [-cb
<filename>]
```

## Description

Activates a tree tab inside the Browser view. If needed the tab is created.

If <iconfile> is present, then the icon will be used as the tab image.

If -keep is present then the content of the tab (if any) is kept, otherwise the tab is reset.

If the -cb <filename> is present, then the associated TCL script file will be evaluated when a double click on a tree object occurs and the OnDoubleClick proc of this TCL script will be called with double-clicked object as parameter.

**Example 667:**

```
std::BrowserReportInit MyBrowser
std::BrowserReport -expanded [std::GetActiveProject]
[std::GetSelection]
```

# std::Button

## Synopsis

```
std::Button [-name <buttoncaption>] [-command <commandproc>]
```

**Example 668:**

```
std::Button -variable B1 -name "Cancel" -parent $parentWnd
-x 20 -y 90 -w 60 -h 25 -command OnCancel
$B1 -name "New Name" -x 40 -y 90 -w 100 -h 25
```

# std::ComboBox

## Synopsis

```
std::ComboBox [-stringlist <list>][-edit <editstate>][-sort
<sortstate>][-selected <item>][-selchangecmd <selchgproc>][-
editchangecmd <editchgproc>]
```

## Description

- `-stringlist <list>`: list of items of the combo box control.
- `-edit <editstate>`: sets the editing capacity of the combo control. The parameter editstate is a Boolean. The value "true" indicates it is possible for the user to edit the text of the combo, and "false" that it is not possible. The default value is "false".
- `-sort <sortstate>`: the parameter sortstate is a boolean. The value "true" indicates that the list displayed in the combo is sorted.
- `-selected <item>`: sets the selected item.
- `-selchangecmd <selchgproc>`: sets the name of the command to call when the selection changes in the combo control. The command selchg-proc has one parameter which is the new selected string.
- `-editchangecmd <editchgproc>`: sets the name of the command to call when the user changes the text in the edit box of the combo control. The command editchgproc has one parameter which is the text of the edit box.

**Example 669:**

```
std ::ComboBox -variable C1 -name "" -parent $parentWnd -x
130 -y 55 -w 70 -h 150
$C1 -stringlist {one two three}
```

# std::Dialog

## Synopsis

```
std::Dialog -variable <dialogvar> -name <dialogname> -w
<dialogwidth> -h <dialogheight> -onbuilddialog <dialogbuildproc> -
oninitdialog <dialoginitproc> -onclosedialog <dialogcloseproc> -
closecmddialog <dialogclosecmd>
```

## Description

Creates a dialog box and immediately builds and displays it.

The description of the dialog box is done in the `dialogbuildproc` command. The dialog box is modal.

**Example 670:**

```
std::Dialog  -variable "MyDialogBox" -name "Projects" -w
450 -h 300 -onbuilddialog OnBuildDialog -oninitdialog
OnInitDialog -onclosedialog OnCloseDialog -
closecmddialog CloseDialog
```

- `-variable <dialogvar>`: sets the variable containing the dialog box.
- `-name <dialogname>`: sets the title of the dialog box.
- `-w <dialogidth>`: sets the width of the dialog box. The dialog box is always created on the center of the screen..
- `-h <dialogheight>`: sets the height of the dialog box.
- `-onbuilddialog <dialogbuildproc>`: specifies the TCL command called when the dialog box is built. You can create here the controls of the window. The dialog box has to be built each time it is displayed.

**Example 671:**

```
proc OnBuildDialog {adrDialog} {
  std::Label -name "Quality Model:" -parent $parentWnd -x
10 -y 25 -w 70 -h 25
}
```

- `-oninitdialog <dialoginitproc>`: specifies the TCL command called when the dialog box is about to be displayed. You can initialize here the data displayed in the dialog box.

**Example 672:**

```
proc OnInitDialog {adrDialog} {
        std ::Output  "Dialog initialization\n"
}
```

- `-onclosedialog <dialogcloseproc>`: specifies the TCL command called when the dialog box is about to be closed.

**Example 673:**

```
proc OnCloseDialog {parentWnd} {
    return 1
}
```

- `-closecmddialog <dialogclosecmd>`: defines a new command in the TCL environment which can be called in order to close the dialog box programmatically.

## std::DirectoryDialog

Display directory selection dialog.

### Synopsis

```
package require dialogs
std::DirectoryDialog ?-fromdir directory?
```

### Description

This command displays a directory selection dialog allowing the user to browse and select a directory. Optionally, *directory* can be used to set the initial directory to be selected in the dialog. The default value is the current directory.

The return value is the path of the selected directory, or 0 if the user cancels the operation.

**Example 674:**

This example displays a directory selection dialog, initially set to `C:\Temp`, and stores the resulting selection in a variable.

```
package require dialogs
set dirPath [std::DirectoryDialog -fromdir "C:\\Temp"]
```

## std::ExecuteCOMClient

Invoke interactive COM client.

### Synopsis

```
std::ExecuteCOMClient COMAddInProgId entityRef ?entityRef ...?
```

### Description

This command is used for calling an interactive COM client that implements the interface `ITtdInteractiveClient`. The command takes two arguments, *COMAddinProgId* and one or more *entityRefs*, representing the programmatic id of the COM client and the model entities passed to it respectively.

**Example 675**

A COM client, with id "`Tau.CodeGen`", operating on the active project in the current workspace, is called as follows:

```
set project [std::GetActiveProject]
set session [std::GetModels -kind U2 -project $project]
std::ExecuteCOMClient "Tau.CodeGen"  $session
```

### See also

COM API

## std::FileOpenDialog

Display a file selection dialog.

### Synopsis

```
std::FileOpenDialog ?-filter visibility?
```

### Description

This command displays a file selection dialog that allows the user to browse and select a file. Optionally, the *visibility* argument controls what files are visible in the dialog, using a string as follows:

- `text|pattern|`
  *Text* is the text to be displayed, while *pattern* controls the actual filtering.

The return value is the path of the selected file, or 0 if the user cancels the operation.

**Example 676:**

A file selection dialog

```
set res [std::FileOpenDialog -filter "Text Files
(*.txt)|*.txt|"]
```

## std::FileSaveDialog

Display a file save dialog.

### Synopsis

**std::FileSaveDialog** ?-**filename** *filename*? ?-**fileext** *extension*? ?-**filter** *filter*?

### Description

This command opens a file save dialog that allows the user to browse and specify a file. Three optional arguments are accepted. *Filename* sets the default name of the file, while *extension* sets the default extension of the file, that is to say that the extension to use unless specified by the user. *Filter* specifies which file filter to use in the dialog. The filter string should be on the following format:

- `text|pattern|`
  *Text* is the text to be displayed, while *pattern* controls the actual filtering.

The return value is the path of the specified file, or 0 if the user cancels the operation.

If the specified file already exists, the user is prompted to confirm the replacement.

**Example 677:**

This example illustrates how to display a file save dialog, with initial settings of the filename and file extension, and how to capture the users choice in a variable.

```
set res [std::FileSaveDialog -filename "myTextfile" -
```

```
fileext "txt"]
```

## std::Frame show-window

Hide, Show Tau frame window

### Synopsis

```
std::Frame show-window show-window-value
```

### Description

`std::Frame show-window` can take any of the following values:

- `hide` Hides this window and passes activation to another window.
- `minimize` Minimizes the window and activates the top-level window in the system's list.
- restore Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position.
- `show` Activates the window and displays it in its current size and position.
- `showmaximized` Activates the window and displays it as a maximized window.
- `showminimized` Activates the window and displays it as an icon.
- `showminnoactive` Displays the window as an icon. The window that is currently active remains active.
- `showna` Displays the window in its current state. The window that is currently active remains active.
- `shownoactivate` Displays the window in its most recent size and position. The window that is currently active remains active.
- `shownormal` Activates and displays the window. If the window is minimized or maximized, Windows restores it to its original size and position.

**Example 678:**

```
std::Frame show-window minimize
```

## std::GetActiveProject

Get active project in current workspace.

### Synopsis

```
std::GetActiveProject
```

### Description

A Tau workspace contains one or more projects of which one is always active. This command returns a reference to the active project.

**Example 679**

This example shows how to store a reference to the currently active project in a variable.

```
set curProj [std::GetActiveProject]
```

## std::GetInstallationDirectory

Get path to Tau installation directory.

### Synopsis

```
std::GetInstallationDirectory
```

### Description

This command returns a character string containing the path to the directory where Tau is installed.

**Example 680:**

The following code shows how to combine the Tau installation directory with the path to the `addins` subdirectory, using the Tcl built-in command 'file'.

```
set installDir [std::GetInstallationDirectory]
set addinsDir [file join $installDir addins]
```

# std::GetKind

Get a string identifying the general kind of a Tcl object.

### Synopsis

```
std::GetKind object
```

### Description

This command returns a string which identifies the overall classification of a Tcl object. It is mainly used in conditions to make sure that only objects of a certain kind are passed to a Tcl command which only supports input objects of that kind. Currently there are two different identifiers that may be returned by GetKind:

- **U2**
  The object is a UML model entity. This is the kind of most objects visible in the Model View.
- **project**
  The object is a “project model” entity. This is the kind of most objects visible in the File View.

**Example 681:**

The following code prints the kind of the node that is selected in the workspace window (Model View or File View):

```
output \n[std::GetKind [std::GetSelection] ]
```

# std::GetLocaleDirectory

### Synopsis

```
std::GetLocaleDirectory
```

### Description

The `std::GetLocaleDirectory` command returns the absolute path to the installation directory in which locale specific DLLs and files are stored.

**Example 682:**

If the current locale is Japanese (“jp”) then std::GetLocaleDirectory will return the concatenation of the absolute path to the Tau installation and “/locale/jp”.

## std::GetModels

Get list of loaded models.

### Synopsis

```
std::GetModels ?-kind modelKind? ?-project projRef?
```

### Description

This command returns a list of loaded models. If no arguments are specified, all models in the current workspace are returned, independent of their respective kind.

The optional argument *modelkind* can be used to restrict the output to only contain models of a certain kind. Currently supported values are:

- **U2**
  Only UML models are returned.

The optional parameter *projRef* is used to restrict the output to only contain models from the project in question.

**Example 683:**

Get all models from all projects in the workspace:

```
set allModels [std::GetModels]
```

**Example 684:**

Get all UML models from all projects in the workspace:

```
set umlModels [std::GetModels -kind U2]
```

**Example 685:**

Get the UML model from the active project in the workspace:

```
set curProject [std::GetActiveProject]
set model [std::GetModels -kind U2 -project $curProject]
```

## std::GetProject

Get list of projects.

### Synopsis

**std::GetProject** ?*entityRef*?

### Description

This command returns a list of projects. If no argument is given, a list containing references to all projects in the current workspace is returned.

The optional argument *entityRef* is used when a reference to the project that contains a specific entity is of interest.

**Example 686:**

This example shows how to call procedure "`ProcessIt`" on all projects in the current workspace.

```
set allProjs [std::GetProject]

foreach p $allProjs {
  ProcessIt $p
}
```

**Example 687:**

This shows how to get the project that contains an entity:

```
set thisProj [std::GetProject $anEntity]
```

# std::GetProjectPath

## Synopsis

```
std::GetProjectPath <projectRef>
```

## Description

Returns the absolute path to the given project.

**Example 688:**

```
# first select a project node in the FileView
set selection [std::GetSelection]
set pathname [std::GetProjectPath $selection]

if { $selection != "" } {
std::Output "This is the selected project's path:
$pathname\n";
} else {
        std::Output "A non-project node is selected.\n";
}
```

# std::GetSelection

Get list of currently selected entities.

## Description

This command returns a list containing references to all entities currently selected.

**Example 689:**

This example shows how to call the procedure "`ProcessIt`" on all selected elements.

```
set sel [std::GetSelection]

foreach s $sel {
  ProcessIt $s
}
```

## std::GetUserAddinsDirectory

Get path to user Add-Ins directory.

### Synopsis

```
std::GetUserAddinsDirectory
```

### Description

This command returns a character string containing the path to the directory used for storing user Add-Ins.

The default location on Windows:

```
c:\Documents and Settings\<username>\Application
Data\Telelogic\<Tau_version>\addins
```

For the default location on UNIX the $HOME variable is used and then "`Telelogic`" is appended. From there on the path will be the same.

The environment variable `TAU_USER_ADDINS_DIR` can be set to change the default location.

## std::GetTeamAddinsDirectory

Get path to team Add-Ins directory.

### Synopsis

```
std::GetTeamAddinsDirectory
```

### Description

This command returns a character string containing the path to the directory used for storing user Add-Ins.

The default location on Windows:

```
c:\Documents and Settings\<username>\Application
Data\Telelogic\Shared\TeamAddins
```

For the default location on UNIX the $HOME variable is used and then "`Telelogic`" is appended. From there on the path will be the same.

The environment variable `TAU_TEAM_ADDINS_DIR` can be set to change the default location.

## std::GetCompanyAddinsDirectory

Get path to company Add-Ins directory.

### Synopsis

```
std::GetCompanyAddinsDirectory
```

### Description

This command returns a character string containing the path to the directory used for storing user Add-Ins.

The default location on Windows:

```
c:\Documents and Settings\<username>\Application
Data\Telelogic\Shared\CompanyAddins
```

For the default location on UNIX the $HOME variable is used and then "`Telelogic`" is appended. From there on the path will be the same.

The environment variable `TAU_COMPANY_ADDINS_DIR` can be set to change the default location.

## std::GetUserDirectory

Get path to user information directory.

### Synopsis

```
std::GetUserDirectory
```

### Description

This command returns a character string containing the path to the directory used for storing user information. For example:

```
c:\Documents and Settings\<username>\Application
Data\Telelogic\<Tau Version>
```

## std::GetWebServerPort

Get the TCP/IP port currently used by the Tau Web Server.

### Synopsis

```
std::GetWebServerPort
```

### Description

This command returns the TCP/IP port number that is currently used by the Tau Web Server. It is typically used for constructing an URL which is later passed to the std::HtmlReport command. For example:

**Example 690:**

```
set url
"http://localhost:[std::GetWebServerPort]/file/index.htm
l"
std::HtmlReport $url
```

## std::HtmlReport

Open `html` file.

### Synopsis

**`std::HtmlReport`** *`url`*

### Description

This command loads and displays a `html` file in Tau. The *url* argument is a string containing the URL in question.

**Example 691:**

Displaying the Telelogic home page on the Tau desktop is done like this:

```
std::HtmlReport "www.telelogic.com"
```

**Example 692:**

Displaying a `html` file on the Tau desktop is done like this:

```
std::HtmlReport "C:\\test.htm"
```

## std::IsModified

Check if project is modified.

### Synopsis

**`std::IsModified`** *projRef*

### Description

This command checks if a project, specified with a project reference *projRef*, is modified or not. It returns 0 if not modified and 1 otherwise.

**Example 693:**

This example saves the workspace in case the active project is modified.

```
if { [std::IsModified $std::activeproject] } {
  std::SaveAll
}
```

## std::Label

### Synopsis

**`std::Label`** `[-name <staticname>]`

### Description

`-name <staticname>`: sets the caption of the static control.

**Example 694:**

```
std::Label -name "Quality Model:" -parent $parentWnd -x 10
```

```
-y 25 -w 70 -h 25
```

# std::Locate

## Synopsis

```
std::Locate <locatestring>
```

## Description

This function locates an object described by <locatestring>. This string must be understood by a DataServer.

# std::MessageDialog

Display a modal message dialog.

## Synopsis

```
std::MessageDialog ?-name caption? ?-message message? ?-style style?
?-icon icon?
```

## Description

This command displays a modal message dialog with a title set to *caption* and a message text equal to *message*. The two following arguments, *style* and *icon*, control the button configuration and what icon to display in the dialog. All arguments are optional.

The return value mirrors the button pressed by the user when leaving the dialog. Possible return values are 1 for ok, 2 for cancel, 4 for retry, 6 for yes and finally 7 for no.

*Style* can take on any of the following values:

- **ok**
  An OK button is provided. This is the default behavior.
- **okcancel**
  An OK button and a cancel button is provided.
- **retrycancel**
  A retry button and a cancel button is provided.

- **yesno**
  A yes button and a no button is provided.
- **yesnocancel**
  A yes button, a no button and a cancel button is provided.

*Icon* can take on any of the following values:

- **stop**
  A stop icon is provided.
- **question**
  A question icon is provided.
- **warning**
  A warning icon is provided.
- **information**
  An information icon is provided. This is the default behavior.

**Example 695:** ———————————————————————————————

The following example pops up a question dialog and stores the users answer, yes (6), no (7) or cancel (2), in a variable.

```
set res [std::MessageDialog -name "Telelogic Tau" -
message "Save changes?" -style yesnocancel -icon
question]
```

———————————————————————————————————————————

# std::OpenDocument

Open document.

## Synopsis

**std::OpenDocument** *filename*

### Description

This command opens a document, specified with the argument *filename*, in Tau. The most common use of the command is for loading a Tau workspace (a `.ttw` file) or Tau project (a `.ttp` file), but it can also be used to load and display for example plain text documents. The file extension controls the result of a load.

If the stated file can not be opened, a Tcl error will be generated.

**Example 696:**

A Tau workspace is opened like this:

```
std::OpenDocument "C:\\Work\\MyProjects.ttw"
```

## std::Output

Print message in Tcl output tab.

### Synopsis

```
std::Output <text>
```

### Description

The Output command prints a *message* in the Tcl output tab of the Output window.

**Example 697:**

This example prints the string “Hello World” followed by a new line in the Tcl output tab.

```
std::Output "Hello World\n"
```

# std::OutputTab

## Synopsis

```
std::OutputTab clear [clear [-ident <tabident> | <tabname>] |
activate [-ident <tabident>] | <tabname>]
```

## Description

Clears the content of an output tab or activates an output tab.

**Example 698:**

```
set selection [std::GetSelection]
std::ReportInit MyTab -ident mt MyFirst 100 0 MySecond 200 0
std::Report $selection -ident mt "This is the second column."
std::OutputTab clear -ident mt
```

# std::Report

## Synopsis

```
std::Report <sourceobject> [-ident <tabident>] [<string>]*
```

## Description

This command is used for adding new lines in a report tab.

The <sourceobject> will be used to fill the first column (icon + text) automatically.

The <string> values are labels, used to fill the remaining columns.

**Example 699:**

```
set selection [std::GetSelection]
std::ReportInit MyTab -ident mt MyFirst 100 0 MySecond 200 0
std::Report $selection -ident mt "This is the second column."
```

# std::Quit

Equivalent to the “close window” button.

### Synopsis

**std::Quit**

### Description

The `std::Quit` command posts a “close” message to the main window of Tau. This is equivalent to clicking the “close window” button on the upper right-hand side of the main window.

**Example 700:**

```
std::Quit
```

**Note**

*This does not guarantee exit. There can be some files still open that need to be saved, in which case Tau will pop-up a “Do you want to save?” message box.*

## std::ReportInit

### Synopsis

```
std::ReportInit <tabname> [-ident <tabident>] [-check] [-cb
<filename>] [<columnlabel> <columnwidth> <columnalignment>]+
```

### Description

This command is used to create a new report tab in the output window.

If the `-check` is used, a check box is displayed before each result line. These check boxes allow a more powerful navigation: only the checked lines will be selected when navigating with the F4 key, others are ignored.

If the `-cb <filename>` is present, then the associated Tcl script file will be evaluated when a double click on a line object occurs and the `OnDoubleClick proc` of this TCL script will be called with the double-clicked object as parameter.

For each `<columnlabel> <columnwidth> <columnalignment>` item, a new column is created with the corresponding label, width and alignment. An alignment of 0 means left align and an alignment of 1 means right. For sorting, columns right aligned are considered as number and columns left aligned are considered as text.

**Example 701:**

```
set selection [std::GetSelection]
std::ReportInit MyTab -ident mt MyFirst 100 0 MySecond 200 0
std::Report $selection -ident mt "This is the second column."
```

## std::SaveAll

Save workspace or project.

### Synopsis

**std::SaveAll** ?*filePath*?

### Description

This command saves the entire content of a workspace. Optionally, in case the *filePath* of a loaded project is specified, the entire content of the project is saved.

**Example 702:**

The complete workspace is saved like this:

```
std::SaveAll
```

## std::TextReport

Open a file and locate a position.

### Synopsis

**std::TextReport** <filename> [<line>] [<column>]

### Description

This command opens the file <filename>, and locates (puts the position of the cursor) to a <line> if specified, and to a <column> if specified. A line can be specified without specifying any column.

**Example 703:**

```
std::TextReport [std::GetLocaleDirectory]/java.ini 5 0
```

## std::View

Controls the desktop area.

### Synopsis

```
std::View [maximize | restore | minimize | close | coordinates]
[<view>]
activate [<view>|next]
getactive
count
```

### Description

The std::View command allows you to control the windows canvas area.

- `maximize` - maximizes the specified view.
- `restore` - restores minimized view to its previous size.
- `minimize` - minimizes the specified view.
- `close` - closes the specified view.
- `coordinates` - returns the coordinates of the specified view.
- `activate` - activates the specified view or the “next” view.
- `getactive` - returns a handle to the active view to be used in the above options.
- `count` - returns the number of open views.

**Example 704:**

```
set activeview [std::View getactive]
std::View minimize $activeview
```

# User Interface Add-in Specific Commands

The following commands are used by Tcl scripts run by add-ins in order to customize the user interface of Tau, typically extending it to provide a user interface for an add-in.

| Command | Description |
|---|---|
| std::AddCommand | Add new command. |
| std::AddContextMenu | Add shortcut menu. |
| std::AddMenu | Add menu. |
| std::AddToolbar | Add toolbar. |
| std::Declare | Separate definition and use. |

Usage of the commands in this section requires loading of the `commands` package into the Tcl interpreter as follows:

```
package require commands
```

The user interface add-in specific customization commands are only available from Tcl scripts executed via Add-Ins. They are not available in Tcl scripts outside add-ins, for example in Tcl agents or when running a standalone Tcl script.

## std::AddCommand

Add new command.

### Synopsis

```
package require commands

std::AddCommand -variable var -name name -statusmessage message
-tooltip tooltip -accelerator accelerator -imagefile imagefile
-onactivatecommand activateproc ?-onenablecommand enableproc?
?-oncheckcommand checkproc?
```

### Description

This command defines a new command, identified by *var*, allowing it to be called from a menu and/or a toolbar. A command needs to be defined before any call to for example std::AddMenu or std::AddToolbar is done.

The text to display for the command in a menu, in the Tau status bar and in a tool tip, are specified by arguments *name*, *message*, and *tooltip*, respectively. An accelerator, or shortcut key, for the command is defined by the argument *accelerator*, using a string that contains a combination of “Ctrl+”, “Shift+” and/or “Alt+”, followed by a character (in upper case regardless if Shift is part of the accelerator command). In addition, an image to display in a toolbar can be specified by a path to a bitmap file, relative to the Tau bin directory, using the argument *imagefile*.

*Activateproc* is the name of the procedure to call when the command is invoked. Optionally, a name of procedure controlling whether the commands menu item or toolbar button should be displayed or not can be specified vith the *enableproc* argument. Similarly, *checkproc* controls whether the menu item or toolbar button is ticked or not. These three procedures must all accept one argument, which at run-time will be assigned the name of the command. In addition *enableproc* and *checkproc* must both return a boolean value (a numeric value, where 0 is false and anything else is true, or a string value such as true or yes for true and false or no for false).

**Example 705:** ———————————————————————————

The following example shows a command definition making use of all possible arguments.

```
package require commands

proc OnTest { cmd } {
  std::Output "OnTest was called\n"
}

proc OnEnableTest { cmd } {
  return 1
}

proc OnCheckTest { cmd } {
  return 0
}

std::AddCommand -variable cmdTst -name "Test" -
statusmessage "To test a command" -tooltip "Test
command" -accelerator "Ctrl+Shift+Z" -imagefile
```

```
"../addins/Tst/etc/Tst.bmp" -onactivatecommand OnTest -
onenablecommand OnEnableTest -oncheckcommand OnCheckTest
```

### See also

"std::Declare" on page 2219

"std::AddContextMenu" on page 2216

"Additional tasks" on page 2455 in Chapter 91, *Dialog Help* for managing command syntax.

## std::AddContextMenu

Add shortcut menu.

### Synopsis

```
package require commands

std::AddContextMenu -variable menuvar -commands commands
-onenablemenu enableproc
```

### Description

This command adds a shortcut menu identified by the *menuvar* argument. The *commands* argument is a list of command identifiers representing the commands to present in the shortcut menu, in their order of appearance. To insert a separator between two menu items, use the command name "separator". The procedure controlling whether the shortcut menu should be displayed or not is specified with the *enableproc* argument. This procedure has one argument; the identity of the window from where the shortcut menu is called, for example "view", "output tab" or "browser tab", and returns a boolean value.

**Example 706:**

The example below shows how to add a shortcut menu that is displayed in case it is invoked from the File View.

```
package require commands

proc OnMenuEnable { ident } {
  if { $ident == "FILEVIEW" } {
```

```
    return 1
  } else {
    return 0
  }
}

std::AddContextMenu -variable cmTest -commands { cmdTst
} -onenablemenu OnMenuEnable
```

**See also**

"std::AddCommand" on page 2214

"std::Declare" on page 2219

"Additional tasks" on page 2455 in Chapter 91, *Dialog Help* for managing command syntax.

## std::AddMenu

Add menu.

### Synopsis

```
package require commands

std::AddMenu -variable menuvar -commands commands -path path
?-position pos?
```

### Description

This command adds a menu identified with the argument *menuvar*. The *commands* argument is a list of command identifiers representing the commands to present in the menu, in their order of appearance. To insert a separator between two menu items, use the command name "separator". The position of the menu is controlled with the argument *path*, which is a list of strings, starting with a menu name. Non-existing elements in the path are created automatically.

An optional argument, *pos*, can be used to control the positioning of the menu. *Pos* can take any of the following forms:

- **after** `menu`
  The new menu item will be inserted after *menu*, which is a sibling menu item identifier.

- **first**
  The new menu item will be inserted as the first child of its parent.
- **last**
  The new menu item will be inserted as the last child of its parent.

**Example 707:**

The example below shows how to add a separator and a menu item, for an added command called `cmdTest`, last in the Tools menu.

```
package require commands

std::AddMenu -variable mTest1 -commands { separator
cmdTst } -path { &Tools } -position last
```

**Example 708:**

Another example illustrates how to make a command accessible via a sub-menu item in the Tools menu.

```
std::AddMenu -variable mTest2 -commands { cmdTst } -path
{ &Tools Test }
```

### See also

"std::AddCommand" on page 2214

"std::Declare" on page 2219

## std::AddToolbar

Add toolbar.

### Synopsis

```
package require commands

std::AddToolbar -variable tbvar -commands commands
```

### Description

This command adds a toolbar identified with the argument *tbvar*. The *commands* argument is a list of command identifiers representing the commands to present in the toolbar, in their order of appearance. To insert a separator between two toolbar buttons, use the command name “separator”.

**Example 709:**

A toolbar with a button associated with the command `cmdTest` is added like this:

```
package require commands
std::AddToolbar -variable tbTest -commands { cmdTst }
```

## See also

“std::AddCommand” on page 2214

“std::Declare” on page 2219

# std::Declare

Separate definition and use.

### Synopsis

```
package require commands
std::Declare option
```

### Description

This command allows definition and use of for example a command to be separated from each other. In this way, a definition can be done in the beginning of a file, or in another file, while being used elsewhere.

The *option* argument can take on any of the following values:

- **addcommand** `args`
  A command will be declared, with *args* representing the arguments used for std::AddCommand.

- **addcontextmenu** `args`
  A command will be declared, with *args* representing the arguments used for std::AddContextMenu.
- **addmenu** `args`
  A command will be declared, with *args* representing the arguments used for std::AddMenu.
- **addtoolbar** `args`
  A command will be declared, with *args* representing the arguments used for std::AddToolbar.

**Example 710:**

An example of how to separate a command definition and its use looks like this:

```
package require commands

std::Declare addcommand -variable cmdTst -name "Test" -
statusmessage "For testing" -tooltip "Test command" -
accelerator "CTRL+SHIFT+Z" -imagefile "" -
onactivatecommand OnTest

proc OnTest { cmd } {
std::Output "OnTest was called\n"
}

proc Init{} {
  std::AddCommand cmdTst

  std::AddMenu -variable mTest -commands { separator
cmdTst } -path { &Tools } -position last
}
```

## See also

"std::AddCommand" on page 2214

"std::AddContextMenu" on page 2216

"std::AddMenu" on page 2217

"std::AddToolbar" on page 2218

# Model Commands

The majority of the Tcl commands available for model access are the same as the corresponding COM API methods, found in the ITtdModel section.

| | |
|---|---|
| u2::FindByGuid | Finds the entity with the specified GUID. |
| u2::New | Creates a new entity. |
| u2::Parse | Parses a string with U2P syntax and returns the resulting entities. |
| u2::XMLDecode | Decodes a piece of model encoded as XML and returns the resulting entities. |
| u2::Save | Saves the model. |
| u2::CreateResource | Creates a new resource (a file) in the model. |
| u2::LoadFile | Loads a UML model file (.u2) into the model. |
| u2::InvokeAgent | Invokes an agent programmatically on a specified model context. |

**Example 711:**

The mapping of optional COM parameters to Tcl options is illustrated with the following COM model access command:

```
HRESULT Parse(
          [in] BSTR strConcreteSyntax,
          [in, optional] VARIANT parseAs,
          [out, retval] ITtdEntities** ppEntities);
```

The corresponding Tcl command looks as follows:

```
u2::Parse ITtdModel strConcreteSyntax ?-parseAs value?
```

This command returns a reference to a model fragment. Two parameters, a model reference, *ITtdModel*, and a string containing a textual description, *strConcreteSyntax* need to be given. Optionally, *value*, can be used as a hint for the parser on how it should attempt to parse the provided string. It could be called as follows:

```
set expr [u2::Parse $model "x = 10" -parseAs
```

```
"Expression"]
```

The Tcl command `InvokeAgent` is a bit special since the corresponding COM method uses an in/out parameter to pass agent parameters to and from the invoked agent:

```
u2::InvokeAgent ITtdModel agent modelContext ?in/out <list>?
```

This means that instead of passing a Tcl string for this parameter you should pass a Tcl variable holding a list of agent parameters. The strings of this list will be interpreted by the called agent as values of certain types, according to the following rules (in priority order):

1. A Tcl id string representing an object realizing the ITtdEntity interface will be interpreted as an entity.
2. The strings “0” and “1” will be interpreted as the boolean values `false` and `true` respectively.
3. Any numeric string will be interpreted as an integer value.
   To disambiguate the integers `0` and `1` from the boolean values `false` and `true`, you may use the strings “00” and “01” instead.
4. A string that is a well-formed Tcl list with all elements being objects realizing the ITtdEntity interface, will be interpreted as a list of entities.
   To disambiguate an ITtdEntity object from a list of ITtdEntity objects (the Tcl representation for these are identical) you may add an extra “0” to the list. These extra “0”:s will not be translated to NULL pointers of type ITtdEntity, but they are allowed in order to be able to specify lists containing exactly one object.
5. If none of the above rules apply, the string will be interpreted as a plain string.

**Example 712:**

This example shows how to call an agent referenced by `$agent` on a model context referenced by `$modelContext`. The agent will receive 5 actual arguments, of the following types:

1. The boolean value `true`
2. The integer value `2`
3. The entity referenced by `$entity`
4. A list of entities containing the entity referenced by `$entity`

5. The string "`hello`".

After the agent has been invoked the resulting parameter list is printed.

```
set p [lappend p 1 2 $entity [list $entity 0] "hello"]
u2::InvokeAgent $model $agent $modelContext p
output "$p\n"
```

### See also

"Mapping of COM to Tcl commands" on page 2186

## u2::SelectMetaModel

Select which metamodel to be active.

### Synopsis

**`u2::SelectMetaModel`** *`metaModelPackage`*

### Description

This command makes it possible to control which metamodel to be active in a model. It corresponds to the **View - Reconfigure Model View** menu command. See Reconfigure ModelView for more information.

The argument to the command should be a reference to a package with the <<metamodel>> stereotype applied.

**Example 713:**

Setting TTDDiagramView as active metamodel in the first model found in the current workspace.

```
set model [lindex [std::GetModels -kind U2] 0]
set mm [u2::FindByName $model "TTDDiagramView"]
u2::SelectMetaModel $mm
```

# Entity Commands

Tcl commands for entity access are the same as the corresponding methods available in the COM API methods, found in the ITtdEntity section. The first Tcl argument for all entity access commands is a reference to a model entity.

| | |
|---|---|
| u2::ApplyStereotype | Instantiates the given stereotype and applies it on an entity. |
| u2::Bind | Binds all references in a model fragment, or a single reference of an entity. |
| u2::Clone | Creates a clone of an entity. |
| u2::Create | Creates a new entity in the context of an entity, that is adds a new direct or indirect child to an entity. |
| u2::CreateInstance | Creates an instance of a Signature. |
| u2::Delete | Deletes an entity from the model. |
| u2::FindByName | Performs a name-lookup from the context of an entity to find another entity by its (qualified) name. |
| u2::GetContainerMetaFeature | Returns the name of the metafeature in which an entity is contained. |
| u2::GetDescriptiveName | Returns a description of an entity. |
| u2::GetEntities | Returns the value of a metafeature for an entity as a collection of entities. |
| u2::GetEntity | Returns the value of a metafeature for an entity as an entity. |
| u2::GetMetaClassName | Returns the name of the entity's Metaclass. |
| u2::GetModel | Returns the model to which an entity belongs. |
| u2::GetOwner | Returns the composition owner of an entity. |
| u2::GetReference | Returns the identifier of a metafeature representing a reference. |
| u2::GetReferringEntities | Returns a collection of entities that refer to an entity through a particular metafeature. |

| | |
|---|---|
| u2::GetTaggedValue | Returns the specified property (tagged value) of an element. Can also be used to obtain an arbitrary value from an instance representation. |
| u2::GetValue | Returns the value of a metafeature for an entity as a string. |
| u2::HasAppliedStereotype | Determines if an entity has a certain stereotype applied. |
| u2::IsKindOf | Determines if an entity is of a particular Metaclass kind. |
| u2::MetaVisit<br><br>Note: This command corresponds to the COM method MetaVisitEx. | Traverses a model fragment and calls a method in a callback interface for each entity it contains.<br>See Example 714 on page 2226 for an example on how to use this command. |
| u2::Move | Moves an entity from its current location in the model to another owner. |
| u2::Replace | Replaces an entity with another entity. |
| u2::SetEntity | Sets the value of a metafeature for an entity as an entity. |
| u2::SetTaggedValue | Sets a property (tagged value) on an element. Can also be used to set an arbitrary value of an instance representation. |
| u2::SetValue | Sets the value of a metafeature for an entity as a string. |
| u2::UnlinkFromOwner | Unlinks an entity from it current owner in the model. |
| u2::Unparse | Unparse of an entity into a concrete syntax representation. |
| u2::XMLEncode | Encodes an entity into an XML representation. |

The Tcl usage for one of the entity access commands, MetaVisit, differs slightly from COM. Instead of specifying the callback function with a pointer to an interface, the Tcl command accepts a procedure. The example below shows this in practice.

**Example 714: Traverse of model entities**

This is how all model entities in all currently loaded models are traversed. For each model entity visited, a check is done to see whether the element in question is a Definition. If so, an output message is generated.

```
proc PrintDef { def } {
  if { [u2::IsKindOf $def "Definition"] } {
    set name [u2::GetValue $def "Name"]
    std::Output "Found def: $name\n"
  }
}

u2::MetaVisit [std::GetModels -kind U2] PrintDef
```

**See also**

"Mapping of COM to Tcl commands" on page 2186

# Resource Commands

A resource represents the physical storage unit where a model is stored, typically a text file with the extension '.u2'. Tcl commands for resource handling are the same as the corresponding methods available in the COM API methods, found in the ITtdResource section. The first Tcl argument for all resource handling commands is thus a reference to a resource.

**Note**

*Since a Resource also is an Entity, all Entity Commands are also available for resources.*

| Command | Description |
|---|---|
| u2::SaveResource<br><br>Note: This command corresponds to the COM method Save. | Saves the model entities associated with the resource (typically saves the corresponding `.u2` file). |

**See also**

"Mapping of COM to Tcl commands" on page 2186

# Presentation Element Commands

A presentation element is an element with a graphical appearance, for example a diagram, symbol or a line. Tcl presentation element commands are the same as the corresponding methods available in the COM API methods, found in the ITtdPresentationElement section. The first Tcl argument for all presentation element commands is thus a reference to a presentation element.

**Note**

*Since a Presentation Element also is an Entity, all Entity Commands are also available for presentation elements.*

| | |
|---|---|
| u2::GenerateEMF | Generates an EMF file (Enhanced Meta File) for a presentation element (deprecated). |
| u2::GenerateEMFEx | Generates an EMF file (Enhanced Meta File) for a presentation element with support for scaling. |
| u2::GenerateImage | Generates an image file for a presentation element. |

## u2::GenerateEMF

This command generates an EMF file (Enhanced Meta File) for the graphical appearance of a presentation element. This is a deprecated function. Use `u2::GenerateEMFEx` instead.

### Synopsis

```
u2::GenerateEMF ITtdPresentationElement strFileName ?-maxWidth
<Integer>? ?-maxHeight <Integer>? ?-optimizeForVectorGraphics
<Boolean>? ?-includeFrame <Boolean>?
```

### Description

The presentation element will have the same appearance in this EMF file as when shown in the tool's editors.

### Parameters

The `strFileName` argument is the file name of the EMF file to generate. If `strFileName` is a relative path, it will be interpreted as relative to the current working directory of the client application.

Another four optional arguments are accepted.

The `maxWidth` and `maxHeight` parameters can be used to specify the maximum size of the generated image. The image will be scaled to fit into the specified size. If the parameters are omitted, the generated image will be the same size as shown in the tool's editors. The unit of the height and width numbers is 1/10:th of a millimeter. These parameters are currently only considered if the presentation element on which the method is called is a diagram.

If `optimizeForVectorGraphics` is set to true, the EMF generation will be optimized for vector graphics. The default behavior is to not do this optimization. This parameter is considered deprecated and exists for backward compatibility purposes.

The parameter `includeFrame` specifies whether the frame symbol of a diagram should be included in the EMF generation or not. By default it will be included. This parameter is currently only considered if the presentation element on which the method is called is a diagram.

### Return value

The command does not return a value.

# u2::GenerateEMFEx

This command generates an EMF file (Enhanced Meta File) for the graphical appearance of a presentation element. This is the recommended command to use in order to generate an EMF file for a presentation element.

## Synopsis

```
u2::GenerateEMFEx ITtdPresentationElement strFileName ?-maxWidth
<Integer>? ?-maxHeight <Integer>? ?-optimizeForVectorGraphics
<Boolean>? ?-includeFrame <Boolean>? ?-scaleFactor <Integer>?
```

## Description

The presentation element will have the same appearance in this EMF file as when shown in the tool's editors.

## Parameters

The `strFileName` argument is the file name of the EMF file to generate. If `strFileName` is a relative path, it will be interpreted as relative to the current working directory of the client application.

Another four optional arguments are accepted.

The `maxWidth` and `maxHeight` parameters can be used to specify the maximum size of the generated image. The image will be scaled to fit into the specified size. The image will be scaled to fit into the specified size if no specific `scaleFactor` is given. If the parameters are omitted, the generated image will be the same size as shown in the tool's editors. The unit of the height and width numbers is 1/10:th of a millimeter. These parameters are currently only considered if the presentation element on which the method is called is a diagram.

If `optimizeForVectorGraphics` is set to true, the EMF generation will be optimized for vector graphics. The default behavior is to not do this optimization. This parameter is considered deprecated and exists for backward compatibility purposes.

The parameter `includeFrame` specifies whether the frame symbol of a diagram should be included in the EMF generation or not. By default it will be included. This parameter is currently only considered if the presentation element on which the method is called is a diagram.

If the optional parameter `scaleFactor` is given the original diagram is scaled before generating any images. The scale factor should be given as an integer and is interpreted as percent of the original diagram size. If both the `scaleFactor` and `maxWidth`/`maxHeight` are given as arguments then more than one image may be generated as a result of the operation. If the `scaleFactor` parameter is given then the name of the generated files is the same as the `strFileName` parameter but with a number added before the file extension.

### Return value

The command does not return a value.

## u2::GenerateImage

This command generates an image file of a specified kind for the graphical appearance of a presentation element. The presentation element will have the same appearance in this image file as when shown in the tool's editors.

### Synopsis

```
u2::GenerateImage ITtdPresentationElement imgKind strFileName
```

### Description

The presentation element will have the same appearance in this image file as when shown in the tool's editors.

### Parameters

The `imgKind` parameter specifies which kind of image file to generate. The table below lists valid values for this parameter.

| imgKind value | Description |
|---|---|
| `JPEG` | Generate a JPEG image file. |
| `BMP` | Generate a BMP image file. |
| `GIF` | Generate a GIF image file. |
| `TIFF` | Generate a TIFF image file. |

| imgKind value | Description |
|---|---|
| TARGA | Generate a TGA (Targa) image file. |
| DIB | Generate a device independent bitmap file. |
| PCX | Generate a PCX image file. |

The strFileName argument is the file name of the image file to generate. If strFileName is a relative path, it will be interpreted as relative to the current working directory of the client application.

**Return value**

The command does not return a value.

**See also**

"Mapping of COM to Tcl commands" on page 2186

# Symbol Commands

A symbol is a presentation element with a two-dimensional graphical appearance. Tcl commands for working with symbols are the same as the corresponding methods available in the COM API methods, found in the ITtdSymbol section. The first Tcl argument for all symbol handling commands is thus a reference to a symbol.

**Note**

*Since a Symbol also is a PresentationElement and an Entity, all Presentation Element Commands and all Entity Commands are also available for symbols.*

| Command | Description |
|---|---|
| u2::SetSize | Sets the size of the symbol. |
| u2::SetPosition | Sets the position of the symbol. |

**See also**

"Mapping of COM to Tcl commands" on page 2186

# Expression Commands

Expressions may appear at various places in a model. Tcl commands for working with expressions are the same as the corresponding methods available in the COM API methods, found in the ITtdExpression section. The first Tcl argument for all symbol handling commands is thus a reference to an expression.

**Note**

*Since an Expression also is an Entity, all Entity Commands are also available for expressions.*

| Command | Description |
|---|---|
| u2::GetType | Computes the type of the expression. |
| u2::EvaluateConstantIntegralExpression | Evaluates the integral value of a constant expression. |
| u2::GetInstanceChildExpression | Finds a child expression of an instance. |

**See also**

"Mapping of COM to Tcl commands" on page 2186

# Library Handling Commands

The library handling part contains the following commands:

| Command | Description |
|---|---|
| u2::LoadLibrary | Load UML Library. |

| | |
|---|---|
| u2::UnloadLibrary | Unload UML library. |
| u2::LoadProfile | Load UML profile. |
| u2::UnloadProfile | Unload UML profile. |

**See also**

"LoadFile" on page 2069

## u2::LoadLibrary

Load UML Library.

### Synopsis

```
u2::LoadLibrary filename
```

### Description

This command loads a UML library into Tau. The library is specified with the argument *filename*, which is a string specifying the path to a `.u2` file containing a UML library.

**See also**

"u2::UnloadLibrary" on page 2233.

## u2::UnloadLibrary

Unload UML library.

### Synopsis

```
u2::UnloadLibrary packageRef ?-deleteDependencies?
```

### Description

This command unloads a loaded UML library. The library is specified with the argument *packageRef*, which must contain a reference to a UML library package.

If the `-deleteDependencies` option is given top-level dependencies to the library package will also be removed. This can be useful if the unloaded library is a profile, since top-level `<<access>>` dependencies then have been added automatically when loading it. However, note that all top-level dependencies referring to the package will be deleted, even those added manually by the user.

**Example 715:**

A loaded library can be unloaded in a special procedure, `BeforeUnload`, which is called upon an add-in deactivation. A library needs to be unloaded once for each loaded model as follows:

```
proc BeforeUnload { } {
  set models [std::GetModels -kind U2]

  foreach model $models {
    set profile [u2::FindByGuid $model "@MyProfile"]
    u2::UnloadLibrary $profile
  }
}
```

## u2::LoadProfile

Load UML profile.

### Synopsis

**`u2::LoadProfile`** *filename*

### Description

Use `LoadLibrary` instead. `LoadProfile` is a deprecated function and will be removed in future versions.

This command loads a UML profile into Tau. The profile is specified with the argument *filename*, which is a string specifying the path to a `.u2` file containing a UML profile.

### See also

"u2::UnloadProfile" on page 2235.

## u2::UnloadProfile

Unload UML profile.

### Synopsis

**u2::UnloadProfile** *packageRef*

### Description

Use `UnLoadLibrary` instead. `UnLoadProfile` is a deprecated function and will be removed in future versions.

This command unloads a loaded UML profile. The profile is specified with the argument *packageRef*, which must contain a reference to a UML profile package.

**Example 716:**

A loaded profile can be unloaded in a special procedure, `BeforeUnload`, which is called upon an add-in deactivation. A profile needs to be unloaded once for each loaded model as follows:

```
proc BeforeUnload { } {
  set models [std::GetModels -kind U2]

  foreach model $models {
    set profile [u2::FindByGuid $model "@MyProfile"]
    u2::UnloadProfile $profile
  }
}
```

# Semantic Checker Commands

The semantic checker commands provides a possibility to add user defined UML semantic checks to the set of built-in semantic checks. A typical scenario is to add semantic checks that tests constraints defined in a UML library.

The following semantic checker commands are provided:

| Command | Description |
|---|---|
| u2::Check | Check model semantically. |
| u2::CreateSemGroup | Creates a new semantic checker group. |
| u2::CreateSemRule | Create new semantic checker rule. |
| u2::DeleteSemEntity | Delete semantic checker group or rule. |
| u2::EnableSemEntity | Enables or disables a semantic entity. |
| u2::GetSemEntities | Returns names of entities in a group. |
| u2::IsSemEntityEnabled | Returns the status of a semantic entity. |
| u2::IsSemGroup | Check if entity is a semantic group. |
| u2::QuickCheck | Check model semantically. |
| u2::SemMessage | Report semantic checker error. |

# u2::Check

Check model semantically.

### Synopsis

**u2::Check** *entityRef*

### Description

This command performs a complete semantic check of a hierarchy of model elements, specified with the *entityRef* argument which must be a reference to a model or entity.

**Example 717:**

The example shows how to perform a check on all currently loaded models.

```
set models [std::GetModels]

foreach m $models {
  u2::Check $m
}
```

## u2::CreateSemGroup

Creates a new semantic checker group.

### Synopsis

```
u2::CreateSemGroup path name
```

### Description

This command creates a new semantic checker group with a name specified by the *name* argument.

The *path* argument is a string that defines the groups location in the semantic checker group hierarchy. The string must start with a '/', denoting the root group, followed by any number of potential parent groups, all separated by '/'. If a parent group in the *path* does not exist, a Tcl error will be generated. No error will be reported if the group already exists.

"u2::CreateSemRule" on page 2237

## u2::CreateSemRule

Create new semantic checker rule.

### Synopsis

```
u2::CreateSemRule path name metaclass priority script
```

### Description

This command defines a new semantic checker rule, using the argument *path* to specify the location of the new rule in the semantic checker group hierarchy and *name* to set the name of the rule.

The **metaclass** argument determines that the new rule would be invoked only on entities of that Metaclass.

The *priority* sets the priority of the rule compared with other rules

The *script* argument is the name of a Tcl script to call when rule is executed. A Tcl id of the model entity is appended at the end of script, so, it is recommended that the script has a return parameter. If *script* removes the model en-

tity on which it is invoked in the process of transformation, it should return either the value ‘1’ or ‘true’. This allows the semantic analysis not to call other rules on this entity. Failure to return ‘1’ or ‘true’ after deletion of the model entity may cause the program to malfunction.

## u2::DeleteSemEntity

Delete semantic checker group or rule.

### Synopsis

```
u2::DeleteSemEntity path name
```

### Description

This command deletes a semantic checker group, called *name,* located in the semantic checker group hierarchy as specified by *path*. **Predefined** rules cannot be deleted with this command.

## u2::EnableSemEntity

Enables or disables a semantic entity.

### Synopsis

```
u2::EnableSemEntity path status
```

### Description

The *path* argument is a string that defines the entity location in the semantic checker group hierarchy.

*Status* is a string argument which sets the desired status of an entity:

- **enabled**
- **disabled**

## u2::GetSemEntities

Returns names of entities in a group.

### Synopsis

```
u2::GetSemEntities path
```

### Description

The command **GetSemEntities** returns a list consisting of names of the semantic sub-entities in a given group.

The *path* argument is a UNIX style path to the semantic group. For example, the path to the root group is '/', to 'UML' group (which lies in root group) is '/UML'.

As a consequence of such naming convention, names of semantic entities cannot contain a **slash** ('/'), because it would be considered to be a path delimiter. There exists no possible escape character for the slash character, the semantic checker will always treat a slash as a separator.

## u2::IsSemEntityEnabled

Returns the *status* of a semantic entity.

### Synopsis

```
u2::IsSemEntityEnabled path
```

### Description

Returns the *status* of a semantic entity specified by *path*.

The *path* argument is a string that defines the entity location in the semantic checker group hierarchy.

*status* is a string which is the status of an entity:

- **enabled**
- **disabled**

## u2::IsSemGroup

Check if entity is a semantic group.

### Synopsis

```
u2::IsSemGroup path
```

### Description

Returns **true** or **false**, depending on if semantic entity specified by *path* is semantic group (true) or not (false).

The *path* argument is a string that defines the entity location in the semantic checker group hierarchy.

## u2::QuickCheck

Check model semantically.

### Synopsis

```
u2::QuickCheck entityRef
```

### Description

This command performs a complete semantic check of a hierarchy of model elements, specified with the *entityRef* argument which must be a reference to a model or entity. A standard error list (the same as in u2::Check) is used. This command performs a subset of semantic checks, and only operates on the entity it is invoked on (any composition children will not be checked).

## u2::SemMessage

Report semantic checker error.

### Synopsis

```
u2::SemMessage severity message entityId
```

### Description

This command puts an error message into the error list used by Semantic Analyzer. This function should be used to report errors, warnings and information messages during checking, otherwise the semantic analysis will not stop after an error which is generated by a user-defined check.

*severity* is a string with the following possible values:

- **error**
  An error message is reported.
- **fatal**
  A fatal error message is reported.
- **information**
- **warning**
  A warning message is reported.

*message* is a string with the message which would be reported to the error list.

*entityId* is an optional reference to a model element to which the message is related.

**See also**

"u2::CreateSemRule" on page 2237

# Utility Interface Commands

The C++ and COM APIs of Tau are, in contrast to the Tcl API, based on a set of interfaces. The majority of these interfaces are implemented by meta classes of the Tau meta model (model interfaces), which means that functions in these interfaces have a corresponding Tcl command in one of the above mentioned group of commands. However, there are some API interfaces which are not implemented by any meta class (utility interfaces), but by other objects in the Tau IDE. To be able to access such functionality from Tcl there exists Tcl commands corresponding to the functions in such interfaces.

Tcl commands corresponding to functions in utility interfaces are described in this section.

| Command | Description |
|---|---|
| u2::AddSourceBufferText | Add text to a buffer of (source code) text. |
| u2::AddMessage | Add a message to a message list |

# u2::AddSourceBufferText

Add text to a buffer of (source code) text.

This command corresponds to the API function AddText of the ITtdSourceBuffer interface.

### Synopsis

```
u2::AddSourceBufferText ITtdSourceBuffer text
```

### Description

This command adds a piece of text to a buffer of text. Typically this buffer represents a file of source code currently being generated by a code generator.

The ITtdSourceBuffer reference is obtained as a parameter of an agent triggered by an interactive tool event.

**Example 718: Generating a header for a Java file**

Below is a sample implementation of a Tcl agent which may trigger on the JavaPrintFile tool event.

```
proc AddJavaHeader { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  u2::AddSourceBufferText [lindex $ap 0] "This is a
generated file! Do not edit!"
}
```

# u2::AddMessage

Add a message to a message list

This command corresponds to the API function AddMessage of the ITtdMessageList interface.

### Synopsis

```
u2::AddMessage ITtdMessageList text severity ?-subject ITtdEntity?
```

### Description

This command adds a message to a message list. Message lists are typically obtained as an agent parameter, and can for example represent messages produced during semantic checking or code generation.

The severity parameter should be one of the following strings

- **information**
  Specifies that the message is an information message.
- **warning**
  Specifies that the message is a warning.
- **error**
  Specifies that the message is an error.
- **fatal**
  Specifies that the message is a fatal error.

By using the -subject switch it is possible to attach a model entity as subject for the message. A message which has a subject entity can be double-clicked in an output tab in order to navigate to the entity.

**Example 719: Implementing a custom semantic check in Tcl**

Below is a sample implementation of a Tcl agent which may trigger on the Check tool event.

```
proc MyTclCheck { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  if {[u2::IsKindOf $context "Class"] } {
    if {[u2::GetEntity $context "Destructor"] == 0 } {
      u2::AddMessage [lindex $ap 0] "A class should have
a destructor." "warning" -subject $context
    }
  }
}
```

# 78

# *C++ API*

This chapter is the reference documentation of the Tau C++ API. From a functional point of view the C++API is very similar to the Tau COM API, so in many places the documentation for the COM API will be referenced.

Intended readers are developers of client applications that use the C++ API to access a UML model. These client applications could be everything from small interactive Add-Ins to full fledged code generators or import applications. A basic knowledge of C++ is assumed throughout this chapter.

# Introduction

The C++ API consists of a set of interfaces (abstract C++ classes), each with a number of member functions. The design of the API is very similar to the design of the COM API - in fact the COM API is implemented in terms of the C++ API.

A client of the C++ API may execute in different environments. The behavior of some API functions will depend on the execution environment of the client code. For example, if the client executes in the IDE (a so called interactive client), those API functions that modify the model will use an implementation that is appropriate for interactive use (undo/redo capabilities etc.). Another example is a client that executes in a batch application (a so called non-interactive client), for which some parts of the API will be inaccessible (those parts that have to do with the IDE).

# Accessing the API

A client that wants to use the C++ API should include the file `U2ModelAccess.h`. This file can be found in the installation at `/include/ToolAPI`. The client should also link with corresponding libraries that are provided in the Tau installation. For more detailed information about the platform specific steps that are required for setting up a client to use the C++ API, see "C++ API Set-up" on page 2278.

Once the client has been properly set-up for API usage, the API may be accessed. The way to access the API depends on whether the client is interactive or non-interactive.

### Accessing the API from non-interactive clients

A non-interactive client accesses the C++ API by calling the `u2::GetModelAccess` function. This function returns a pointer to a singleton object that implements the `u2::ITtdModelAccess` interface. See "ITtdModelAccess" on page 2250 for more information about this interface.

**Note**

*It is important to initialize the API before starting to use it from a non-interactive client. This is done by calling* `u2::InitializeModelAccess`. *After the last API call the API should also be finalized, by a call to* `u2::FinalizeModelAccess`. *It is allowed to repeat initialization and finalization as long as the calls are balanced.*

**Example 720**

Initializing the API, obtaining a pointer on the `u2::ITtdModelAccess` interface, and finalizing the API afterwards:

```
u2::InitializeModelAccess();
u2::ITtdModelAccess* pMA = u2::GetModelAccess();
u2::FinalizeModelAccess();
```

### Accessing the API from interactive clients

Interactive clients of the C++ API are typically Agents that receive a pointer on an `u2::ITtdEntity` interface representing the context of these agents. This pointer can be used as the starting point for accessing the model from that context. Thus it is not always necessary for an interactive client to access the API through the `u2::ITtdModelAccess` interface. However, the `u2::GetModelAccess` function is of course available also for interactive clients, and the object it returns will then be suited for use in an interactive execution environment.

API initialization and finalization is not necessary for an interactive client.

An interactive C++ client can be used to implement an Add-In module in a more efficient way than using a Tcl script. However, it is currently not possible to develop the entire Add-In using only a C++ client; Tau requires at least a minimal Tcl script to execute. This script can use the Tcl API command `InvokeAgent` in order to transfer execution to an agent implemented in C++.

## See also

"Accessing the API from interactive clients" on page 2065 for guidelines of how to decide which Add-in functionality that is best implemented in Tcl and C++ respectively.

## Important!

*An interactive C++ client runs in the same memory space as the Tau application. If it crashes, the Tau application will also crash. Make sure you have saved all changes before running an C++ agent under development.*

## Changer object

All API functions that modify the model take a 'changer' argument typed by a class called `Cu2Changer` (defined in the `u2dll` namespace). The purpose of this class is to be able to make model modifications in a way that works well both in an interactive environment (where the changes must be "logged" in order to allow undo and redo) and in a non-interactive environment (where changes can be done immediately without logging). The definition of `Cu2Changer` is not published in the API (only a forward declaration is present), and the reason for this is that the client code shall not use this class explicitly. All 'changer' arguments can be left unspecified and then defaults to a changer object that will perform the change immediately, without logging. An interactive client (such as an agent) receives a `Cu2Changer` object when invoked from the Tau IDE, and in order to make a model change that is possible to undo and redo, that changer object should be used when calling the C++ API functions.

## Interface casting

The mechanism for dynamically casting a pointer from one interface to another is based on a virtual function called `QueryInterface`. This function is defined in the `u2::IUnknown` interface, and is available in all interfaces since they all inherit from `u2::IUnknown`.

**Note**

*The C++ API has borrowed both the design and terminology of interface casting from COM, hence the names QueryInterface and IUnknown. Contrary to the COM IUnkown interface, u2::IUnknown does not use reference counting, i.e. there is no need to call AddRef or Release on interface pointers*

The C++ API provides a convenient utility template function `u2::cast` which facilitates interface casting with the familiar C++ cast syntax and semantics.

**Example 721**

Dynamic casting between interfaces using `u2::cast`.

```
u2::ITtdModelAccess* pMA = u2::GetModelAccess();
u2::ITtdModel* pModel = pMA->LoadFile(_T("x.u2"));
u2::ITtdEntity* pEntity = u2::cast<u2::ITtdEntity>(pModel);
if (pEntity) {
  // Casting succeeded! you may safely use pEntity here!
```

```
}
```

**Note**

*Do not attempt to do interface casting by using the standard* `dynamic_cast` *operator, as the Tau object model does not contain standard C++ run-time type information (RTTI).*

# Handling API Errors

The C++ API reports all errors by throwing an `u2::APIError` exception. Client code should be prepared for errors by catching this exception. A textual description of the error can be obtained by using the `u2::GetAPIErrorMessage` function.

**Example 722**

A typical catch clause for handling C++ API errors could look like this:

```
try {
  // access the C++ API
}
catch (u2::APIError e) {
  cout << "Error while using the Tau Developer C++ API:" << endl;
  wcout << u2::GetAPIErrorMessage(e) << endl;
}
```

On Unix use `cout` instead of `wcout` since `GetAPIErrorMessage` there returns a narrow (single-byte) string.

# Client restrictions

The C++ API imposes few restrictions on its clients. However, please keep in mind the following when designing the API client:

### Bare only

The API does not support safe simultaneous access of the model from multiple threads. Interactive clients will be executed in the main thread of the Tau application.

### Consistent string encoding

Strings passed into, or obtained as result from, C++ API functions are typed by the `tstring` type. This type is a wide (double-byte) Unicode string on Windows platforms, and a narrow (single-byte) ANSI string on Unix platforms. The API client is responsible for performing any string conversions that might be necessary on input or output strings.

# API Interfaces and Functions

All interfaces of the C++ API are placed in the `u2` namespace. This section lists all API interfaces and functions and gives a short description of each. Some examples of typical use are provided. Note, however, that the examples only serve as an illustration of how to use a particular API function or interface, and are not always complete. In particular the recommended error handling (see "Handling API Errors" on page 2249) is usually omitted from the examples for brevity reasons.

The names of the C++ interfaces and their member functions are the same as the corresponding interfaces and methods of the COM API. To get more detailed documentation or examples of a C++ API item, please refer to the corresponding item of the COM API documentation.

Many of the interfaces are implemented by classes representing metaclasses in the implementation of the UML Metamodel. Some of the methods in the C++ API require knowledge of the metamodel in order to be useful.

**Note**

*In this context a "C++ interface" means an abstract class, i.e. a class with all its member functions being pure virtual*

## ITtdModelAccess

The ITtdModelAccess interface contains functions that do not operate directly on the model. Use it from a non-interactive client to get access to a UML model by loading it from files (project file or `.u2` file), or create a new model from scratch. An interactive client may also use this interface to access certain parts of the Tau IDE.

See "Accessing the API" on page 2246 to learn how to obtain the `ITtdModelAccess` interface from the client application.

The COM API documentation contains a more detailed description of ITtdModelAccess.

### LoadProject

```
virtual u2::ITtdModel*
LoadProject(const tstring& strProject,
  bool bBind = true,
  ITtdMessageList* pMessages = 0)
throw(u2::APIError) = 0;
```

Loads the project stored in the specified project file (or URI). If the project file does not exist, or some other error occurs while loading the project, an APIError exception will be thrown. If `strProject` is a relative path, it will be interpreted as relative to the current working directory of the client application.

By default the model will be bound after loading, but this can be suppressed by setting `bBind` to false.

If a message list is passed to the function all messages that are produced during loading and binding will be added to that list. A message list can be obtained by calling GetMessageList.

**Note**

*If LoadProject is used from an interactive agent, the implementation uses the COM API LoadProject implementation. This for example means that the parameters 'bBind' and 'pMessages' have no effect and that no exception will be thrown in error situations (instead a new model will be created).*

**Example 723**

Loading a project from a non-interactive client:

```
u2::ITtdModelAccess* pMA = u2::GetModelAccess();
u2::ITtdModel* pModel = pMA->LoadProject(_T("x.ttp"));
```

Loading a project and using a message list for printing load and bind messages:

```
u2::ITtdMessageList* pMessages = pMA->GetMessageList();
u2::ITtdModel* pModel = pMA->LoadProject(_T("y.ttp"), true /* bBind
*/, pMessages);
tstring strErrors;
pMessages->GetDescription(strErrors, _T("\n"));
pMA->WriteMessage(strErrors);
delete pMessages;
```

### LoadFile

```
virtual u2::ITtdModel*
LoadFile(const tstring& strFile,
  bool bLibrary = false)
throw(u2::APIError) = 0;
```

Loads the specified `.u2` file (or URI). If the file does not exist, or some error occurs while loading it, an APIError exception will be thrown. If `strFile` is a relative path, it will be interpreted as relative to the current working directory of the client application. If `bLibrary` is true, the file will be loaded as a library.

See Example 721 on page 2248 for an example of how to use `LoadFile`.

### CreateModel

```
virtual u2::ITtdModel*
CreateModel()
throw(u2::APIError) = 0;
```

Creates a new empty model. If a new model cannot be created (for example due to a memory or license problem), an APIError exception will be thrown.

The created empty model can be used as a starting-point for creating a whole new UML model. The example below shows the creation of a simple model containing one package with one class. It also shows how to save the new model afterwards.

**Example 724**

Creating a new model and saving it to a `.u2` file:

```
u2::ITtdModel* pModel = pModelAccess->CreateModel();
u2::ITtdEntity* pModelRoot = u2::cast<u2::ITtdEntity>(pModel);
u2::ITtdEntity* pPackage = pModelRoot->Create(_T("Package"), false,
_T("OwnedMember"));
u2::ITtdEntity* pClass = pPackage->Create(_T("Class"));
u2::ITtdResource* pResource;
pResource = pModel->CreateResource(_T("test.u2"));
// Insert the created package pPackage as a root of pResource
u2::cast<u2::ITtdEntity>(pResource)->SetEntity(_T("Root"),
pPackage);
pModel->Save(); // Saves all resources in the model
```

### GetActiveProject

```
virtual ITtdModel*
GetActiveProject()
throw(u2::APIError) = 0;
```

Returns the model of the currently active project in the open workspace. This function shall only be used from interactive API clients. If it is used from a non-interactive client an APIError will be thrown.

### GetProjectItem

```
virtual ITtdModel*
GetProjectItem(long index)
throw(u2::APIError) = 0;
```

Returns the model of the project with the specified index in the open workspace. This function shall only be used from interactive API clients. If it is used from a non-interactive client an APIError will be thrown.

### GetProjectCount

```
virtual long
GetProjectCount()
throw(u2::APIError) = 0;
```

Returns the number of projects in the open workspace. This function shall only be used from interactive API clients. If it is used from a non-interactive client an APIError will be thrown.

### WriteMessage

```
virtual void
WriteMessage(const tstring& strMessage)
throw(u2::APIError) = 0;
```

Writes a message to the default message area (typically to `stdout` in a batch environment, or to the Messages tab in the IDE).

See Example 723 on page 2251 for an example of how to use `WriteMessage`.

### GetMessageList

```
virtual ITtdMessageList*
GetMessageList() = 0;
```

Creates and returns a new message list. This message list can be used as input to functions that support reporting of messages into an ITtdMessageList. If no message list can be created, NULL is returned.

It is the responsibility of the caller to delete the message list when it will not use it anymore.

See Example 723 on page 2251 for an example of how to use `GetMessageList`.

### GetDefaultMessageList

```
virtual ITtdMessageList*
GetDefaultMessageList() = 0;
```

Returns a default message list appropriate for using in the current execution environment. For example, in the context of a code generator the returned message list is mapped to the build log of the code generator.

The returned message list belongs to Tau and should not be deleted by the caller.

### GetLicense

```
virtual void
GetLicense(const tstring& strFeature)
throw (u2::APIError) = 0;
```

Requests a run-time license for a licensed feature with the specified name. If successul the license is taken. If not, an APIError exception will be thrown.

This function is intended to be used by API clients that require a license to be used.

## ITtdModel

The ITtdModel interface is implemented by the Session class of the Meta-model, which represents the top-level entity of a UML model.

The ITtdModel interface contains methods that do not need a specific model entity context for their execution. Typically these methods operate on the model as a whole, rather than on a particular entity.

**Note**

*Since a Session also is an Entity, a cast from ITtdModel to ITtdEntity will always succeed.*

The COM API documentation contains a more detailed description of ITtd-Model.

### FindByGuid

```
virtual u2::ITtdEntity*
FindByGuid(const tstring& strGuid) const
```

```
throw(u2::APIError) = 0;
```

Returns the entity in the model with the specified GUID, or NULL if no such entity exists.

The COM API documentation contains a more detailed description of FindByGuid.

### New

```
virtual u2::ITtdEntity*
New(const tstring& strMetaClass) const
throw(u2::APIError) = 0;
```

Creates a new instance of the specified Metaclass. If a metaclass with the specified name does not exist, or the operation fails for some other reason, an APIError exception is thrown.

The COM API documentation contains a more detailed description of New.

### Parse

```
virtual void
Parse(const tstring& strConcreteSyntax,
  std::list<u2::ITtdEntity*>& listEntities,
  const tstring& strParseAs = _T("Definition")) const
throw(u2::APIError) = 0;
```

Parses the specified piece of concrete textual UML syntax. The optional trailing parameter `strParseAs` specifies the grammar to use when parsing. By default, the Definition grammar will be used, i.e. the text should then specify one or many definitions. Other supported grammars are Expression and Action. In case a non-existing grammar is specified or the text contains syntax errors, an APIError exception will be thrown. The result built by the parser will be inserted into the list.

The COM API documentation contains a more detailed description of Parse.

### XMLDecode

```
virtual void
XMLDecode(const tstring& strXMLEncoding,
  std::list<u2::ITtdEntity*>& listEntities) const
throw(u2::APIError) = 0;
```

Decodes the XML encoded string into a list of model entities. If the decoding fails (e.g. because of a syntax error in the XML) an APIError will be thrown.

The COM API documentation contains a more detailed description of XML-Decode.

### Save

```
virtual void
Save() const
throw(u2::APIError) = 0;
```

Saves the model into its resources. If the model does not contain any resources, nothing will happen.

If the model cannot be saved an APIError exception will be thrown.

The COM API documentation contains a more detailed description of Save.

### CreateResource

```
virtual u2::ITtdResource*
CreateResource(const tstring& strFile,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger)
throw(u2::APIError) = 0;
```

Creates a new resource for the model. The resource will be a file with the specified file name. If the creation fails an APIError will be thrown. The model modification is performed using the changer.

See Example 724 on page 2252 for an example of using CreateResource.

The COM API documentation contains a more detailed description of CreateResource.

### LoadFile

```
virtual u2::ITtdResource*
LoadFile(const tstring& strFile,
  bool bLibrary = false)
throw(u2::APIError) = 0;
```

Loads the specified file into the model. A resource representing the file will be created and returned. If `bLibrary` is true, the file will be loaded as a library. In cases of error (e.g. load errors) an APIError will be thrown.

The COM API documentation contains a more detailed description of LoadFile.

### InvokeAgent

```
virtual void
InvokeAgent(u2::ITtdEntity* pAgent,
```

```
  u2::ITtdEntity* pModelContext,
  u2::AgentParameters& agentParameters) const
throw(u2::APIError) = 0;
```

Invokes the specified agent on the specified model context.

The `AgentParameters` type is a list of Agent Parameters, representing actual arguments passed to the invoked agent. For information on how to use the `AgentParameters` type see the file `U2Agent.h` (found in the Tau installation at `/include/ToolAPI`).

**Example 725**

Invoking an agent, passing a string and a boolean value as actual arguments. We assume here that `pAgent` is a pointer to the agent to be invoked, and `pContext` is a pointer to the entity that is the model context of the agent invocation.

```
u2::AgentParameters params; // Always allocate on caller's stack
params.push_back(u2::AgentParameter::Create(_T("Hello!")));
params.push_back(u2::AgentParameter::Create(true));

u2::ITtdModel* pModel = pAgent->GetModel();
pModel->InvokeAgent(pAgent, pContext, params);
```

Note that the invoked agent may be implemented in another language than C++. Conversion of the C++ representation of an agent parameter into the representation used in the technology with which the agent is implemented, is taken care of by the agent framework. However, this conversion will only work correctly if you follow the instructions on how to use the `AgentParameters` type (once again, see `U2Agent.h`).

If the invoked agent has `in/out` or `out` parameters, the values for these parameters shall be obtained after the call to `InvokeAgent` by iterating over the agent parameter list and using one of the `Get`-functions that are available on the `u2::AgentParameter` type.

**Example 726**

Assuming that the second parameter (boolean) of the agent invoked in Example 725 on page 2257 is an in/out parameter, this is how the resulting value for this parameter is obtained after the call to `InvokeAgent`:

```
if (params.size() == 2) // An agent should not (but could) add or
remove parameters to the agent parameter list
{
  try
  {
```

```
      bool result = params.back()->GetBoolean();
    }
    catch (u2::AgentParameter::ETypeMismatch)
    {
      // An agent should not (but could) change the type of an agent
  parameter.
      // Add appropriate error handling here!
    }
  }
```

The COM API documentation contains a more detailed description of InvokeAgent.

# ITtdEntity

The ITtdEntity interface is implemented by the Entity class of the Metamodel, which represents a general entity of a UML model.

The ITtdEntity interface contains methods that need a specific model entity context for their execution. Typically these methods operate on the entity on which they are called.

### ApplyStereotype

```
virtual u2::ITtdEntity*
ApplyStereotype(u2::ITtdEntity* pStereotype,
  u2::TtdReferenceKind referenceKind =
TTD_RK_MINIMAL_QUALIFIER,
  u2::ITtdEntity* pInsertElement = 0,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger)
throw(u2::APIError) =0;
```

Instantiates the given stereotype and applies it on the entity (referred to as the host entity). The qualifier in the reference to the stereotype is calculated based on the `referenceKind`, see table below for a description of possible values for `u2::TtdReferenceKind`. If `pInsertElement` is given the stereotype is logically instantiated on the host entity, but physically placed on the `pInsertElement`. In that case the stereotype instance will point to the host entity. In this way, stereotype instances may be “applied” to an entity without modifying the entity itself. This technique is known as “stereotype injection”.

The model modification is performed using the changer.

If the application of the stereotype fails an APIError exception is thrown.

| u2::TtdReferenceKind | Description |
|---|---|
| TTD_RK_GUID | The reference will only contain a GUID reference. |
| TTD_RK_NO_QUALIFIER | The reference will not be qualified. That is, it will only contain the name of the stereotype. |
| TTD_RK_FULL_QUALIFIER | The reference will contain a full qualifier. |
| TTD_RK_MINIMAL_QUALIFIER | The reference will contain the minimum qualifier needed to reference the stereotype. This option may at most return the same qualifier as relativeQualifier. The presence of <<access>>/<<import>> dependencies may make it shorter. |
| TTD_RK_RELATIVE_QUALIFIER | The reference will be a relative qualifier to the stereotype. If there are no common upper scopes of the stereotype and the host entity, a full qualifier is calculated, otherwise a shorter qualifier starting from the nearest common scope is calculated. |

### GetValue

```
virtual void
GetValue(const tstring& strMetaFeature,
  tstring& strValue,
  u2dll::eposition index = E_POSITION_END) const
throw(u2::APIError) = 0;
```

Gets the value of the specified metafeature for the entity. If no metafeature has the specified name, an APIError exception will be thrown. The value is encoded as a string. Use it for metafeatures with single or multiple multiplicity. In the case of multiple multiplicity use the index to specify which value to get.

The GetValue function can be used on all metafeatures of an entity that can have their values encoded as a string. This is the case for all metafeatures except derived features of Metaclass type, owner links and composition links.

**Note**

`index` *is an index starting at 1, and* `E_POSITION_END` *(==0) always specifies the last entity.*

**Example 727**

Using `GetValue` to retrieve the name of a definition:

```
if (pEntity->IsKindOf(_T("Definition"))) {
  tstring name;
  pEntity->GetValue(_T("Name"), name);
}
```

The COM API documentation contains a more detailed description of GetValue.

### GetEntity

```
virtual u2::ITtdEntity*
GetEntity(const tstring& strMetaFeature,
  u2dll::eposition index = E_POSITION_END) const
throw(u2::APIError) = 0;
```

Gets the value of the specified metafeature as an `ITtdEntity` pointer. Use it for metafeatures of Metaclass type that have either single or multiple multiplicity. In the case of multiple multiplicity use the index to specify which entity to get. If no metafeature has the specified name, an APIError exception will be thrown.

**Note**

*If the metafeature is unbound,* `pValue` *will be NULL. You may then want to use the GetValue function to get the value as a string representation instead, or GetReference to obtain the model representation of the unbound reference.*

**Note**

`index` *is an index starting at 1, and* `E_POSITION_END` *(==0) always specifies the last entity.*

The COM API documentation contains a more detailed description of GetEntity.

### GetEntities

```
virtual void
GetEntities(const tstring& strMetaFeature,
  std::list<u2::ITtdEntity*>& listEntities) const
```

```
throw(u2::APIError) = 0;
```

Gets the value of the specified metafeature as a list of entities. Use it for metafeatures of Metaclass type that have either single or multiple multiplicity. In the case of single multiplicity the result list will contain one or zero entities. If no metafeature has the specified name, an APIError exception will be thrown.

The COM API documentation contains a more detailed description of GetEntities.

### GetReference

```
virtual u2::ITtdEntity*
GetReference(const tstring& strMetaFeature,
  u2dll::eposition index = E_POSITION_END) const
throw(u2::APIError) = 0;
```

Returns the identifier representation of a metafeature reference. In the case of a metafeature with multiple multiplicity use the index to specify which reference to get. If no metafeature has the specified name, an APIError exception will be thrown.

**Note**

`index` *is an index starting at 1, and* `E_POSITION_END` *(==0) always specifies the last entity.*

The COM API documentation contains a more detailed description of GetReference.

### GetOwner

```
virtual u2::ITtdEntity*
GetOwner)() const = 0;
```

Returns the composition owner of the entity. If the entity has no owner NULL is returned.

The COM API documentation contains a more detailed description of GetOwner.

### GetMetaClassName

```
virtual void
GetMetaClassName(tstring& strMetaClassName) const = 0;
```

Finds the name of the entity's Metaclass and puts it in `strMetaClassName`.

The COM API documentation contains a more detailed description of GetMetaClassName.

### GetReferingEntities

```
virtual void
GetReferingEntities(const tstring& strMetaFeature,
  std::list<u2::ITtdEntity*>& listReferee) const = 0;
```

Finds out what entities that refer to the entity through the specified metafeature. A pointer to these referring entities will be put in the result list.

`strMetaFeature` may be an empty string in order to find all refering entities regardless of through which metafeature they refer to the entity.

The COM API documentation contains a more detailed description of GetReferringEntities.

### GetTaggedValue

```
virtual u2::ITtdEntity*
GetTaggedValue(const tstring& strSelector,
  bool bIdentsAreGuids = false) const
throw(u2::APIError) = 0;
```

Returns the expression representing the tagged value selected by the selector pattern. The entity should be either an extendable element (in which case the tagged value is looked for among the applied stereotype instances of the extendable element) or an instance expression (in which case the tagged value is looked for in that particular instance only).

A selector pattern specifies the path from the entity to the tagged value using the textual UML syntax of an instance expression. The function will check that the pattern matches both the instance tree (by structure) and the corresponding signatures (by name or GUID). If `bIdentsAreGuids` is true, identifiers of the pattern will be interpreted as GUIDs. Otherwise they will be interpreted as names. If no tagged value is selected by the selector pattern, NULL is returned.

Some selector pattern examples:

```
"T1 (. .)"
```

will return the first instance of the applied T1 stereotype

```
"T1 (. x .)"
```

will return the tagged value of the attribute x in the T1 stereotype

```
"T1 (. a1 = T2 (. a2 .) .)"
```

will return the value of a2 in a T2 instance being the value of T1.a1.

**Note**

*Although a selector pattern on the form "`X (. .)`" can be used to test if a stereotype X is applied on an entity, it is better to use HasAppliedStereotype for this purpose. The reason is that a stereotype that is automatically applied (due to a non-optional extension on a matching metaclass) will not be instantiated until at least one of its attributes get a tagged value that differs from the default value of the attribute.GetTaggedValue thus has no stereotype instance to return for that particular case. However, when used with a selector pattern that selects a tagged value, GetTaggedValue will also consider such automatically applied stereotypes.*

The COM API documentation contains a more detailed description of GetTaggedValue.

### HasAppliedStereotype

```
virtual bool
HasAppliedStereotype(const tstring& strStereotype,
  bool bGuid = false) const = 0;
```

Determines if the entity has a certain stereotype applied. The stereotype can be specified either by name or by guid. In the latter case `bGuid` should be set to true. This is the recommended function for checking for applied stereotypes on an entity. It will consider both explicitly applied stereotypes, and stereotypes that are automatically applied due to non-optional extensions from a metaclass that matches the metaclass of the entity.

### IsKindOf

```
virtual bool
IsKindOf(const tstring& strMetaClass) const = 0;
```

Returns true if the entity is of the specified Metaclass, false otherwise.

See Example 727 on page 2260 for an example of how to use `IsKindOf`.

The COM API documentation contains a more detailed description of IsKindOf.

### Unparse

```
virtual void
Unparse(tstring& strText) const
throw(u2::APIError) = 0;
```

Un-parses the entity into `strText` using textual UML syntax. The following kinds of entities can be unparsed: Definitions, Actions, Expressions. An attempt to unparse another kind of entity will yield an APIError.

The COM API documentation contains a more detailed description of Unparse.

### SetValue

```
virtual void
SetValue(const tstring& strMetaFeature,
  const tstring& strValue,
  u2dll::eposition index = E_POSITION_END,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger)
throw(u2::APIError) = 0;
```

Sets the value of a metafeature. The value is encoded as a string.

The `SetValue` function can be used on all writable metafeatures of an entity that can have their values encoded as a string. This is the case for all metafeatures except derived features (which are read-only), owner links and composition links.

If the metafeature has non-single multiplicity, the index argument can be used to insert the value before the value at the specified position.

If an error occurs (e.g. because of a non-existing metafeature) an APIError exception is thrown.

The COM API documentation contains a more detailed description of SetValue.

### SetEntity

```
virtual void
SetEntity(const tstring& strMetaFeature,
  u2::ITtdEntity* pValue,
  u2dll::eposition index = E_POSITION_END,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger)
throw(u2::APIError) = 0;
```

Sets the value of a metafeature. The value is an entity pointer. Use it for all writable metafeatures of Metaclass type. If the metafeature has non-single multiplicity, the index argument can be used to insert the entity before the value at the specified position.

**Note**

*If* `pValue` *is NULL, the metafeature will be made unbound. However this does not apply for owner links. To unset an owner link, i.e. to unlink an entity from its composition owner, use UnlinkFromOwner.*

If an error occurs (e.g. because of a non-existing metafeature) an APIError exception is thrown.

The COM API documentation contains a more detailed description of SetEntity.

### SetTaggedValue

```
virtual void
SetTaggedValue(const tstring& strSelector,
  const tstring& strValue,
  bool bOverwrite = true,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger)
throw(u2::APIError) = 0;
```

Sets the tagged value of the attribute selected by the selector pattern. See GetTaggedValue for the format of this pattern. The entity can either be an element with applied stereotypes or any instance expression. In the latter case the pattern is matched against the instance expression, while in the former case the first matching applied stereotype instance will be used.

By default an existing value for the selected attribute will be overwritten, but if `bOverwrite` is false this will not be the case.

**Note**

*In order to be able to overwrite an existing value for the specified attribute, the instance expression must be bound to the Signature of which it is an instance. If that is not the case, an APIError exception will be thrown.*

After the new value has been set all references in the entity is attempted to be bound, so that the set value can be accessed by GetTaggedValue directly after the call to this function.

The model modification is performed using the changer.

The COM API documentation contains a more detailed description of SetTaggedValue.

### Create

```
virtual u2::ITtdEntity*
Create(const tstring& strMetaClass,
```

```
    bool bBuildModelForPresentations = true,
    const tstring& strMetaFeature = _T(""),
    u2dll::Cu2Changer& changer = u2dll::defaultChanger)
throw(u2::APIError) = 0;
```

Creates an entity of the specified metaclass as a child of this entity. In the IDE execution environment this function is identical to Create of the COM API. Otherwise it will just attempt to create an instance of the specified metaclass and insert it as an immediate child of the entity in the specified metafeature. Model elements for presentation elements will not be automatically created in that case.

The metafeature can be left unspecified (an empty string) as long as the entity only contains one metafeature that can contain the created entity.

The model modification is performed using the changer.

The COM API documentation contains a more detailed description of Create.

### CreateInstance

```
virtual u2::ITtdEntity*
CreateInstance() const
throw(u2::APIError) = 0;
```

Call this function on entities that are signatures that can be instantiated, to create an instance of the signature. A common use for this function is to create an instance of a stereotype in order to apply the stereotype on an element. In case the creation fails an APIError exception will be thrown.

The COM API documentation contains a more detailed description of CreateInstance.

### Delete

```
virtual void
Delete(u2dll::Cu2Changer& changer =
u2dll::defaultChanger) = 0;
```

Deletes the entity. The model modification is performed using the changer.

The COM API documentation contains a more detailed description of Delete.

### XMLEncode

```
virtual void
XMLEncode(tstring& strXMLEncoding) const
```

```
throw(u2::APIError) = 0;
```

Puts the XML encoding of the entity in `strXMLEncoding`. If an error occurs, an APIError will be thrown.

The COM API documentation contains a more detailed description of XMLEncode.

### MetaVisit

```
virtual void
MetaVisit(u2::ITtdMetaVisitCallback* pCallback,
  bool bVisitAll = false,
  bool bVisitRefs = false) const
throw(u2::APIError) = 0;
```

Performs a Metamodel driven traversal of the model rooted at the entity. If `visitAll` is false libraries and the predefined package will be excluded from the traversal. If `visitRefs` is false, the identifier representations of references will be excluded from the traversal.

For each visited entity, the OnVisitedEntity function will be called on the `pCallback` object. This function is called for an entity when the model traversal reaches that entity, but before its contained entities have been visited. When all contained entities have been visited, the OnAfterVisitedEntity function will be called on the `pCallback` object. This allows actions to be performed on the “back recursion” of the traversal.

**Example 728**

This example shows how the `MetaVisit` function can be used in order to print the name of all definitions in a model:

```
class DefinitionFinder : public u2::ITtdMetaVisitCallback {
public:
  virtual bool OnVisitedEntity(u2::ITtdEntity* pEntity) {
    if (pEntity->IsKindOf(_T("Definition"))) {
      tstring name;
      pEntity->GetValue(_T("Name"), name);
      std::wcout << name << std::endl;
    }
    return true;
  }

  virtual void OnAfterVisitedEntity(u2::ITtdEntity* pEntity) {}

};

void foo() {
  u2::ITtdEntity* pEntity = u2::cast<u2::ITtdEntity>(pModel);
  DefinitionFinder finder;
  pEntity->MetaVisit(&finder);
```

```
}
```

The method in the COM API that corresponds to the MetaVisit function is called MetaVisitEx, whereas the COM method MetaVisit is a slightly more limited version of the same functionality.

### Bind

```
virtual void
Bind(const tstring& strMetaFeature = _T(""))
throw(u2::APIError) = 0;
```

Attempts to bind the specified metafeature on the entity (or all metafeatures if `strMetaFeature` is empty). If a non-existing metafeature is specified, an APIError exception will be thrown.

The COM API documentation contains a more detailed description of Bind.

### Locate

```
virtual void
Locate() const
throw(u2::APIError) = 0;
```

Locates the entity in the ModelView and/or diagrams. The effect in the IDE will be that the entity is shown in the model view (if possible) and in the diagrams (if a presentation for the entity exists in a diagram, that is). If this function is used when the IDE is not available, an APIError exception will be thrown.

### Clone

```
virtual u2::ITtdEntity*
Clone(bool bPreserveBindings = false,
  bool bPreserveGuids = false) const
throw(u2::APIError) = 0;
```

Creates a clone of the entity. By default the clone will be unbound and have new unique GUIDs (i.e. the copy of the entity itself and the copy of all contained entities will get new unique GUIDs). The optional arguments can be used to create a clone that has the same bindings as the original entity, and/or has the same GUIDs as the original entity.

**Important!**

*Be careful when cloning an entity without changing GUIDs. Such a clone should not be inserted into the same model as the original entity, or GUID conflicts will arise. If a model with GUID conflicts is saved, it might not be possible to load again.*

In order to be able to preserve bindings of the clone, the original entity must belong to a model (i.e. GetModel called on the original entity must not return NULL).

If the cloning fails for one reason or another an APIError exception will be thrown.

### Move

```
virtual void
Move(u2::ITtdEntity* pNewOwner,
  const tstring& strMetaFeature = _T(""),
  eposition index = E_POSITION_END,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger)
throw(u2::APIError) = 0;
```

Moves the entity from its current location in the model into the context of `pNewOwner`. If the entity would fit in more than one metafeature of the new owner, `strMetaFeature` must be specified to disambiguate. The `index` argument may be specified to control the position where to move the entity when the target metafeature has non-single multiplicity.

The model modification is performed using the changer.

If the move fails an APIError exception will be thrown.

### GetModel

```
virtual u2::ITtdModel*
GetModel() const = 0;
```

Returns the model to which the entity belongs. If the entity does not belong to a model NULL is returned. This function is often the most convenient way to get an ITtdModel interface from the context of an ITtdEntity interface.

### UnlinkFromOwner

```
virtual void
UnlinkFromOwner(u2dll::Cu2Changer& changer =
u2dll::defaultChanger) const = 0;
```

Unlinks the entity from its current owner. The entity will not be deleted, and can for example be inserted in another place in the model, or in another model.

The model modification is performed using the changer.

### Replace

```
virtual void
Replace(u2::ITtdEntity* pReplacementEntity,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger)
const
throw(u2::APIError) = 0;
```

Replaces the entity with another entity, without deleting the original entity. If the replacement is not possible to perform, an APIError exception will be thrown.

The model modification is performed using the changer.

**Note**

*If the entity is an identifier representing a reference it will be replaced with a clone of* `pReplacementEntity`*, rather than* `pReplacementEntity` *itself.*

### GetContainerMetaFeature

```
virtual void
GetContainerMetaFeature(tstring& strMetaFeature,
  u2dll::eposition& index) const = 0;
```

Sets `strMetaFeature` to the name of the metafeature in which the entity is contained. If the entity is orphan an empty string is used. `index` will be setup to the position within the metafeature where the entity is located.

### FindByName

```
virtual u2::ITtdEntity*
FindByName(const tstring& strName) const = 0;
```

Finds an entity by a name (possibly qualified) from the context of the entity. `strName` should be a valid identifier.

If no entity is found, NULL is returned.

### GetDescriptiveName

```
virtual void
GetDescriptiveName(tstring& strDescription) const = 0;
```

Sets `strDescription` to a descriptive name of the entity. The description includes the Metaclass of the entity, its name (full signature for event classes) and its location in the model.

## ITtdResource

The ITtdResource interface is implemented by the Resource class of the Metamodel, which represents a resource where a UML model could be persistently stored. Typically a Resource corresponds to a `.u2` file.

**Note**

*Since a Resource also is an Entity, a cast from ITtdResource to ITtdEntity will always succeed.*

### Save

```
virtual void
Save() const
throw(u2::APIError) = 0;
```

Saves the model entities that are associated with the resource on which the method is called. For the common case when the resource represents a `.u2` file, this means that the file will be saved.

The COM API documentation contains a more detailed description of Save.

## ITtdPresentationElement

The ITtdPresentationElement interface is implemented by the PresentationElement class of the Metamodel. A presentation element is an element with a graphical appearance, for example a symbol, line or diagram.

**Note**

*Since a PresentationElement also is an Entity, a cast from ITtdPresentationElement to ITtdEntity will always succeed.*

### GenerateEMF

```
virtual void
GenerateEMF(const tstring& strFileName,
  long maxWidth = 0,
  long maxHeight = 0,
  bool bOptimizeForVectorGraphics = false,
  bool bIncludeFrame = false) const
throw(u2::APIError) = 0;
```

Generates an EMF file (Enhanced Meta File) for the graphical appearance of a presentation element. This is a deprecated function. Use GenerateEMFEx instead.The presentation element will have the same appearance in this EMF file as when shown in the Tau editors.

The COM API documentation contains a more detailed description of GenerateEMF.

### GenerateEMFEx

```
virtual void
GenerateEMFEx(const tstring& strFileName,
  long maxWidth = 0,
  long maxHeight = 0,
  bool includeFrame = false)
  long scaleFactor = 0) const
throw(u2::APIError) = 0;
```

Generates an EMF file (Enhanced Meta File) for the graphical appearance of a presentation element. The presentation element will have the same appearance in this EMF file as when shown in the tool's editors.

**Note**

*GenerateEMF and GenerateEMFEx are only available in the interactive execution environment. An attempt to call these functions from another execution environment will yield an APIError exception.*

The COM API documentation contains a more detailed description of GenerateEMFEx.

### GenerateImage

```
virtual void
GenerateImage(ImageKind nKind,
  const tstring& strFileName) const
throw(u2::APIError) = 0;
```

Generates an image file from the graphical appearance of a presentation element. The presentation element will have the same appearance in this image file as when shown in the tool's editors.

Valid values in the `ImageKind` enumeration, and their meaning, are listed in the table below:

| u2::ImageKind | Description |
|---|---|
| IK_JPEG | Generate a JPEG image file. |
| IK_BMP | Generate a BMP image file. |
| IK_GIF | Generate a GIF image file. |
| IK_TIFF | Generate a TIFF image file. |
| IK_TARGA | Generate a TGA (Targa) image file. |
| IK_DIB | Generate a device independent bitmap file. |
| IK_PCX | Generate a PCX image file. |

## ITtdSymbol

The ITtdSymbol interface is implemented by the Symbol class of the Meta-model. A symbol is a presentation element with a two-dimensional graphical appearance.

**Note**

*Since a Symbol also is a PresentationElement and an Entity, a cast from ITtdSymbol to ITtdPresentationElement or ITtdEntity will always succeed.*

### SetSize

```
virtual void
SetSize(unsigned long width,
  unsigned long height,
  u2dll::Cu2Changer& changer = u2dll::defaultChanger) =
0;
```

Sets the size of the symbol. The unit of the width and height is 1/10:th of a millimeter. In a non-interactive execution environment SetSize will just update the value of the ‘size’ metafeature for the symbol. In the interactive execution environment, however, SetSize can also perform additional model changes related to the resize. This could for example happen if the symbol size has a semantic significance. It is therefore recommended to always use SetSize in order to set the size of a symbol.

The model modification is performed using the changer.

### SetPosition

```
virtual void
SetPosition(long x,
  long y,
u2dll::Cu2Changer& changer = u2dll::defaultChanger) = 0;
```

Sets the position of the symbol. The unit of the x and y parameters is 1/10:th of a millimeter. In a non-interactive execution environment SetPosition will just update the value of the 'position' metafeature for the symbol. In the interactive execution environment, however, SetPosition can also perform additional model changes related to the repositioning. This could for example happen if the symbol position has a semantic significance. It is therefore recommended to always use SetPosition in order to set the position of a symbol.

The model modification is performed using the changer.

## ITtdExpression

The ITtdExpression interface is implemented by the Expression class of the Metamodel. It represents an expression in the model.

**Note**

*Since an Expression also is an Entity, a cast from ITtdExpression to ITtdEntity will always succeed.*

### GetType

```
virtual u2::ITtdEntity*
GetType() const = 0;
```

Computes and returns the type of the expression. If the type cannot be computed (for example because the expression contains unbound references) NULL is returned.

**Note**

*The returned entity is not guaranteed to be part of the model. In some cases the type is not explicitly defined in the model, and in that case a temporary entity which represents the type will be returned. Be careful not to delete such a temporary entity.*

### EvaluateConstantIntegralExpression

```
virtual long
EvaluateConstantIntegralExpression() const
throw (u2::APIError) = 0;
```

Evaluates the value of the expression, which is expected to be a constant integral expression. If it is not, or the evaluation fails for some other reason, an APIError exception will be thrown.

### GetInstanceChildExpression

```
virtual u2::ITtdExpression*
GetInstanceChildExpression(const tstring& strName) const
throw(u2::APIError) = 0;
```

Use this function on an instance expression (for example a stereotype instance) to obtain the right-hand side of a contained assignment, where the left-hand side of the assignment is an identifier matching `strName` (which thus should be a valid identifier).

If no matching child expression is found, NULL is returned.

In case of an erroneous usage an APIError exception will be thrown.

## ITtdMetaVisitCallback

The ITtdMetaVisitCallback interface is a callback interface that clients using the MetaVisit function must implement.

### OnVisitedEntity

```
virtual bool
OnVisitedEntity(u2::ITtdEntity* pEntity) = 0;
```

Called when using MetaVisit to traverse a model. `pEntity` is the currently visited entity in the model. This function is called before visiting the composition children of `pEntity`. If false is returned traversal will not continue with the composition children of `pEntity`.

### OnAfterVisitedEntity

```
virtual void
OnAfterVisitedEntity(u2::ITtdEntity* pEntity) = 0;
```

Called when using MetaVisit to traverse a model. `pEntity` is the currently visited entity in the model. This function is called after the composition children of `pEntity` have been visited, and can thus be used in order to perform actions on the “back recursion” of the model traversal.

# ITtdSourceBuffer

The ITtdSourceBuffer interface represents a buffer of text (typically source code) used primarily during code generation as a representation of a generated file. It can for example be used by agents of the C++ Application Generator that wish to add some additional text to a generated file.

### AddText

```
virtual void
AddText(const tstring& strText)
throw(u2::APIError) = 0;
```

Writes the specified text to the source buffer

The COM API documentation contains a more detailed description of AddText.

# ITtdMessageList

The ITtdMessageList interface represents a list of messages. Such a list is for example used when reporting errors from semantic analysis or code generation, and the interface may be used by agents in order to add custom messages based on for example a custom semantic check.

### AddMessage

```
virtual void
AddMessage(const tstring& strMessage,
  MessageSeverity severity,
  u2::ITtdEntity* pSubject = 0)
throw(u2::APIError) = 0;
```

Adds a new message to the list with the specified severity and, optionally, a subject entity. The subject entity will be associated with the message. In an interactive execution environment the subject entity can be located from the message.

The `MessageSeverity` enumeration is defined like this:

```
enum MessageSeverity {
  MS_INFORMATION,
  MS_WARNING,
  MS_ERROR,
  MS_FATAL
};
```

The COM API documentation contains a more detailed description of AddMessage.

### GetDescription

```
virtual void
GetDescription(tstring& strErrorMessage,
  const tstring& strSeparator) const = 0;
```

Writes to the `strErrorMessage` string a description of all messages in the message list. The messages are separated with the specified separator string.

### GetCount

```
virtual unsigned int
GetCount(MessageSeverity severity) const = 0;
```

Returns the number of messages with a severity that is equal to `severity`.

## ITtdInteractiveServer

The ITtdInteractiveServer interface represents the Tau IDE as seen as a server for an interactive API client. It defines the interface through which an interactive client communicates with Tau during its execution.

### InterpretTclScript

```
virtual void
InterpretTclScript(const tstring& strScript,
  std::list<u2::ITtdEntity*>& entities,
  tstring& strResult)
throw(u2::APIError) = 0;
```

Interprets a Tcl script on the server, and returns the result to the caller. This function is a means for an interactive client to communicate with the IDE. The entire Tcl API is available.

The COM API documentation contains a more detailed description of InterpretTclScript.

## ITtdCppAppGenServer

The ITtdCppAppGenServer interface represents the Tau C++ Application Generator as seen as a server for a non-interactive API client. It defines the interface through which an agent communicates with the code generator during its execution. The agent obtains this interface as the ‘server’ argument when the agent gets invoked.

### ScheduleForDeletion

```
virtual void
ScheduleForDeletion(u2::ITtdEntity* pEntity)
throw(APIError) = 0;
```

Unlinks the entity immediately from the model, and schedules it for deletion. The entity will be deleted when considered appropriate by the C++ Application Generator.

The COM API documentation contains a more detailed description of ScheduleForDeletion.

# C++ API Set-up

This chapter attempts to guide you through the process of setting up a client application for using the C++ API. As this process to a large extent is platform dependent, a separate description is provided for Unix and Windows platforms.

## Windows clients

Perform the following steps for setting-up a Visual Studio project for using the Tau C++ API:

1. Add the `include\ToolAPI` directory of the Tau installation to the project’s list of include paths. If you do not do this, you will have to use appropriate relative paths in your include directives of the API headers.
2. Add a `#include "U2ModelAccess.h"` to the implementation files from which you want to access the API.
3. Set-up the project to link with the libraries `U2DLLU.lib` and `SBL10U.lib` which can be found in the Tau installation at `lib\win32-vc`.
4. Set-up the project to use the “Multi-threaded DLL” run-time library.

5. Make sure your client uses wide (Unicode) strings. If you want to use single-byte strings in your application you must perform the necessary conversions, since the API will expect wide character (Unicode) strings. To specify the use of Unicode strings, set the macros `UNICODE` and `_UNICODE`.
6. Make sure your `PATH` environment variable includes the Tau installation `bin` directory. This step is not necessary if you are building an interactive API client (e.g. an agent).

**See also**

"Debug C++ agents in Visual Studio" on page 2280 in Chapter 78, *C++ API*

## Unix clients

Perform the following steps for creating a makefile for a Unix application that shall use the Tau C++ API:

1. Add the `include/ToolAPI` directory of the Tau installation to the list of preprocessor include paths. If you do not do this, you will have to use appropriate relative paths in your include directives of the API headers.
2. Add a `#include "U2ModelAccess.h"` to the implementation files from which you want to access the API.
3. Link with the libraries `u2dll.lib` and `sbl10.lib` which can be found in the Tau installation in the `lib` directory. There is one subfolder with a library built for each supported compiler and platform. Make sure to choose the correct one.
4. Compile with CC on Solaris and g++ on Linux. Specify the use of a multi-threaded run-time library using the flag `-mt` on Solaris or `-D_REENTRANT` on Linux.
5. Make sure your client uses narrow (ASCII) strings. If you want to use multi-byte strings in your application you must perform the necessary conversions, since the API will expect narrow character (ASCII) strings.
6. Make sure your `LD_LIBRARY_PATH` environment variable includes the Tau installation `bin` directory. This step is not necessary if you are building an interactive API client (e.g. an agent).

# Debug C++ agents in Visual Studio

This article describes how to successfully debug C++ agents, or in general any client of the Tau C++ API, using Visual Studio 2005.

## Setting up an appropriate debug configuration

The first thing to note is that by default a program built in Debug configuration with Visual Studio will use a debug version of the C run-time library. For an agent DLL, which should use the multi threaded DLL version of the run-time library, this means that the MSVCR8D library will be used. However, since all Tau binaries are delivered as built in Release configuration, they will use the non-debug version of this library (called MSVCR8). Thus, if you attempt to run a Debug-built agent inside a Telelogic Tau binary (such as VCS.EXE) you will most likely run into problems caused by having both the Debug and Release versions of the run-time library in memory at the same time. Memory that is allocated in the agent DLL and de-allocated in Tau, or vice versa, will yield debug assertions and could also crash the application.

The solution to this problem is to always build an agent DLL in the Release configuration. This ensures consistency in the use of the run-time library. However, since Debug information is by default turned off in a Release configuration, you have to turn on Debug information manually in order to be able to debug such an agent. To do that, follow the steps below:

- Start from a Release configuration, either by modifying the default Release configuration, or by creating a new configuration based on the settings in the default Release configuration. The latter is done using the Build -> Configuration Manager command in Visual Studio.
- Open the project property pages and select the C/C++ folder (category General). Set “Debug Information Format” to “Program Database”. Then disable all optimization (in category Optimization).
- Then select the Linker folder (category Debugging). Set the option “Generate Debug Info” to Yes.
- Finally, in the Debugging page set the appropriate Tau binary (for example VCS.EXE) from the Tau installation as the executable (command) for the debug session. Also specify the Tau installation directory as working directory.

- Now you may build the agent DLL, set breakpoints in its implementation, and start a debug session. When Telelogic Tau is started and the agent gets invoked, the breakpoint will be reached and debugging can be done.

## Debugging utilities

There are a few built-in utility functions in the Tau libraries that can be very useful when debugging a C++ agent, or a client of the C++ API in general. These utility functions are not declared in any of the API header files that are part of the Tau installation, but by declaring their function prototypes in the agent implementation file, they can be accessed. They can also be called from within the debugger.

All debug utility functions are defined as exported from the DLL in which they are implemented. For information on how to call these functions from the Visual Studio debugger, see U2ViewModel below. The principle is the same for the other functions.

The table below summarizes the debug utilities that are available, which Tau DLL they are located in, and the namespace in which they are defined:

| Function | Namespace | DLL |
|---|---|---|
| U2ViewModel | u2dll | u2dllu.dll |
| DbgInterpretTclScript | u2ext | u2extu.dll |

Note that the U2DLL DLL is available both for interactive and non-interactive agents, while the U2EXT DLL only is available for interactive agents.

### U2ViewModel

```
namespace u2dll {
  void U2ViewModel(const void* pEntity);
}
```

This function dumps the model rooted at pEntity into a temporary XML file. It is useful for examining the contents of any model fragment. If Internet Explorer (iexplore.exe) is in the path, it will also be launched to show the XML file. Otherwise the generated file can be opened from where it is generated in the user's directory for temporary files. `pEntity` should be the address of an object realizing the `u2::ITtdEntity` interface.

To call this function from the debugger, open the Command Window (in "immediate" mode) or the Quick Watch window and make the call to `U2ViewModel` as shown below. Note that the argument must be specified as an address (as displayed in the Watch window):

```
{,,u2dllu} u2dll::U2ViewModel((void*) 0x12ea4d24)
```

To call the function from the agent implementation, put this declaration in the beginning of the agent implementation file:

```
namespace u2dll {
    __declspec(dllimport)
    void U2ViewModel(const void* pEntity);
}
```

Now you may call U2ViewModel on any ITtdEntity pointer in your agent implementation.

**Hint**

*The Visual Studio debugger sometimes refuses to call functions that are located in DLL:s for which no debug symbols can be found. If you encounter this problem, here is a workaround:*
*Add a declaration of the function you want to call in your agent implementation file as shown for U2ViewModel above. Then create a local wrapper function which calls that function:*

```
void U2ViewModelWrapper(const void* pEntity) {
  u2dll::U2ViewModel(pEntity);
}
```

*Rebuild the agent. Now you may call U2ViewModelWrapper directly from the debugger:*

```
U2ViewModelWrapper((void*) <address>)
```

### DbgInterpretTclScript

```
namespace u2ext {
  char* DbgInterpretTclScript(char* arg);
}
```

This function gives access to the entire Tau Tcl API, which can be used for debugging the agent. In particular it is useful for navigating and exploring the model in the debugged application. The input argument of the function should be a Tcl script, and the return value will be the result of interpreting the script with the Tau Tcl interpreter.

The Tcl script string may refer to model entities in the debugged application. To form such a reference enclose the address of the entity as displayed by the Visual Studio debugger within # marks.

**Example 729: Using DbgInterpretTclScript from the Visual Studio debugger**

Open the Command Windows of Visual Studio and enter "immediate" mode (type the command 'immed').

To find the owner of an entity with address 0x0f958720:

```
{,,u2extu} u2ext::DbgInterpretTclScript("u2::GetOwner
#0x0f958720#")
```

A result on the following form will be printed:

```
0x0d5bcd58 "i.66.f891eb0"
```

The first address in the result is uninteresting, it is just the internal address of the string. The quoted string is the result of executing the Tcl script. When the result is a Tcl object Id, as in this case, it can easily be converted to an address of the debugged program by simply stripping the prefix from the address. Hence, the Tcl object id in this example corresponds to the address 0x0f891eb0. This address can then be used as input to other calls of debug utility functions.

---

Since the Tcl API is implemented in the interactive Tau IDE this function can only be accessed from interactive agents.

# 79

# *Java API*

This chapter is the reference documentation of the Tau Java API, which enables access of a Tau model from a Java program.

Intended readers are developers of client applications that use the Java API to access a UML model. A basic knowledge of Java is assumed throughout this chapter.

# Introduction

The Java API consists of a set of interfaces, each with a number of methods. The design of the API is very similar to the design of the C++ API. In fact the Java API is implemented in terms of the C++ API using the Java Native Interface (JNI).

A client of the Java API is a Java program which executes inside a Java Virtual Machine (JVM). This means that it is only possible to build non-interactive clients using the Java API. An interactive client, such as an agent, cannot be implemented in Java.

### Java version

To use the Java API the client program needs to run inside a JVM for Java 5 or later.

# Accessing the API

A client that wants to use the Java API should place the `tauaccess.jar` in its classpath. This JAR file can be found in the Tau installation at `/lib/Java`. The client must also set-up the environment variable `PATH` (`LD_LIBRARY_PATH` on Unix) so that it includes the Tau installation `bin` directory.

## Execution Environments

Usually a Java API client loads or creates a UML model inside the JVM where it executes. The client is said to execute in a non-interactive environment because the Tau IDE is not involved.

The starting point for accessing the API from a non-interactive client is the method `TauModelAPI.getModelAccess()`. This method returns a reference to a singleton object that implements the ITtdModelAccess interface.

However, by using Tau Access it is also possible to run a Java API client in an interactive execution environment where the implementation of the Java API operates on a running instance of Tau, where the IDE is available.

The starting point for accessing the API from an interactive client is through interfaces provided by the Tau Access API. See Tau Access for more information.

In most cases there is no functional difference of the Java API when used in an interactive vs. non-interactive execution environment. However, the following differences apply:

- API methods that modify the model will be undoable in an interactive execution environment (i.e. Undo/Redo can be used to undo/redo the actions performed by the API methods). In a non-interactive execution environment model modifications are not undoable.
- Some API methods have parameters that are specific for a particular execution environment. The value of such parameters in the other execution environment will be ignored.
- Some API methods can only be used in a particular execution environment - typically because their implementation rely on features of the Tau IDE.
- Some API methods work slightly differently in different execution environments. Typically they may be more complete in their support in an interactive execution environment since features of the Tau IDE then are available.

## API Initialization and Finalization

It is important to initialize the API before starting to use it. This is done by calling `TauModelAPI.initializeModelAccess()`.
After the last API call the API should also be finalized, by a call to `TauModelAPI.finalizeModelAccess()`. It is allowed to repeat initialization and finalization as long as the calls are balanced.

**Example 730 API initialization and finalization**

Initializing the Java API, obtaining a reference to the `ITtdModelAccess` interface, and finalizing the API afterwards:

```
TauModelAPI.intializeModelAccess();
ITtdModelAccess ma = TauModelAPI.getModelAccess();
TauModelAPI.finalizeModelAccess();
```

## Interface Casting

One interface can be casted to another interface dynamically using the usual Java cast operator. If the cast is successful, meaning that the object which implements the source interface also implements the target interface, the result

of the cast is a reference typed by the target interface. However, if the cast fails a `java.lang.ClassCastException` will be thrown. Make sure to catch this exception unless you are certain that the object at hand implements the target interface.

**Example 731 Interface casting**

Dynamic casting between interfaces using the Java cast operator.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.loadProject("x.u2", true, new MessageList());
ITtdEntity e = (ITtdEntity) model; // A model is an entity
try
{
  ITtdSymbol s = (ITtdSymbol) model.findByGuid("abc123");
}
catch (ClassCastException err)
{
  // Error handling here
}
```

The cast from `ITtdModel` to `ITtdEntity` will always succeed (because the model node is also an entity). Therefore we don't need to catch any exception from that cast. However, the cast from `ITtdEntity` to `ITtdSymbol` will not always succeed (not all entities are symbols). Hence we should be prepared for a `ClassCastException` in this case.

---

It is also possible to use the Java `instanceof` operator to test whether an object implements a certain interface.

# Handling API Errors

The Java API reports all errors by throwing an `APIError` exception. Client code should be prepared for errors by catching this exception. The exception class is defined like this:

```
package com.telelogic.tau;

// Common exception class describing an error occurring
while using the Tau APIs.
public class APIError extends Throwable
{
  public APIError(String msg)
  {
    super(msg);
  }
}
```

All methods inherited from the `Throwable` class can be used on `APIError`. In particular the textual description of the error is obtained by calling `getMessage()`.

**Example 732 Catching APIError exceptions**

A typical catch clause for handling Java API errors could look like this:

```
try {
  // access the Java API
}
catch (APIError err)
{
  String s = err.getMessage();
  System.out.println("Error while using the Tau Java API: " + s);
}
```

# Memory Management

Java programmers tend not to think much about memory management due to the presence of garbage collection in Java. However, it's important to understand that the Java garbage collector will only manage Java objects. Objects that are created in the Tau object model are not Java objects, although they implement a Java interface. Therefore such objects should be manually deleted when they are no longer used, to avoid memory leaks. Use `ITtdEntity::`delete to delete an entity from the model.

In practise, however, a UML model that is created or loaded through the Java API often lives throughout the entire lifetime of the Java application, and hence memory management becomes no concern.

# Client restrictions

The Java API imposes few restrictions on its clients. However, please keep in mind the following when designing the API client:

### Bare only

The API does not support safe simultaneous access of the model from multiple threads.

# API Interfaces and Methods

All interfaces of the Java API are placed in the package `com.telelogic.tau`. This section lists all API interfaces and methods and gives a short description of each. Some examples of typical use are provided. Note, however, that the examples only serve as an illustration of how to use a particular API method or interface, and are not always complete. In particular the recommended error handling (see Handling API Errors) is usually omitted from the examples for brevity reasons.

The names of the Java interfaces and their methods are the same as the corresponding interfaces and methods of the COM and C++ APIs. However, note that by convention Java method names start with a lowercase letter.

Many of the interfaces are implemented by classes representing metaclasses in the implementation of the UML Metamodel. Some of the methods in the Java API require knowledge of the metamodel in order to be useful.

## ITtdModelAccess

The `ITtdModelAccess` interface contains methods that do not operate directly on the model. Use it to get access to a UML model from a non-interactive API client by loading it from files (project file or `.u2` file), or create a new model from scratch.

See Accessing the API to learn how to obtain the `ITtdModelAccess` interface from the client application.

### loadProject

```
ITtdModel loadProject(String strProject, boolean bind,
ITtdMessageList messages) throws APIError;
```

Loads the project stored in the specified project file (or URI). If the project file does not exist, or some other error occurs while loading the project, an `APIError` exception will be thrown. If `strProject` is a relative path, it will be interpreted as relative to the current working directory of the client application.

If the `bind` parameter is set to true, the model will be bound after it has been loaded. Set it to false to suppress this binding.

If a message list is passed to the function all messages that are produced during loading and binding will be added to that list. A message list can be obtained by defining a class which implements the ITtdMessageList interface. In case load messages are not of interest you may pass `null` for this parameter.

**Example 733**

This example loads a model from a Tau project file, without binding the loaded model. All messages are ignored.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.loadProject("x.ttp", false, null);
```

This example loads a project and uses a message list for printing load and bind messages:

```
public class Test
{
  public static class MessageList implements ITtdMessageList
  {
    public void addMessage(String text, MessageSeverity severity,
ITtdEntity subject)
    {
      System.out.println(text);
      if (subject != null)
        System.out.println(subject.getMetaClassName());
    }
  }

  public static void main(String[] args) throws APIError
  {
    TauModelAPI.intializeModelAccess();
    ITtdModelAccess ma = TauModelAPI.getModelAccess();
    ITtdModel model = ma.loadProject("x.ttp", true, new
MessageList());
    TauModelAPI.finalizeModelAccess();
  }
}
```

### loadFile

```
ITtdModel loadFile(String strFile, boolean library)
throws APIError;
```

Loads the specified `.u2` file (or URI). If the file does not exist, or some error occurs while loading it, an `APIError` exception will be thrown. If `strFile` is a relative path, it will be interpreted as relative to the current working directory of the client application. If `library` is true, the file will be loaded as a library. In that case the file should contain a package, which then will be placed in the Libraries section of the model.

**Example 734**

This example loads a model from a Tau `.u2` file as a library. It then prints the name of all library packages in the model. The last package printed will be the package contained in the loaded `.u2` file.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.loadFile("x.u2", true);
ITtdEntity entityModel = (ITtdEntity) model;
List<ITtdEntity> lst = entityModel.getEntities("Library");
for (ITtdEntity e : lst)
{
  System.out.println(e.getValue("Name", 0));
}
```

Note that `ITtdModelAccess::loadFile()` loads the file into a new model. In order to load a `.u2` file into an existing model use `ITtdModel::loadFile()`.

### createModel

```
ITtdModel createModel() throws APIError;
```

Creates a new empty model. If a new model cannot be created (for example due to a memory or license problem), an `APIError` exception will be thrown.

The created empty model can be used as a starting-point for creating a whole new UML model. The example below shows the creation of a simple model containing one package with one class. It also shows how to save the new model afterwards.

**Example 735**

This example creates a new model, containing a package with a class, and saves it to a `.u2` file.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity modelRoot = (ITtdEntity) model;
ITtdEntity pkg = modelRoot.create("Package", false, "OwnedMember");
ITtdEntity cls = pkg.create("Class", false, "");
ITtdResource resource = model.createResource("C:\\temp\\test.u2");
// Insert the created package pkg as a root of the resource
((ITtdEntity) resource).setEntity("Root", pkg, 0);
model.save(); // Saves all resources in the model
```

### writeMessage

```
void writeMessage(String message);
```

Writes a message to the default message area (typically to `stdout` in a non-interactive execution environment, or to the Message tab in an interactive execution environment).

**Example 736**

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ma.writeMessage("Hello from Java!");
```

# ITtdModel

The `ITtdModel` interface is implemented by the Session class of the Metamodel, which represents the top-level entity of a UML model.

The `ITtdModel` interface contains methods that do not need a specific model entity context for their execution. Typically these methods operate on the model as a whole, rather than on a particular entity.

**Note**

*Since a Session also is an Entity, a cast from* `ITtdModel` *to* `ITtdEntity` *will always succeed.*

## findByGuid

```
ITtdEntity findByGuid(String strGuid);
```

Returns the entity in the model with the specified GUID, or `null` if no such entity exists.

**Example 737**

This example creates a new model, and then locates the predefined Integer datatype by its GUID.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity integerType = model.findByGuid("@Predefined@Integer");
ma.writeMessage("Integer is a " + integerType.getMetaClassName());
```

## New

```
ITtdEntity New(String strMetaClass) throws APIError;
```

Creates a new instance of the specified Metaclass. If a metaclass with the specified name does not exist, or the method fails for some other reason, an `APIError` exception is thrown.

**Note**

*It is the responsibility of the client to take care of the returned entity. It should either be inserted into the model, or deleted, to avoid a memory leak. See Memory Management for more information.*

The `New` method is intended to be used for low-level creation of object model entities, for example to create temporary model fragments which are not inserted into a model. Do not use New to build up a complete model - use `ITtdEntity::`create instead.

**Example 738**

This example creates a new temporary package and sets up its name. It then prints its textual (unparsed) representation. Finally the package is deleted.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity pkg = model.New("Package");
pkg.setValue("Name", "MyPackage", 0);
ma.writeMessage(pkg.unparse());
pkg.delete();
```

### parse

```
List<ITtdEntity> parse(String strConcreteSyntax, String
parseAs) throws APIError;
```

Parses the specified piece of concrete textual UML syntax. The parameter `parseAs` specifies the grammar to use when parsing, i.e. which kind of entities that should be defined in the text. Supported grammars are Definition, Expression and Action. In case a non-supported grammar is specified or the text contains syntax errors, an `APIError` exception will be thrown. The result built by the parser will be inserted into the returned list.

**Note**

*It is the responsibility of the client to take care of the returned entities. They should either be inserted into the model, or deleted, to avoid a memory leak. See Memory Management for more information.*

**Example 739**

This example creates two classes by parsing their textual definitions. The names of the classes are then printed. Finally the classes are deleted.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
List<ITtdEntity> lst = model.parse("class C1 {} class C2 {}",
"Definition");
for (ITtdEntity e : lst)
{
  System.out.println(e.getValue("Name", 0));
  e.delete();
}
```

### XMLDecode

```
List<ITtdEntity> XMLDecode(String strXMLEncoding) throws
APIError;
```

Decodes the U2 XML encoded string into a list of model entities. If the decoding fails (e.g. because of a syntax error in the XML) an `APIError` will be thrown.

## Note

*It is the responsibility of the client to take care of the returned entities. They should either be inserted into the model, or deleted, to avoid a memory leak. See Memory Management for more information.*

**Example 740**

This example creates a model with one class. It then encodes the class as an XML string using `ITtdEntity::`XMLEncode. Then it calls XMLDecode to obtain the class again, and prints its metaclass name. The new class is in effect a clone of the original class. Note that `ITtdEntity::`clone is an easier way to clone a model entity.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity pkg = ((ITtdEntity) model).create("Class", false, "");
String xml = pkg.XMLEncode();
List<ITtdEntity> lst = model.XMLDecode(xml);
for (ITtdEntity e : lst)
{
  System.out.println(e.getMetaClassName());
  e.delete();
}
```

### save

```
void save() throws APIError;
```

Saves the model into its resources. If the model does not contain any resources, nothing will happen.

If the model cannot be saved an `APIError` exception will be thrown.

See Example 735 on page 2292 for an example of using `save` to save a model.

### createResource

```
ITtdResource createResource(String strFileName) throws
APIError;
```

Creates a new resource for the model. The resource will be a file with the specified file name. If the creation fails an `APIError` will be thrown.

See Example 735 on page 2292 for an example of using `createResource` to create a new resource in a model.

### loadFile

```
ITtdResource loadFile(String strFileName, boolean
profile) throws APIError;
```

Loads the specified file into the model. A resource representing the file will be created and returned. If `profile` is true, the file will be loaded as a library. In cases of error (e.g. load errors) an `APIError` will be thrown.

**Example 741**

This example creates a model and loads an existing file into it. It then obtains the last resource from the model (the one created by `loadFile`), and prints its URI.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
model.loadFile("x.u2", false);
ITtdEntity resource = ((ITtdEntity) model).getEntity("Resource",
0);
String uri = resource.getValue("uri", 0);
System.out.println(uri);
```

### invokeAgent

```
void invokeAgent(ITtdEntity agent, ITtdEntity
modelContext, List<Object> agentParameters) throws
```

```
APIError;
```

Invokes the specified agent on the specified model context.

The `agentParameters` type is a list of Agent Parameters, representing actual arguments passed to the invoked agent. An agent parameter is in Java represented as an `Object` instance.

If the agent cannot be invoked, or the invoked agent signals an error, an `APIError` exception will be thrown.

**Example 742**

This example invokes the predefined query agent `GetGeneralizationChildren` in order to find all definitions that inherit from the predefined `Collection` type. The agent takes two parameters; a list of resulting definitions and a boolean specifying whether also indirect generalization children should be returned.

Note that the first parameter is an in/out parameter. See the note below for how to deal with such parameters.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity collection = model.findByGuid("@Predefined@Collection");
ITtdEntity query =
model.findByGuid("@TTDQuery@GetGeneralizationChildren");

LinkedList<Object> agentParameters = new LinkedList<Object>();
agentParameters.add(new LinkedList<Object>());
agentParameters.add(true);

model.invokeAgent(query, collection, agentParameters);

List<Object> lst = (List<Object>) agentParameters.element();

for (Object obj : lst)
{
  if (obj instanceof IUnknown)
  {
    ITtdEntity entity = (ITtdEntity) obj;
    System.out.println(entity.getValue("Name" ,0) + " is a
Collection!");
  }
}
```

**Note**

*If the invoked agent has* `in/out` *or* `out` *parameters, the values for these parameters shall be obtained after the call to* `InvokeAgent` *by iterating over the agent parameter list. After the agent invocation it is not safe to access a local reference to such a parameter that has been obtained prior to the agent invocation.*

The table below shows the mapping of agent parameter types to Java types:

| Agent parameter type (UML) | Java type |
|---|---|
| `Charstring` | `String` |
| `Integer` | `Integer` |
| `Boolean` | `Boolean` |
| `Reference to interface (for example u2::ITtdEntity)` | `com.telelogic.tau.IUnknown` |
| `u2::ITtdEntity [*]` | `List<ITtdEntity>` |

# ITtdEntity

The `ITtdEntity` interface is implemented by the Entity class of the Meta-model, which represents a general entity of a UML model.

The `ITtdEntity` interface contains methods that need a specific model entity context for their execution. Typically these methods operate on the entity on which they are called.

### applyStereotype

```
ITtdEntity applyStereotype(ITtdEntity stereotypeToApply,
TtdReferenceKind referenceKind, ITtdEntity insertElement)
throws APIError;
```

Instantiates the given stereotype and applies it on the entity (referred to as the host entity). The qualifier, if any, in the reference to the stereotype is calculated based on the `referenceKind`, see table below for a description of possible values. If `insertElement` is given the stereotype is logically instantiated on the host entity, but physically placed on the `insertElement`. In that case the stereotype instance will point to the host entity. In this way, stereotype instances may be “applied” to an entity without modifying the entity itself. This technique is known as “stereotype injection”.

If the application of the stereotype fails an `APIError` exception is thrown.

| TtdReferenceKind | Description |
|---|---|
| TTD_RK_GUID | The reference will only contain a GUID reference. |
| TTD_RK_NO_QUALIFIER | The reference will not be qualified. That is, it will only contain the name of the stereotype. |
| TTD_RK_FULL_QUALIFIER | The reference will contain a full qualifier. |
| TTD_RK_MINIMAL_QUALIFIER | The reference will contain the minimum qualifier needed to reference the stereotype. This option may at most return the same qualifier as TTD_RK_RELATIVE_QUALIFIER would. The presence of <<access>> or <<import>> dependencies may make it shorter. |
| TTD_RK_RELATIVE_QUALIFIER | The reference will be a relative qualifier to the stereotype. If there are no common upper scopes of the stereotype and the host entity, a full qualifier is calculated, otherwise a shorter qualifier starting from the nearest common scope is calculated. |

Usually TTD_RK_MINIMAL_QUALIFIER is the recommended value to use, because it gives the same behavior as when the stereotype is applied using the Tau user interface.

**Example 743**

This example creates a model with an actor inside. An actor is represented by an attribute stereotyped by the predefined <<actor>> stereotype. To verify that the operation was successful, the unparsed representation of the actor is printed.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity a = ((ITtdEntity) model).create("Attribute", false, "");
a.setValue("Name", "MyActor", 0);
ITtdEntity actorStereotype = model.findByGuid("@Predefined@actor");
a.applyStereotype(actorStereotype,
```

```
TtdReferenceKind.TTD_RK_MINIMAL_QUALIFIER, null);
System.out.println(a.unparse());
```

### getValue

```
String getValue(String strMetaFeature, int index) throws
APIError;
```

Gets the value of the specified metafeature for the entity. If no metafeature has the specified name, an `APIError` exception will be thrown. The value is encoded as a string. Use it for metafeatures with single or multiple multiplicity. In the case of multiple multiplicity use the `index` to specify which value to get.

The `getValue` method can be used on all metafeatures of an entity that can have their values encoded as a string. This is the case for all metafeatures except derived features of Metaclass type, owner links and composition links. If such a metafeature is specified an `APIError` exception will be thrown.

**Note**

`index` *is an index starting at 1, and 0 specifies the last entity in the metafeature collection.*

**Example 744**

Using `getValue` to retrieve some information about the predefined `PLUS_INFINITY` attribute:

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity def = model.findByGuid("@Predefined@PLUS_INFINITY");
String name = def.getValue("Name", 0);
boolean isConst = def.getValue("changeability",
0).equals("CkFrozen");
String type = def.getValue("type", 0);
System.out.println(name + " is a " + (isConst ? "constant" : "non-
constant") + " attribute of type " + type);
```

Running this example will print the output:

```
PLUS_INFINITY is a constant attribute of type ref:Real
```

Note the format of the text encoded value of a metafeature that is a reference (like ‘type’ above). The name of the target definition will be prefixed according to the rules described in “GetValue” on page 2089 in Chapter 76, *COM API*.

### getEntity

```
ITtdEntity getEntity(String strMetaFeature, int index)
throws APIError;
```

Gets the value of the specified metafeature as an `ITtdEntity` reference. Use it for metafeatures of Metaclass type that have either single or multiple multiplicity. In the case of multiple multiplicity use the `index` to specify which entity to get. If no metafeature has the specified name, an `APIError` exception will be thrown.

**Note**

*If the metafeature is unbound, the return value will be* `null`*. You may then want to use the getValue method to get the value as a string representation instead, or getReference to obtain the model representation of the unbound reference.*

**Note**

`index` *is an index starting at 1, and 0 specifies the last entity in the metafeature collection.*

**Example 745**

Using `getEntity` to retrieve the type of the predefined `PLUS_INFINITY` attribute:

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity def = model.findByGuid("@Predefined@PLUS_INFINITY");
ITtdEntity type = def.getEntity("type", 0);
if (type != null)
  System.out.println("The type of PLUS_INFINITY is " +
type.getValue("Name", 0));
```

Running this example on a bound model will print the output:

```
The type of PLUS_INFINITY is Real
```

---

### getEntities

```
List<ITtdEntity> getEntities(String strMetaFeature)
throws APIError;
```

Gets the value of the specified metafeature as a list of entities. Use it for metafeatures of Metaclass type that have either single or multiple multiplicity. In the case of single multiplicity the result list will contain one or zero entities. If no metafeature has the specified name, an `APIError` exception will be thrown.

See Example 734 on page 2292 for an example of using `getEntities` to obtain the list of libraries in a model.

### getReference

```
ITtdEntity getReference(String strMetaFeature, int
index) throws APIError;
```

Returns the identifier representation of a metafeature reference. In the case of a metafeature with multiple multiplicity use the index to specify which reference to get. If no metafeature has the specified name, or the metafeature is not a reference, an `APIError` exception will be thrown.

**Note**

`index` *is an index starting at 1, and 0 specifies the last entity in the metafeature collection.*

**Example 746**

This example constructs an attribute with a type reference that is not just a simple name. The model representation of the type reference is obtained by calling `getReference`, and its unparsed representation is printed.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
List<ITtdEntity> lst = model.parse("Predefined::String<MyClass>
my_var;", "Definition");
for (ITtdEntity e : lst)
{
  ITtdEntity typeRef = e.getReference("Type", 0);
  System.out.println("The type of 'my_var' is " +
typeRef.unparse());
  e.delete();
}
```

Running this example prints the output:

```
The type of 'my_var' is Predefined::String<MyClass>
```

### getOwner

```
ITtdEntity getOwner();
```

Returns the composition owner of the entity. If the entity has no owner `null` is returned.

**Example 747**

This example locates the `equal` operation of the predefined `Integer` type. It then prints the composition owners of that operation.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity e =
model.findByGuid("@Predefined@Integer@equal@Boolean@Integer@Integer
");
do
{
  if (e.isKindOf("Definition"))
    System.out.println(e.getValue("Name", 0));
  else
    System.out.println(e.getMetaClassName());

  e = e.getOwner();
}
while (e != null);
```

Running this example prints the output:

```
equal
Integer
Predefined
Session
```

---

### getMetaClassName

```
String getMetaClassName();
```

Returns the name of the entity's Metaclass.

See Example 747 on page 2302 for an example of using `getMetaClassName`.

### getReferingEntities

```
List<ITtdEntity> getReferingEntities(String
strMetaFeature);
```

Returns the entities that refer to the entity through the specified metafeature. `strMetaFeature` may be an empty string in order to find all referring entities regardless of through which metafeature they refer to the entity.

**Example 748** ———————————————————————————————

This example locates the predefined `Boolean` type, and examines the number of entities that refer to this type, either as their type or in another way.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity integer = model.findByGuid("@Predefined@Boolean");
List<ITtdEntity> r1 = integer.getReferingEntities("Type");
List<ITtdEntity> r2 = integer.getReferingEntities("");
System.out.println("There are " + r1.size() + " type references to
Boolean.");
System.out.println("There are " + (r2.size() - r1.size()) + " other
```

```
references to Boolean.");
```

### getTaggedValue

```
ITtdEntity getTaggedValue(String strSelector, boolean
interpretIdentsAsGuids) throws APIError;
```

Returns the expression representing the tagged value selected by the selector pattern. The entity should be either an extendable element (in which case the tagged value is looked for among the applied stereotype instances of the extendable element) or an instance expression (in which case the tagged value is looked for in that particular instance only).

A selector pattern specifies the path from the entity to the tagged value using the textual UML syntax of an instance expression. The function will check that the pattern matches both the instance tree (by structure) and the corresponding signatures (by name or GUID). If `interpretIdentsAsGuids` is true, identifiers of the pattern will be interpreted as GUIDs. Otherwise they will be interpreted as names. If no tagged value is selected by the selector pattern, `null` is returned.

Some selector pattern examples:

```
"T1 (. .)"
```

will return the first instance of the applied T1 stereotype

```
"T1 (. x .)"
```

will return the tagged value of the attribute x in the T1 stereotype

```
"T1 (. a1 = T2 (. a2 .) .)"
```

will return the value of a2 in a T2 instance being the value of T1.a1.

**Note**

*Although a selector pattern on the form "*`X (. .)`*" can be used to test if a stereotype X is applied on an entity, it is better to use hasAppliedStereotype for this purpose. The reason is that a stereotype that is automatically applied (due to a non-optional extension on a matching metaclass) will not be instantiated until at least one of its attributes gets a tagged value that differs from the default value of the attribute.*`getTaggedValue` *thus has no stereotype instance to return for that particular case. However, when used with a selector pattern that selects a tagged value,* `getTaggedValue` *will also consider such automatically applied stereotypes.*

**Example 749**

This example locates the `TTDMetamodel` representation of the `Model` metaclass. It then uses `getTaggedValue` to extract the 'base' tagged value of the <<metaclass>> stereotype.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity e = ((ITtdEntity)
model).findByName("TTDMetamodel::Model");
ITtdEntity taggedValue = e.getTaggedValue("metaclass (. 'base' .)",
false);
System.out.println("Model's base metaclass is " +
taggedValue.unparse());
```

Running this example prints the output:

```
Model's base metaclass is "Session"
```

### hasAppliedStereotype

```
boolean hasAppliedStereotype(String strStereotype,
boolean guid);
```

Determines if the entity has a certain stereotype applied. The stereotype can be specified either by name or by GUID. In the latter case `guid` should be set to true.

This is the recommended method for checking for applied stereotypes on an entity. It will consider both explicitly applied stereotypes, and stereotypes that are automatically applied due to non-optional extensions from a metaclass that matches the metaclass of the entity.

**Example 750**

This example prints the names of all libraries in a model, and whether the library is an ordinary library or a profile library. Profile libraries have the `<<profile>>` stereotype applied.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
List<ITtdEntity> lst = ((ITtdEntity) model).getEntities("Library");
for (ITtdEntity e : lst)
{
  System.out.print(e.getValue("Name", 0) + " is ");
  if (e.hasAppliedStereotype("@Predefined@profile", true))
    System.out.println("a profile library");
  else
    System.out.println("an ordinary library");
}
```

### isKindOf

```
boolean isKindOf(String strMetaClass);
```

Returns true if the entity is of the specified Metaclass, false otherwise.

`isKindOf` returns true also if the entity's metaclass inherits from the specified metaclass. To perform an exact match use getMetaClassName.

**Example 751**

This example creates a new StateMachine entity. A StateMachine is a special kind of Operation.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity sm = model.New("StateMachine");
if (sm.isKindOf("StateMachine"))
  System.out.println("sm is a StateMachine");
if (sm.isKindOf("Operation"))
  System.out.println("sm is an Operation");
String metaclass = sm.getMetaClassName();
System.out.println("Exact metaclass is " + metaclass);
```

Running this example prints the output:

```
sm is a StateMachine
sm is an Operation
Exact metaclass is StateMachine
```

---

### unparse

```
String unparse() throws APIError;
```

Returns the unparsed representation of the entity using textual UML syntax. The following kinds of entities can be unparsed:

- Definitions
- Actions
- Expressions

An attempt to unparse another kind of entity will yield an `APIError`. The text returned by unparse can later be passed to `ITtdModel::`parse in order to build the corresponding model entities from the syntax again.

See Example 746 on page 2302, Example 749 on page 2305 or Example 752 on page 2307 for an example of using `unparse`.

### setValue

```
void setValue(String strMetaFeature, String strValue,
int index) throws APIError;
```

Sets the value of a metafeature. The value is encoded as a string.

The `setValue` method can be used on all writable metafeatures of an entity that can have their values encoded as a string. This is the case for all metafeatures except derived features (which are read-only), owner links and composition links.

If the metafeature has non-single multiplicity, the `index` parameter can be used to insert the value before the value at the specified position.

**Note**

`index` *is an index starting at 1, and 0 specifies the last entity in the metafeature collection.*

If an error occurs (e.g. because of a non-existing metafeature) an `APIError` exception is thrown.

**Example 752**

This example creates an attribute in a model, and sets up the name and type of the attribute using `setValue`:

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity attribute = ((ITtdEntity) model).create("Attribute",
false, "");
attribute.setValue("Name", "a", 0);
attribute.setValue("Type", "ref:Integer", 0);
System.out.println(attribute.unparse());
```

Running this example will print the output:

```
Integer a;
```

Note the format of the text encoded value of a metafeature that is a reference (like 'Type' above). The name of the target definition will be prefixed according to the rules described in "GetValue" on page 2089 in Chapter 76, *COM API*.

**Important!**

*Calling* `setValue` *on a reference metafeature implicitly deletes the entity that represents the reference (obtained by getReference). See Example 757 on page 2312 for more information.*

### setEntity

```
void setEntity(String strMetaFeature, ITtdEntity entity,
int index) throws APIError;
```

Sets the value of a metafeature. The value is an entity reference. Use it for all writable metafeatures of Metaclass type. If the metafeature has non-single multiplicity, the index argument can be used to insert the entity before the value at the specified position.

**Note**

`index` *is an index starting at 1, and 0 specifies the last entity in the metafeature collection.*

**Note**

*If* `entity` *is* `null`*, the metafeature will be made unbound. However this does not apply for owner links. To unset an owner link, i.e. to unlink an entity from its composition owner, use unlinkFromOwner.*

If an error occurs (e.g. because of a non-existing metafeature) an `APIError` exception is thrown.

**Example 753**

This example creates a class from its textual syntax, and inserts the class into a model using `setEntity`.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity cls = model.parse("class C {}", "Definition").get(0);
((ITtdEntity) model).setEntity("OwnedMember", cls, 0);
if (cls.getOwner() == (ITtdEntity) model)
  System.out.println("Class successfully inserted in model!");
```

**Important!**

*Calling* `setEntity` *on a reference metafeature implicitly deletes the entity that represents the reference (obtained by getReference). See Example 757 on page 2312 for more information.*

### setTaggedValue

```
void setTaggedValue(String strSelector, String strValue,
boolean overwrite) throws APIError;
```

Sets the tagged value of the attribute selected by the selector pattern. See getTaggedValue for the format of this pattern. The entity can either be an element with applied stereotypes or any instance expression. In the latter case the pattern is matched against the instance expression, while in the former case the first matching applied stereotype instance will be used.

Usually you want an existing value for the selected attribute to be overwritten, but if `overwrite` is false this will not be the case.

**Note**

*In order to be able to overwrite an existing value for the specified attribute, the instance expression must be bound to the Signature of which it is an instance. If that is not the case, an* `APIError` *exception will be thrown. Use bind to make sure the instance expression is bound before calling* `setTaggedValue`.

After the new value has been set all references in the entity is attempted to be bound, so that the set value can be accessed by getTaggedValue directly after the call to this method.

`strValue` must be a valid UML expression, and the type of this expression should match the type of the attribute for which the tagged value is set. For example, if the attribute is typed by `Charstring`, `strValue` should be a string literal.

**Example 754**

This example creates a package in a new model. It then applies the <<icon>> stereotype on the package. Finally `setTaggedValue` is used to set the 'IconFile' tagged value.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity iconStereotype = ((ITtdEntity)
model).findByName("TTDStereotypeDetails::icon");
ITtdEntity pkg = ((ITtdEntity) model).create("Package", false,
"OwnedMember");
pkg.applyStereotype(iconStereotype,
TtdReferenceKind.TTD_RK_MINIMAL_QUALIFIER, null);
pkg.bind("");
pkg.setTaggedValue("icon (. IconFile .)",
"\"C:\\\\pics\\\\x.jpg\"", true);
System.out.println(pkg.unparse());
```

Running this example will print the output:

```
<<icon(.IconFile = "C:\\pics\\x.jpg".)>> package '' {

}
```

Note that since `IconFile` is a `Charstring` attribute, the tagged value must be enclosed in double quotes, and contained backslashes must be escaped.

---

### create

```
ITtdEntity create(String strMetaClass, boolean
buildModelForPresentations, String strMetaFeature)
throws APIError;
```

Creates an entity of the specified metaclass as an immediate child of this entity. The created entity will be inserted in the specified metafeature. The metafeature can be left unspecified (an empty string) as long as the entity only contains one metafeature that can contain the created entity.

**Note**

*The parameter* `buildModelForPresentations` *is only used in an interactive execution environment. It can then be set to true in order to automatically create the semantic entity behind a created presentation element (such as a symbol or a line). For example if creating a ClassSymbol, the corresponding Class would be automatically created.*

**Example 755**

This example creates a package with a class diagram in a new model.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity pkg = ((ITtdEntity) model).create("Package", false,
"OwnedMember");
ITtdEntity diagram = pkg.create("ClassDiagram", false, "");
```

Note that there are two metafeatures at the top model level where packages can be created: “Library” (for library packages) and “OwnedMember” for ordinary packages. Therefore we must specify the name of the metafeature in the first call to `create` above.

---

### createInstance

```
ITtdEntity createInstance() throws APIError;
```

Call this method on entities that are signatures that can be instantiated, to create an instance of the signature. One use for this method is to create an instance of a stereotype in order to apply the stereotype on an element (however, applyStereotype is the easiest way to achieve this). Another use is for creating instance models.

In case the creation fails an `APIError` exception will be thrown.

**Note**

*It is the responsibility of the client to take care of the returned entity. It should either be inserted into the model, or deleted, to avoid a memory leak. See Memory Management for more information.*

`createInstance` implements the UML semantics of signature instantiation. If the signature contains parts with an initial cardinality these will also be instantiated recursively.

**Example 756**

This example locates the predefined `<<cppHeaderFile>>` stereotype, and creates an instance of it. The unparsed representation of that instance is then printed.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity stereotype =
model.findByGuid("@TTDFileModel@cppHeaderFile");
ITtdEntity instance = stereotype.createInstance();
System.out.println(instance.unparse());
```

Running this example will print the output:

```
cppHeaderFile (.includeProtSettings = IncludeProtectionSettings
(..).)
```

`<<cppHeaderFile>>` has an attribute ‘includeProtSettings’ which is a part with multiplicity 1. Hence, when `<<cppHeaderFile>>` is instantiated a contained instance of `IncludeProtectionSettings` will be created too.

---

### delete

```
void delete();
```

Deletes the entity. Memory allocated by the entity will be freed.

Note that the Java garbage collector only can manage the memory allocated by the JVM. Model entities must be deleted, when they no longer are needed, by calling `delete` (see Memory Management).

**Important!**

*Do not access a reference to an entity after it has been deleted! Doing so usually leads to an access error and program crash.*

In most cases it is not difficult to comply with the above rule, since deleting an entity is an explicit operation. However, there is one situation when use of other API methods will implicitly delete an entity. This happens when setting a value of a reference metafeature using setValue or setEntity as shown in the example below.

**Example 757**

This example creates an attribute typed by `Integer`. The entity representing the type reference is then obtained by calling getReference. Then the attribute is retyped to `Boolean` using setEntity. This call will implicitly delete the `Integer` reference entity. Hence, the program must be careful not to access this entity afterwards.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity attribute = model.parse("Integer a;",
"Definition").get(0);
ITtdEntity typeRef = attribute.getReference("Type", 0);
attribute.setEntity("Type",
model.findByGuid("@Predefined@Boolean"), 0);
// System.out.println(typeRef.unparse()); <--- ERROR! typeRef is
deleted!
```

For an example of using an explicit call of `delete`, see Example 739 on page 2295.

### XMLEncode

```
String XMLEncode() throws APIError;
```

Returns the XML encoding of the entity. If an error occurs, an `APIError` exception will be thrown.

See Example 740 on page 2295 for an example of how to use `XMLEncode`.

### metaVisit

```
void metaVisit(ITtdMetaVisitCallback callbackInterface,
boolean visitAll) throws APIError;
```

Performs a Metamodel driven traversal of the model rooted at the entity. If `visitAll` is false libraries and the predefined package will be excluded from the traversal.

For each visited entity, the onVisitedEntity method will be called on the `callbackInterface` object. This method is called for an entity when the model traversal reaches that entity, but before its contained entities have been

visited. When all contained entities have been visited, the onAfterVisitedEntity method will be called on the `callbackInterface` object. This allows actions to be performed on the "back recursion" of the traversal.

**Example 758**

This example shows how the `MetaVisit` method can be used in order to print the name of all definitions in a model. First a class implementing the callback interface is defined:

```
public class DefinitionFinder implements ITtdMetaVisitCallback
{
  public boolean onVisitedEntity(ITtdEntity visitedEntity)
  {
    if (visitedEntity.isKindOf("Definition"))
    {
      try
      {
        System.out.println(visitedEntity.getValue("Name", 0));
      }
      catch (APIError e) {}
    }
    return true;
  }

  public void onAfterVisitedEntity(ITtdEntity visitedEntity)
  {
  }
```

`MetaVisit` can now be called on the model like this:

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
((ITtdEntity) model).metaVisit(new DefinitionFinder(), true);
```

For more information about the callback interface used with `MetaVisit`, see ITtdMetaVisitCallback.

### bind

```
void bind(String strMetaFeature) throws APIError;
```

Attempts to bind the specified metafeature on the entity (or all metafeatures if `strMetaFeature` is empty). If a non-existing metafeature is specified, an `APIError` exception will be thrown.

**Example 759**

This example creates an attribute typed by Integer. The type reference of the attribute is accessed before and after calling `bind` on the type reference.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
```

```
ITtdModel model = ma.createModel();
ITtdEntity attribute = ((ITtdEntity) model).create("Attribute",
false, "");
attribute.setValue("Type", "ref:Integer", 0);
ITtdEntity type = attribute.getEntity("Type", 0);
System.out.println("type is " + ((type == null) ? "null" :
type.getValue("Name", 0)));
attribute.bind("Type");
type = attribute.getEntity("Type", 0);
System.out.println("type is " + ((type == null) ? "null" :
type.getValue("Name", 0)));
```

Running this example will print the output:

```
type is null
type is Integer
```

As can be seen the type reference is not bound to the `Integer` definition until after `bind` has been called.

---

### locate

```
void locate() throws APIError;
```

This method is only available in an interactive execution environment.

The method locates visually the entity in the ModelView and/or diagrams. The effect in the IDE will be that the entity is shown in the model view (if possible) and in the diagrams (if a presentation for the entity exists in a diagram, that is).

If this method is used when the IDE is not available (in a non-interactive execution environment), an `APIError` exception will be thrown.

**Example 760**

This example locates the predefined `Boolean` type.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity b = model.findByGuid("@Predefined@Boolean");
b.locate();
```

---

### clone

```
ITtdEntity clone(boolean preserveBindings, boolean
preserveGuids) throws APIError;
```

Creates a clone of the entity.

The recommended value for both the parameters is `false`. This means that the clone will be unbound and have new unique GUIDs (i.e. the copy of the entity itself and the copy of all contained entities will get new unique GUIDs).

If `preserveBindings` is set to `true`, the clone will have the same bindings as the original entity. In order to be able to preserve bindings of the clone, the original entity must belong to a model (i.e. getModel called on the original entity must not return `null`).

If `preserveGuids` is set to `true`, the clone will have the same GUID as the original entity.

**Important!**

*Be careful when cloning an entity without changing GUIDs. Such a clone should not be inserted into the same model as the original entity, or GUID conflicts will arise. If a model with GUID conflicts is saved, it might not be possible to load again.*

If the cloning fails for one reason or another an `APIError` exception will be thrown.

**Note**

*It is the responsibility of the client to take care of the returned entity. It should either be inserted into the model, or deleted, to avoid a memory leak. See Memory Management for more information.*

**Example 761**

This example creates an expression by parsing textual UML syntax. The resulting expression is then cloned and unparsed. Finally it is deleted to avoid a memory leak.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity expr = model.parse("1+x*(3+z)", "Expression").get(0);
ITtdEntity clone = expr.clone(false, false);
System.out.println(clone.unparse());
clone.delete();
```

Running this example will print the output:

```
1 + x * (3 + z)
```

**move**

```
void move(ITtdEntity newOwner, String metafeature, int
```

```
index) throws APIError;
```

Moves the entity from its current location in the model into the context of `newOwner`. If the entity would fit in more than one metafeature of the new owner, `metaFeature` must be specified to disambiguate. The `index` argument may be specified to control the position where to move the entity when the target metafeature has non-single multiplicity.

If the move fails an `APIError` exception will be thrown.

**Example 762**

This example creates a new package “MyPackage” in a model. The package is created as an ordinary package (located in the “OwnedMember” composition). Then the package is moved into the “Library” composition of the model. It is inserted there as the 3rd library package. Finally the names of all library packages of the model are printed.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdEntity model = (ITtdEntity) ma.createModel();
ITtdEntity pkg = model.create("Package", false, "OwnedMember");
pkg.setValue("Name", "MyPackage", 0);
pkg.move(model, "Library", 3);
List<ITtdEntity> lst = model.getEntities("Library");
for (ITtdEntity e : lst)
{
  System.out.println(e.getValue("Name", 0));
}
```

## getModel

```
ITtdModel getModel();
```

Returns the model to which the entity belongs. If the entity does not belong to a model `null` is returned. This method is often the most convenient way to get an `ITtdModel` interface from the context of an `ITtdEntity` interface. It is particularly useful when the API client works with more than one model simultaneously.

**Example 763**

This example gets the last library in a created model, and calls `getModel` to make sure it belongs to the same model that was created (which it of course will do).

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdEntity model = (ITtdEntity) ma.createModel();
ITtdEntity library = model.getEntity("Library", 0);
if (model == library.getModel())
```

```
System.out.println(library.getValue("Name", 0));
```

### unlinkFromOwner

```
void unlinkFromOwner();
```

Unlinks the entity from its current owner. The entity will not be deleted, and can for example be inserted in another place in the model, or in another model.

**Example 764**

This example creates two models. In the first model a class is created. Then `unlinkFromOwner` is called to unlink the class from the first model, and `setEntity` is called to insert it in the second model instead.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdEntity m1 = (ITtdEntity) ma.createModel();
ITtdEntity m2 = (ITtdEntity) ma.createModel();
ITtdEntity cls = m1.create("Class", false, "");
cls.unlinkFromOwner();
m2.setEntity("OwnedMember", cls, 0);
System.out.println(m1.getEntities("OwnedMember").size());
System.out.println(m2.getEntities("OwnedMember").size());
```

Running this example will print the output:

```
0
1
```

Note that the move method can be used to move an entity in one single step.

### replace

```
void replace(ITtdEntity replacementEntity) throws
APIError;
```

Replaces the entity with another entity, without deleting the original entity. If the replacement is not possible to perform, an `APIError` exception will be thrown.

**Note**

*If the entity is an identifier representing a reference it will be replaced with a clone of* `replacementEntity`*, rather than* `replacementEntity` *itself.*

**Example 765**

This example creates a model with a class. It then replaces the class with an interface. The original class is deleted to avoid a memory leak.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel m = ma.createModel();
ITtdEntity cls = ((ITtdEntity) m).create("Class", false, "");
cls.replace(m.New("Interface"));
cls.delete();
ITtdEntity i = ((ITtdEntity) m).getEntity("OwnedMember", 0);
System.out.println(i.getMetaClassName());
```

Running this example will print the output:

```
Interface
```

## getContainerMetaFeature

```
String getContainerMetaFeature();
```

Returns the name of the metafeature in which the entity is contained. If the entity is orphan an empty string is returned.

**Example 766**

This example creates a model with an operation. It then prints the names of the container metafeatures for different parts of the operation definition.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel m = ma.createModel();
ITtdEntity op = m.parse("void foo(Integer p1 = 3);",
"Definition").get(0);

class P implements ITtdMetaVisitCallback
{
  public boolean onVisitedEntity(ITtdEntity visitedEntity)
  {
    System.out.println(visitedEntity.getMetaClassName() + " is
contained in metafeature " +
visitedEntity.getContainerMetaFeature());
    return true;
  }

  public void onAfterVisitedEntity(ITtdEntity visitedEntity) { }
};

op.metaVisit(new P(), true);
```

Running this example will print the output:

```
Operation is contained in metafeature
Parameter is contained in metafeature Parameter
IntegerValue is contained in metafeature DefaultValue
```

### findByName

```
ITtdEntity findByName(String strName);
```

Finds an entity by a name (possibly qualified) from the context of the entity. `strName` should be a valid UML identifier.

If no entity is found, `null` is returned.

For examples of using `findByName`, see Example 749 on page 2305 and Example 754 on page 2309.

### getDescriptiveName

```
String getDescriptiveName();
```

Returns a descriptive name of the entity. The description includes the Meta-class of the entity, its name (full signature for event classes) and its location in the model.

**Example 767**

This example locates the predefined `Real` type, and prints its descriptive name:

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel m = ma.createModel();
ITtdEntity real = m.findByGuid("@Predefined@Real");
System.out.println(real.getDescriptiveName());
```

Running this example will print the output:

```
DataType 'Real' in Predefined
```

## ITtdResource

The `ITtdResource` interface is implemented by the Resource class of the Metamodel, which represents a resource where a UML model could be persistently stored. Typically a Resource corresponds to a `.u2` file.

**Note**

*Since a Resource also is an Entity, a cast from* `ITtdResource` *to* `ITtdEntity` *will always succeed.*

### save

```
void save() throws APIError;
```

Saves the model entities that are associated with the resource on which the method is called. For the common case when the resource represents a `.u2` file, this means that the file will be saved.

If the resource cannot be saved an `APIError` exception will be thrown.

**Example 768**

This example creates a new model, containing a package with an interface, and saves it to a `.u2` file.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity modelRoot = (ITtdEntity) model;
ITtdEntity pkg = modelRoot.create("Package", false, "OwnedMember");
ITtdEntity cls = pkg.create("Interface", false, "");
ITtdResource resource = model.createResource("C:\\temp\\test.u2");
// Insert the created package pkg as a root of the resource
((ITtdEntity) resource).setEntity("Root", pkg, 0);
resource.save();
```

# ITtdPresentationElement

The `ITtdPresentationElement` interface is implemented by the PresentationElement class of the Metamodel. A presentation element is an element with a graphical appearance, for example a symbol, line or diagram.

**Note**

*Since a PresentationElement also is an Entity, a cast from* `ITtdPresentationElement` *to* `ITtdEntity` *will always succeed.*

### generateEMF

```
void generateEMF(String strFileName, int maxWidth, int
maxHeight, boolean optimizeForVectorGraphics, boolean
includeFrame) throws APIError;
```

Generates an EMF file (Enhanced Meta File) for the graphical appearance of a presentation element. The presentation element will have the same appearance in this EMF file as when shown in the Tau editors.

**Note**

*This is a deprecated function. Use generateEMFEx instead.*

**Note**

`generateEMF` *is only available in the interactive execution environment. An attempt to call it from another execution environment will yield an* `APIError` *exception.*

### generateEMFEx

```
void generateEMFEx(String strFileName, int maxWidth, int
maxHeight, boolean includeFrame, int scaleFactor) throws
APIError;
```

Generates an EMF file (Enhanced Meta File) for the graphical appearance of a presentation element. The presentation element will have the same appearance in this EMF file as when shown in the Tau editors.

**Note**

`generateEMFEx` *is only available in the interactive execution environment. An attempt to call it from another execution environment will yield an* `APIError` *exception.*

**Example 769**

This example executes a query to find all diagrams in a model. It then generates an EMF file for the first diagram found. A JPG file is also generated using generateImage.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity query =
model.findByGuid("@TTDQuery@ExecuteQueryExpression");
LinkedList<Object> agentParameters = new LinkedList<Object>();
agentParameters.add(new LinkedList<Object>());
agentParameters.add("GetAllEntities().select(IsKindOf(\"Diagram\"))
");
model.invokeAgent(query, (ITtdEntity) model, agentParameters);
List<Object> lst = (List<Object>) agentParameters.element();
if (lst.size() > 0)
{
  ITtdPresentationElement pe = (ITtdPresentationElement)
lst.get(0);
  pe.generateEMFEx("C:\\temp\\d.emf", 0, 0, true, 0);
  pe.generateImage(ImageKind.IK_JPEG, "C:\\temp\\d.jpg");
}
```

### generateImage

```
void generateImage(ImageKind imgKind, String
strFileName) throws APIError;
```

Generates an image file from the graphical appearance of a presentation element. The presentation element will have the same appearance in this image file as when shown in the Tau editors.

Valid values in the `ImageKind` enumeration, and their meaning, are listed in the table below:

| ImageKind | Description |
|---|---|
| `IK_JPEG` | Generate a JPEG image file. |
| `IK_BMP` | Generate a BMP image file. |
| `IK_GIF` | Generate a GIF image file. |
| `IK_TIFF` | Generate a TIFF image file. |
| `IK_TARGA` | Generate a TGA (Targa) image file. |
| `IK_DIB` | Generate a device independent bitmap file. |
| `IK_PCX` | Generate a PCX image file. |

For an example of how to use `generateImage` see Example 769 on page 2321.

## ITtdSymbol

The `ITtdSymbol` interface is implemented by the Symbol class of the Metamodel. A symbol is a presentation element with a two-dimensional graphical appearance.

**Note**

*Since a Symbol also is a PresentationElement and an Entity, a cast from* `ITtdSymbol` *to* `ITtdPresentationElement` *or* `ITtdEntity` *will always succeed.*

### setSize

```
void setSize(int width, int height);
```

Sets the size of the symbol. The unit of the width and height is 1/10:th of a millimeter. In a non-interactive execution environment `setSize` will just update the value of the 'size' metafeature for the symbol. In the interactive execution environment, however, `setSize` can also perform additional model

changes related to the resize. This could for example happen if the symbol size has a semantic significance. It is therefore recommended to always use `setSize` in order to set the size of a symbol.

For an example of how to use `setSize` see Example 770 on page 2323.

### setPosition

```
void setPosition(int x, int y);
```

Sets the position of the symbol. The unit of the x and y parameters is 1/10:th of a millimeter. In a non-interactive execution environment `setPosition` will just update the value of the 'position' metafeature for the symbol. In the interactive execution environment, however, `setPosition` can also perform additional model changes related to the repositioning. This could for example happen if the symbol position has a semantic significance. It is therefore recommended to always use `setPosition` in order to set the position of a symbol.

**Example 770**

This example creates a new model with a class diagram containing a class symbol. It then sets the size and position of this symbol.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdEntity model = (ITtdEntity) ma.createModel();
ITtdEntity diagram = model.create("ClassDiagram", false, "");
ITtdSymbol symbol = (ITtdSymbol) diagram.create("ClassSymbol",
true, "");
symbol.setSize(400, 200);
symbol.setPosition(350, 200);
System.out.println(((ITtdEntity) symbol).XMLEncode());
```

## ITtdExpression

The `ITtdExpression` interface is implemented by the Expression class of the Metamodel. It represents an expression in the model.

**Note**

*Since an Expression also is an Entity, a cast from* `ITtdExpression` *to* `ITtdEntity` *will always succeed.*

### getType

```
ITtdEntity getType();
```

Computes and returns the type of the expression. If the type cannot be computed (for example because the expression contains unbound references, or because the expression is not part of a model) `null` is returned.

**Note**

*The returned entity is not guaranteed to be part of the model. In some cases the type is not explicitly defined in the model, and in that case a temporary entity which represents the type will be returned. Be careful not to delete such a temporary entity. getModel can be used to determine if the returned entity is part of the model.*

**Example 771**

This example creates some expressions by parsing textual UML syntax. Each expression is inserted as the default value of an attribute in the model. Then its type is computed and printed.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity attribute = ((ITtdEntity) model).create("Attribute",
false, "");

class ParseAndPrintType
{
  ParseAndPrintType(ITtdEntity attribute, String s) throws APIError
  {
    ITtdModel model = attribute.getModel();
    ITtdExpression e = (ITtdExpression) model.parse(s,
"Expression").get(0);
    attribute.setEntity("DefaultValue", (ITtdEntity) e, 0);
    ITtdEntity type = e.getType();
    if (type != null)
      System.out.println("The type of " + s + " is " +
type.getValue("Name", 0));
  }
}

new ParseAndPrintType(attribute, "1+2+3");
new ParseAndPrintType(attribute, "2.4 * -2.78");
new ParseAndPrintType(attribute, "true and false");
```

Running this example will print the output:

```
The type of 1+2+3 is Integer
The type of 2.4 * -2.78 is Real
The type of true and false is Boolean
```

### evaluateConstantIntegralExpression

```
int evaluateConstantIntegralExpression() throws
APIError;
```

Evaluates the value of the expression, which is expected to be a constant integral expression. If it is not, or the evaluation fails for some other reason, an `APIError` exception will be thrown.

**Example 772**

This example creates an integral expression by parsing textual UML syntax. The expression is then evaluated and the result is printed.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdExpression e = (ITtdExpression) model.parse("95-8*2",
"Expression").get(0);
int res = e.evaluateConstantIntegralExpression();
System.out.println(((ITtdEntity) e).unparse() + " = " + res);
```

Running this example will print the output:

```
95 - 8 * 2 = 79
```

### getInstanceChildExpression

```
ITtdExpression getInstanceChildExpression(String
strName) throws APIError;
```

Use this method on an instance expression (for example a stereotype instance, or the instance expression of a named instance) to obtain the right-hand side of a contained assignment, where the left-hand side of the assignment is an identifier matching `strName` (which thus should be a valid identifier).

If no matching child expression is found, `null` is returned.

In case of an erroneous usage an `APIError` exception will be thrown.

**Example 773**

This example creates a named instance by parsing textual UML syntax. The values of its slots (which are special kinds of assignments) are printed.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity inst = model.parse("value x : C (. a = 12, b = true,
BASE::c = \"Hello\" .);", "Definition").get(0);
ITtdExpression e = (ITtdExpression) inst.getEntity("Instance", 0);
ITtdExpression s1 = e.getInstanceChildExpression("a");
ITtdExpression s2 = e.getInstanceChildExpression("b");
ITtdExpression s3 = e.getInstanceChildExpression("BASE::c");
System.out.println(inst.getMetaClassName() + " slot values are:");
System.out.println(((ITtdEntity) s1).unparse());
System.out.println(((ITtdEntity) s2).unparse());
System.out.println(((ITtdEntity) s3).unparse());
```

Running this example will print the output:

```
NamedInstance slot values are:
12
true
"Hello"
```

# ITtdMetaVisitCallback

The `ITtdMetaVisitCallback` interface is a callback interface that clients using the metaVisit method must implement.

### onVisitedEntity

```
boolean onVisitedEntity(ITtdEntity visitedEntity);
```

Called when using metaVisit to traverse a model. `visitedEntity` is the currently visited entity in the model. This method is called before visiting the composition children of `visitedEntity`. If false is returned traversal will not continue with the composition children of `visitedEntity`.

See Example 758 on page 2313 for an example of how to use `onVisitedEntity`.

### onAfterVisitedEntity

```
void onAfterVisitedEntity(ITtdEntity visitedEntity);
```

Called when using metaVisit to traverse a model. `visitedEntity` is the currently visited entity in the model. This function is called after the composition children of `visitedEntity` have been visited, and can thus be used in order to perform actions on the “back recursion” of the model traversal.

**Example 774**

This example traverses a parsed expression and prints some logging before and after each entity of the expression is visited.

```
ITtdModelAccess ma = TauModelAPI.getModelAccess();
ITtdModel model = ma.createModel();
ITtdEntity e = model.parse("1 + x", "Expression").get(0);

class P implements ITtdMetaVisitCallback
{
  public boolean onVisitedEntity(ITtdEntity visitedEntity)
  {
    System.out.println("Before visiting " +
visitedEntity.getMetaClassName());
    return true;
  }
```

```
  public void onAfterVisitedEntity(ITtdEntity visitedEntity)
  {
    System.out.println("After visiting " +
visitedEntity.getMetaClassName());
  }
};

e.metaVisit(new P(), true);
```

The expression has the following representation in the model:

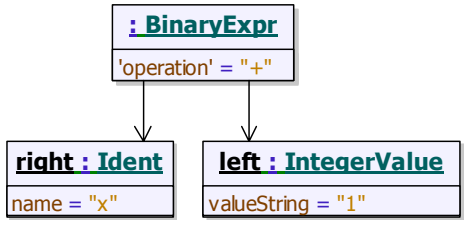



*Figure 288: Model representation of expression 1 + x*

Running this example will therefore print the output:

```
Before visiting BinaryExpr
Before visiting Ident
After visiting Ident
Before visiting IntegerValue
After visiting IntegerValue
After visiting BinaryExpr
```

Note that the order in which the composition children is determined by their order in the Tau metamodel. For example, from the above output we can conclude that the right hand side expression is defined before the left hand side expression of a binary expression.

---

# ITtdMessageList

The `ITtdMessageList` interface represents a list of messages. It is used as a callback interface in API methods which may produce messages, for example loadProject.

### addMessage

```
void addMessage(String text, MessageSeverity severity,
ITtdEntity subject) throws APIError;
```

Adds a new message to the list with the specified severity and, optionally, a subject entity. The subject entity will be associated with the message. In an interactive execution environment the subject entity can be located from the message.

Valid values in the `MessageSeverity` enumeration, and their meaning, are listed in the table below:

| MessageSeverity | Description |
|---|---|
| `MS_INFORMATION` | The message is an information message. |
| `MS_WARNING` | The message is a warning message. |
| `MS_ERROR` | The message is an error message. |
| `MS_FATAL` | The message is a fatal error message. |

See Example 733 on page 2291 for an example of using `addMessage`.

## ITtdStudioAccess

The `ITtdStudioAccess` interface is the entry point for accessing functionality of the Tau IDE which is not specific to UML modeling.

**Note**

*The `ITtdStudioAccess` interface is of no use in a non-interactive execution environment, since the Tau IDE is then not available. It can only be used when using Tau Access to access a running instance of Tau.*

See Tau Access for information about how to obtain the `ITtdStudioAccess` interface.

### openWorkspace

```
ITtdWorkspace openWorkspace(String path) throws
APIError;
```

Opens a Tau workspace file (`.ttw` file) in the Tau IDE. All projects contained in the workspace will be loaded. If an existing workspace is open in the Tau IDE it will first be closed.

This method is the equivalent of opening a workspace using the **File - Open workspace** menu.

`path` is the path to the workspace to open. If it is a relative path, it will be interpreted as relative to the current working directory of Tau (which is typically the `bin` directory of the Tau installation).

If the workspace cannot be opened, an `APIError` exception will be thrown.

**Example 775**

This example uses Tau Access to attach to an instance of Tau running on the local machine. The workspace `C:\MyWorkspace.ttw` is then loaded in that Tau instance.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdWorkspace wksp = sa.openWorkspace("C:\\MyWorkspace.ttw");
```

### newWorkspace

```
ITtdWorkspace newWorkspace(String path) throws APIError;
```

Creates a new Tau workspace and associates it with a workspace file (`.ttw` file). If an existing workspace is open in the Tau IDE it will first be closed.

This method is the equivalent of creating a new blank workspace using the **File - New** menu.

`path` is the path where to save the workspace file. If it is a relative path, it will be interpreted as relative to the current working directory of Tau (which is typically the `bin` directory of the Tau installation).

If the workspace cannot be created or the workspace file cannot be saved in the specified location, an `APIError` exception will be thrown.

**Example 776**

This example uses Tau Access to attach to an instance of Tau running on the local machine. A new workspace is then created in that Tau instance.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdWorkspace wksp = sa.newWorkspace("C:\\temp\\NewWksp.ttw");
```

### openProject

```
ITtdProject openProject(String path) throws APIError;
```

Opens a Tau project file (`.ttp` file). The model associated with the project, if any, will be loaded.

This method is the equivalent of opening a project using the **File - Open** menu.

`path` is the path to the project to open. If it is a relative path, it will be interpreted as relative to the current working directory of Tau (which is typically the `bin` directory of the Tau installation). The string may contain URNs.

If the project cannot be opened, an `APIError` exception will be thrown.

**Example 777**

This example uses Tau Access to attach to an instance of Tau running on the local machine. The project `C:\MyWorkspace.ttp` is then loaded in that Tau instance.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdProject prj = sa.openProject("C:\\MyProject.ttp");
```

### getWorkspace

```
ITtdWorkspace getWorkspace();
```

Returns the workspace that is currently open in the Tau IDE. In case no workspace is open `null` is returned.

**Example 778**

This example uses Tau Access to attach to an instance of Tau running on the local machine. The path of the workspace currently open in that Tau instance is printed.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdWorkspace currentWksp = sa.getWorkspace();
if (currentWksp != null)
  System.out.println("Current workspace is " +
currentWksp.getPath());
```

### interpretTclScript

```
String interpretTclScript(String script) throws
APIError;
```

Interprets a Tcl script in Tau. Which Tcl commands that are available for use in the script depends on what is loaded in Tau. As a general rule all Tcl commands prefixed with `std` are always available, while those that are prefixed with `u2` only are available when a UML model is loaded.

The result of the script interpretation is returned as a string.

In case of Tcl script interpretation errors, an `APIError` exception will be thrown.

**Example 779**

This example uses Tau Access to attach to an instance of Tau running on the local machine. Tcl commands are executed to pop up a couple of message dialogs.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
String res = sa.interpretTclScript("std::MessageDialog -message
\"Cool, eh?\" -style yesno -icon question");
if (res.equals("6")) // Yes
  sa.interpretTclScript("std::MessageDialog -message \"I
agree!\"");
else if (res.equals("7")) // No
  sa.interpretTclScript("std::MessageDialog -message \"I
disagree!\"");
```

Refer to the Tcl API documentation for information about available Tcl commands.

### getApplicationName

```
String getApplicationName();
```

Returns the Tau application product name. Different Tau products support different features, and this method is thus a means for a client to know which Tau features it can utilize.

**Example 780**

This example uses Tau Access to attach to an instance of Tau running on the local machine. It then prints the name of the Tau application, as well as its PID, its version information and the name of the user running that Tau instance.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
System.out.println("Application name: " + sa.getApplicationName());
```

```
System.out.println("PID: " + sa.getApplicationPID());
System.out.println("Version: " + sa.getApplicationVersion());
System.out.println("User: " + sa.getApplicationUserName());
```

The output from running this example could for example be:

```
Application name: Telelogic Tau
PID: 147260
Version: 4.1
User: mmo
```

---

### getApplicationPID

```
String getApplicationName();
```

Returns the process id (PID) of the Tau instance. This method can for example be useful when there are multiple instances of Tau running on a machine, and a client wants to communicate with one particular of these instances. The PID returned by this method is then a means for distinguishing the different Tau instances.

See Example 780 on page 2331 for an example of using `getApplicationPID`.

### getApplicationVersion

```
String getApplicationVersion();
```

Returns the version number of the Tau instance as a string. Different Tau versions support different features, and this method is thus a means for a client to know which Tau features it can utilize.

See Example 780 on page 2331 for an example of using `getApplicationVersion`.

### getApplicationUserName

```
String getApplicationUserName();
```

Returns the name of the user (login name) who is running the Tau instance. This method can for example be useful when there are multiple instances of Tau running on a machine with more than one user logged onto it. Using this method a client can know if a certain Tau application instance runs under the same user as the client, or under some other user.

See Example 780 on page 2331 for an example of using `getApplicationUserName`.

## ITtdWorkspace

The `ITtdWorkspace` interface represents a Tau workspace. It contains methods which operate on that workspace.

**Note**

*The `ITtdWorkspace` interface is of no use in a non-interactive execution environment, since the Tau IDE is then not available. It can only be used when using Tau Access to access a running instance of Tau.*

### getPath

```
String getPath();
```

Returns the full path of the workspace file where the workspace is stored.

See Example 778 on page 2330 for an example of using `getPath`.

### getProject

```
ITtdProject getProject(String path);
```

Returns a project with the specified path which is contained in the workspace. If no matching project is found, `null` is returned.

**Note**

*When searching for a matching project, paths are not normalized, nor are contained URNs expanded. A project will only be found if its path matches the argument exactly.*

**Example 781**

This example uses Tau Access to attach to an instance of Tau running on the local machine. The current workspace in that Tau instance is searched for a project stored at `C:\temp\JAPI.ttp`. If such a project is found it is made the active project of the workspace. Finally it is verified that the found project now is the active project.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdWorkspace wksp = sa.getWorkspace();
ITtdProject prj = wksp.getProject("C:\\temp\\JAPI.ttp");
if (prj != null)
{
  wksp.setActiveProject(prj);
  if (wksp.getActiveProject() == prj)
    System.out.println(prj.getName() + " is now the active
project!");
```

```
}
```

### getActiveProject

```
ITtdProject getActiveProject();
```

Returns the currently active project of the workspace. If no project is active, `null` is returned. This can only happen in case of an empty workspace.

See Example 781 on page 2333 for an example of using `getActiveProject`.

### setActiveProject

```
void setActiveProject(ITtdProject project);
```

Sets a project as the active project of the workspace. There can be at most one active project in a workspace, so if another project was active previously, it will no longer be active after the call to `setActiveProject`.

See Example 781 on page 2333 for an example of using `setActiveProject`.

## ITtdProject

The `ITtdProject` interface represents a Tau project. It contains methods which operate on that project.

**Note**

*The `ITtdProject` interface is of no use in a non-interactive execution environment, since the Tau IDE is then not available. It can only be used when using Tau Access to access a running instance of Tau.*

### getPath

```
String getPath();
```

Returns the full path of the project file where the project is stored.

**Example 782**

This example uses Tau Access to attach to an instance of Tau running on the local machine. The active project in that Tau instance is found, and its name and path are printed.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
```

```
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdWorkspace wksp = sa.getWorkspace();
ITtdProject prj = wksp.getActiveProject();
if (prj != null)
{
  System.out.println("Active project is " + prj.getName());
  System.out.println("Stored at " + prj.getPath());
}
```

### getName

```
String getName();
```

Returns the name of the project. This is usually the name of the project file without path.

See Example 782 on page 2334 for an example of using `getName`.

### getModel

```
IUnknown getModel();
```

Returns the UML model of the project. If the project has no UML model, `null` is returned.

The interface type `IUnknown` is a common super interface for all interfaces of the Java API. It is used here instead of ITtdModel to accommodate for non-UML projects. It can be casted to ITtdModel.

**Example 783**

This example uses Tau Access to attach to an instance of Tau running on the local machine. The active project in that Tau instance is found, and its model is located. The names of the top-level definitions in that model are then printed.

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdWorkspace wksp = sa.getWorkspace();
ITtdProject prj = wksp.getActiveProject();
if (prj == null)
  return;

ITtdModel model = (ITtdModel) prj.getModel();
if (model == null)
  return;

List<ITtdEntity> defs = ((ITtdEntity)
model).getEntities("OwnedMember");
for (ITtdEntity d : defs)
```

```
System.out.println(d.getValue("Name", 0));
```

# 80

# *Tau Access*

This chapter describes Tau Access which is a run-time API for Tau. This API allows clients to programmatically attach to a running instance of Tau, whether that instance runs locally on the same machine as the client program, or remotely on another machine in the same network. Once a client has attached to Tau it can access its features through the standard C++ and Java APIs.

Intended readers are developers of client applications that want to integrate or interact with Tau in one way or another. A basic knowledge of either Java or C++ is assumed throughout this chapter.

**See also**

"Java API" on page 2285 in Chapter 79, *Java API*

"C++ API" on page 2245 in Chapter 78, *C++ API*

# Introduction

Tau Access provides an entry point to the Tau C++ and Java APIs from a client application external to Tau. Similar capabilities are provided by the COM API, but while the COM API is specific for the Windows platform, Tau Access works on both Windows and Unix platforms.

## Implementation Principle

The implementation of Tau Access makes use of the Tau Web Server and the Tcl API. When the client calls a method in the C++ or Java APIs, Tau Access translates this into one or many Tcl commands. These commands are then sent to the remote Tau instance through an HTTP request which will be processed by the web server of that Tau instance. The Tcl commands are synchronously processed in the main thread of the remote Tau instance. Finally, when execution is completed, the Tcl script result is passed back to Tau Access in the HTTP response. Tau Access then translates the response back to a piece of Java or C++ data which is made available to the client program.

Many API methods return objects identified by interface references. Such objects are represented by proxy objects in the client application. The following information is stored in each proxy object:

- The instance of Tau where the real object resides. The Tau instance is identified by the host name of the computer where it runs, and the port used by its web server.
- The remote address of the real object. This is represented by its Tcl identifier.
- Run-time type information about the object. This is an optimization to avoid having the client ask Tau for this information whenever the client needs to know about the object kind.

The Tau Access Java API is built on-top of the Tau Access C++ API through the use of JNI (Java Native Interface).

The picture below illustrates the run-time situation when a Java client uses Tau Access to get a reference to the currently loaded workspace of Tau. ‘wksp’ is a Java object, while ‘proxy’ and ‘object’ are C++ objects.

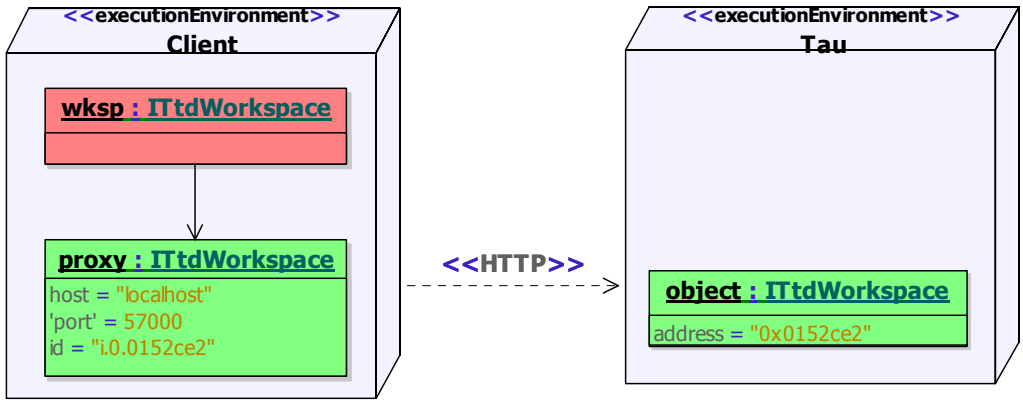



*Figure 289: Run-time objects when using Tau Access for accessing a workspace*

Calls of methods on ‘wksp’ are forwarded to ‘proxy’ and then finally, via HTTP, to ‘object’. Upon method return data flows in the opposite direction.

# Using Tau Access

A client program that wants to use Tau Access should perform the following steps (depending on whether Java or C++ is used):

### Java clients

1. Place the `tauaccess.jar` in the classpath. This JAR file can be found in the Tau installation at `/lib/Java`.
2. Set-up the environment variable `PATH` (`LD_LIBRARY_PATH` on Unix) so that it includes the Tau installation `bin` directory.

### C++ clients

1. Include the header file `TauAccess.h` found in the Tau installation at `include/ToolAPI`. You may also want to include `StudioAccessInterfaces.h` and `U2ModelAccess.h` depending on which parts of the C++ API you intend to use.
2. Link with the libraries `TauAccessU.lib` and `SBL10U.lib` found in the Tau installation at `lib/<platform>`, where `<platform>` depends on the target platform (Windows / Solaris / Linux). If you have included `StudioAccessInterfaces.h` you should also link with `StudioU.lib`, and if you have included `U2ModelAccess.h` you should link with `U2DLLU.lib`.

3. Set-up the environment variable `PATH` (`LD_LIBRARY_PATH` on Unix) so that it includes the Tau installation `bin` directory.
4. Perform the additional standard steps for using the C++ API as described in C++ API Set-up.

Note that the C++ API Debugging utilities are not available when using Tau Access.

A client using Tau Access is categorized as an interactive client, although it runs outside of the Tau executable. This is because the Tau IDE always is available in the Tau instance the client is connected to.

Like with the Tau Java API, Tau Access Java definitions are defined in the `com.telelogic.tau` package.

Like with the Tau C++ API, Tau Access C++ definitions are defined in the `u2` namespace.

## API Entry Point

The entry point of using Tau Access is the `ITtdTauAccess` interface. This interface is obtained in the following way:

**Java clients**

```
ITtdTauAccess tauAccess = TauAccess.getTauAccess();
```

**C++ clients**

```
u2::ITtdTauAccess* pTauAccess = u2::GetTauAccess();
```

## Object Lifetime Management

A C++ client of Tau Access must be explicit about when it no longer intends to use an interface it has obtained from the API. It does this by calling the function `DecRef()` on the interface when it no longer will use it. Internally Tau Access uses reference counting to know when a proxy object can be deleted from the memory of the client application. As a general rule the reference count for an object is incremented by the API function which returns a new interface to the client. It does this by calling the function `IncRef()` on the interface. The client is then responsible for calling `DecRef()` on the interface when it no longer needs to use it. It should also call `IncRef()` for every new reference it makes to the interface.

C++ clients that pass around interface pointers in non-trivial ways may consider using smart pointers for managing the reference count of the interface pointers. One smart pointer implementation that may be used is the `Ptr` template which is defined in the file `TauAPICommon.h`.

**Note**

*Usually when using the Tau C++ API it is not necessary to call the* `IncRef()` *and* `DefRef()` *functions to maintain a correct reference count on objects. This is because Tau then manages the lifetime of the underlying objects. However, when using Tau Access calls of these functions are mandatory to prevent memory leaks.*

A Java client of Tau Access does not need to bother about object lifetime management since Java has a garbage collector. Tau Access will automatically call `DecRef()` on an interface when it is finalized by the garbage collector. Hence both the Java and the C++ proxy objects are deleted automatically in this case.

## Interface Casting

Interface casting when using Tau Access works in the same way as in traditional use of the C++ or Java APIs. See Interface Casting (Java) and Interface casting (C++) for more information.

Note that Tau Access keeps run-time type information in the proxy objects to allow interface casting on the client side without requiring communication with Tau.

## Handling API Errors

Both the Tau Java and C++ APIs use an `APIError` exception for signalling errors that may occur when using the API. There is no difference when accessing these APIs through Tau Access. However, the client application should be prepared to handle `APIError` exceptions even for API methods which normally would never through this exception. The reason is that with Tau Access there are always error situations that can arise from underlying HTTP or network problems regardless of the usual API-level errors.

# Example

The example below shows how to use Tau Access for Java and C++ respectively, in order to attach to an instance of Tau running on the local machine (on port 57000). The active project in that Tau instance is found, and its model is located. The names of the top-level definitions in the model are then printed.

**Example 784**

### Java

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa =
ta.getTauApplicationAtNetworkLocation("localhost", 57000);
ITtdWorkspace wksp = sa.getWorkspace();
ITtdProject prj = wksp.getActiveProject();
if (prj == null)
  return;

ITtdModel model = (ITtdModel) prj.getModel();
if (model == null)
  return;

List<ITtdEntity> defs = ((ITtdEntity)
model).getEntities("OwnedMember");
for (ITtdEntity d : defs)
  System.out.println(d.getValue("Name", 0));
```

### C++

```
using namespace u2;

ITtdTauAccess* pTA = GetTauAccess();
Ptr<ITtdStudioAccess> pSA(pTA-
>GetTauApplicationAtNetworkLocation(_T("localhost"), 57000));
Ptr<ITtdWorkspace> pWksp(pSA->GetWorkspace());
Ptr<ITtdProject> pPrj(pWksp->GetActiveProject());
if (!pPrj)
  return;

Ptr<ITtdModel> pModel(cast<ITtdModel>(pPrj->GetModel()));
if (!pModel)
  return;

std::list<ITtdEntity*> lst;
Ptr<ITtdEntity> pModelEntity(cast<ITtdEntity>(pModel.get()));
pModelEntity->GetEntities(_T("OwnedMember"), lst);
for (std::list<ITtdEntity*>::const_iterator it = lst.begin(); it !=
lst.end(); it++)
{
  Ptr<ITtdEntity> pDef(*it);
  tstring strName;
  pDef->GetValue(_T("Name"), strName);
  std::wcout << strName.c_str() << std::endl;
}
```

Note the use of the `Ptr` smart pointer to avoid explicit calls to the `IncRef()` and `DecRef()` functions.

---

## Other API Differences

This section lists some further differences when using the Tau Java and C++ APIs through Tau Access, as compared to when using these APIs otherwise.

### No callback interfaces

Tau Access does not support callback interfaces, such as for example `ITtdMetaVisitCallback` or `ITtdMessageList`. This is a consequence of the underlying HTTP protocol where all requests are initiated by the client and processed by the server (Tau). It is not possible for the server (Tau) to initiate a request to be processed by the client, as would have been required to support callback interfaces.

The workaround to this limitation is to define an agent which runs inside Tau, and which can be called by the Tau Access client. The agent can then make use of the callback interface and propagate necessary information to the client afterwards.

See Agents for more information about agents.

### Main thread serialization

All requests received by Tau from Tau Access clients will be serialized into the main execution thread. Therefore, if the main thread is blocked (for example because Tau is busy with performing some time consuming blocking operation) all Tau Access client requests will be hanging, waiting for Tau to be ready to process them.

### Performance issues

Naturally the nature of sending API requests over HTTP between different applications imply some overhead, especially if the applications are running on different machines in a network. If performance becomes an issue for a Tau Access client one way to address the problem could be to refactor parts of the client implementation into an agent which is run inside Tau instead. Accessing the Tau API from an agent is significantly faster than doing the

same access through Tau Access. Note, however, that currently an agent cannot be implemented in Java. Instead C++ is usually the best implementation choice for such agents.

See Agents for more information about agents.

# API Interfaces and Methods

This section describes the API interfaces and methods which are specific to Tau Access, and which are not covered by the general API documentation. See API Interfaces and Methods (Java) and API Interfaces and Functions (C++) for the documentation of the other parts of these APIs.

## ITtdTauAccess

The `ITtdTauAccess` interface contains methods for accessing a running Tau application, or to start a new instance of Tau.

See API Entry Point to learn how to obtain the `ITtdTauAccess` interface from the client application.

### GetRunningTauApplications

**Java**

```
List<ITtdStudioAccess> getRunningTauApplications()
throws APIError;
```

**C++**

```
virtual void
GetRunningTauApplications(std::list<ITtdStudioAccess*>&
instances) throw (u2::APIError) = 0;
```

Obtains a list of all Tau applications that are running on the local machine. Each Tau application is represented by an `ITtdStudioAccess` interface. Tau applications older than version 4.1 will not be part of the list.

If an error occurs while accessing the Tau applications, an `APIError` exception will be thrown.

**Example 785**

This example prints the name and version of the Tau applications running on the local machine.

### Java

```
ITtdTauAccess ta = TauAccess.getTauAccess();
List<ITtdStudioAccess> apps = ta.getRunningTauApplications();
for (ITtdStudioAccess tau : apps)
{
  System.out.println(tau.getApplicationName() + " (" +
tau.getApplicationVersion() + ")");
}
```

### C++

```
using namespace u2;

ITtdTauAccess* pTA = GetTauAccess();
std::list<ITtdStudioAccess*> apps;
pTA->GetRunningTauApplications(apps);
for (std::list<ITtdStudioAccess*>::const_iterator it =
apps.begin(); it != apps.end(); it++)
{
  Ptr<ITtdStudioAccess> tau(*it, true);
  tstring strName, strVersion;
  tau->GetApplicationName(strName);
  tau->GetApplicationVersion(strVersion);
std::wcout << strName << _T(" (") << strVersion << _T(" )") <<
std::endl;
}
```

---

## StartNewTauApplication

### Java

```
ITtdStudioAccess startNewTauApplication() throws
APIError;
```

### C++

```
virtual ITtdStudioAccess* StartNewTauApplication() throw
(u2::APIError) = 0;
```

Starts a new instance of Tau on the local machine. An `ITtdStudioAccess` interface representing the launched Tau instance is returned.

In case multiple versions of Tau are installed on the machine, the version which matches the used Tau Access libraries will be launched. This means that a Tau Access client built against the libraries of one particular Tau version can only launch instances of that particular Tau version.

In case the launch fails, an `APIError` exception will be thrown.

**Example 786**

This example launches a new instance of Tau on the local machine. It then prints the PID of the launched Tau application.

**Java**

```
ITtdTauAccess ta = TauAccess.getTauAccess();
ITtdStudioAccess sa = ta.startNewTauApplication();
System.out.println(sa.getApplicationPID());
```

**C++**

```
using namespace u2;

ITtdTauAccess* pTA = GetTauAccess();
Ptr<ITtdStudioAccess> pSA(pTA->StartNewTauApplication());
tstring strPID;
pSA->GetApplicationPID(strPID);
std::wcout << strPID << std::endl;
```

### GetTauApplicationWithPID

**Java**

```
ITtdStudioAccess getTauApplicationWithPID(int pid);
```

**C++**

```
virtual ITtdStudioAccess* GetTauApplicationWithPID(PID
pid) = 0;
```

Returns an `ITtdStudioAccess` interface representing a Tau application with the specified process ID running on the local machine. If no such Tau application exists `null` is returned.

Only Tau applications with version 4.1 or later can be found by this method.

### GetTauApplicationAtNetworkLocation

**Java**

```
ITtdStudioAccess
getTauApplicationAtNetworkLocation(String host, int
port);
```

**C++**

```
virtual ITtdStudioAccess*
GetTauApplicationAtNetworkLocation(const tstring& host,
unsigned int port) = 0;
```

Returns an `ITtdStudioAccess` interface representing a Tau application running on the specified host, with a web server using the specified port. If no such Tau application exists, `null` is returned.

Only Tau applications with version 4.1 or later can be found by this method.

See Example 784 on page 2342 for an example of using this method.

# 81

# *XML Framework Library*

This chapter describes the TTDXMLFramework library, which is a framework for working with general XML documents in UML. The library contains a number of agents that are useful when importing XML to UML, or when generating XML from UML. These agents can be used as utilities when creating custom XML-based importers or exporters in Tau.

# Activating the XMLFramework Addin

The TTDXMLFramework library is loaded by activating an add-in. Follow these steps:

1. In the **Tools** menu, select **Customize...**.
2. Click the Add-Ins tab and check the `XMLFramework` add-in.
3. Click **OK**.

As a result you should now see the TTDXMLFramework package loaded as a library.

# Importing XML Documents

The agent `ParseXMLFromFile` can be used to import an XML document into a UML representation. The agent has the following parameters:

- `file : Charstring`
  XML file to import.
- `out model : ITtdEntity`
  Created UML representation of the XML document.
- `messages : ITtdMessageList[0..1]`
  Optional message list where messages (errors, warnings etc.) are reported. If this parameter is omitted messages will be printed to the Messages tab (if the agent executes in an interactive execution environment) or to `stdout` (if the agent is non-interactive).

The model context of this agent should be the model (ITtdModel) where the created UML representation of the XML document should be inserted.

**Example 787: Importing an XML document**

This Tcl script imports an XML file `x.xml`:

```
set curProject [std::GetActiveProject]

set model [std::GetModels -kind U2 -project $curProject]

set agent [u2::FindByGuid $model
"@TTDXMLFramework@ParseXMLFromFile"]

set p [lappend p "x.xml" 0]
```

```
u2::InvokeAgent $model $agent $model p
```

The model representation of the XML document consists of a package which contains one single attribute ‘contents’. The default value of this attribute is a list of expressions representing the top-level entities in the XML document. These top-level expressions then in turn contain other expressions representing nested XML entities.

**Example 788: Unparsing the UML representation of an XML document**

Add the following lines to the script in Example 787 on page 2350:

```
set pkg [lindex $p 1]
output "[u2::Unparse $pkg]\n"
```

Assuming the file x.xml looks like this:

```
<HTML>
<HEAD><TITLE>The Title</TITLE>
</HEAD>
<BODY>
<H3>A Simple First Page</H3>
</BODY>
</HTML>
```

The following printout is obtained in the Script tab:

```
package '' {
    XML::Entity contents = {HTML ("\n", HEAD (.TITLE ("The
Title".), "\n".), "\n", BODY ("\n", H3 ("A Simple First
Page".), "\n".), "\n".)};
}
```

For more details about how different XML constructs are represented in UML see UML Representation of XML.

# Exporting XML Documents

The agent `WriteXMLToFile` can be used to export a UML representation of XML into an XML file. The agent has the following parameters:

- `file : Charstring`
  Name of XML file to write to.
- `messages : ITtdMessageList[0..1]`
  Optional message list where messages (errors, warnings etc.) are reported. If this parameter is omitted messages will be printed to the Messages tab (if the agent executes in an interactive execution environment) or to `stdout` (if the agent is non-interactive).

The model context of this agent should be a UML representation of the XML document to write to the file. It should be a top-level instance expression.

**Example 789: Exporting XML from a UML representation**

Add the following lines to the script in Example 787 on page 2350:

```
set pkg [lindex $p 1]
set a [u2::GetEntity $pkg "OwnedMember"]
set l [u2::GetEntity $a "DefaultValue"]
set instance [u2::GetEntity $l "Expression"]
set agent [u2::FindByGuid $model
"@TTDXMLFramework@WriteXMLToFile"]
set p2 [lappend p2 "y.xml"]
u2::InvokeAgent $model $agent $i p2
```

The generated file y.xml should now be identical to the original file x.xml, except for the use of insignificant whitespace characters which may differ.

# UML Representation of XML

An XML document is represented in UML using expressions. The top-level XML entities are represented with a list expression, with one contained expression for each top-level XML entity.

Supported XML constructs, and their representation as expressions in UML, are described below.

## Tag

An XML tag is represented by means of a UML instance expression. The name of the tag corresponds to the name of the UML class for the instance expression.

The nesting of tags inside other tags corresponds to the nesting of UML instance expressions.

**Example 790: Representation of XML tags**

**XML:**

```
<HTML>
<HEAD>
</HEAD>
</HTML>
```

**UML:**

```
HTML (. HEAD (. .) .)
```

## Attribute

An XML attribute is represented by an assignment, i.e. a binary expression with '=' as operator. The left hand side of the assignment is an identifier corresponding to the attribute name, and the right hand side is a character string value corresponding to the value of the attribute.

The assignment is inserted in the instance expression corresponding to the container tag.

**Example 791: Representation of XML attributes**

**XML:**

```
<A HREF="foo" onClick="testSub"></A>
```

**UML:**

```
A (. HREF = "foo", onClick = "testSub" .)
```

## Text Node

An XML text node is represented by a character string value which contains the text of the text node.

The string value is inserted in the instance expression corresponding to the container tag.

**Example 792: Representation of XML text nodes**

**XML:**

```
<A HREF="foo">LinkText</A>
```

**UML:**

```
A (. HREF = "foo", "LinkText" .)
```

## Processing Instruction

Not supported. An initial processing instruction

```
<?xml version="1.0" encoding="UTF-8"?>
```

will currently always be added when generating XML documents.

## Comment

An XML comment is represented by a parenthesis expression where the contained expression is a character string value which contains the text of the comment node.

The parenthesis expression is inserted in the instance expression corresponding to the container tag.

**Example 793: Representation of XML comments**

**XML:**

```
<HTML>
<!--
```

```
This is a comment!
-->
</HTML>
```

**UML:**

```
HTML (. ("This is a comment") .)
```

# 82

# *Tau Web Server*

Telelogic Tau contains a web server which enables HTTP clients, such as web browsers, to access tool functionality. This chapter describes the URLs which the Tau Web Server recognizes, and how they can be used from web pages in different ways.

# Purpose of the Tau Web Server

The Tau Web Server serves the following main purposes.

- It provides a means for accessing Tau (and the UML model hosted by Tau) remotely over a network. Although this is also possible on Windows using the COM API (DCOM), it is a more standard, platform-neutral way to use HTTP.
- It makes it possible to use HTML pages as the GUI of a Tau add-in. The HTML pages can be opened inside Tau using the std::HtmlReport Tcl command.
- It makes it possible to integrate Tau with other tools, using HTTP as the communication protocol to interchange information between the tools.

# Configuring the Tau Web Server

The Tau Web Server is contained in the main Tau executable called `vcs.exe`. When this application is launched the web server is automatically started, and set-up to listen for incoming HTTP requests. By default the web server will try to use the TCP/IP port 57000, but if there is already another instance of Tau running on the machine it will increment the port number by one until it finds an available port to use. If no available port is found an error message will be printed in the Message tab.

**To change the port number used by the Tau Web Server do the following:**

1. Open the **Option** dialog box (**Tools - Options...**)
2. Make sure the **"Show advanced option page"** checkbox is checked.
3. In the **Advanced** option page select the Studio server, and browse to **Studio - Settings - WebServer**.
4. The options **PortRangeBegin** and **PortRangeEnd** define the range of TCP/IP ports which the Tau Web Server will use. Make sure that these ports are available on your machine, and that the range is big enough to accomodate the maximum number of Tau instances that will be running simultaneously on the machine.

It is also possible to specify which port to use when starting Tau. This is done by means of the `-port` command line option. For example:

```
vcs.exe -port 57123
```

If the specified port is not available (typically because another Tau instance is using it) a warning message will be printed, and the usual procedure for finding an available port takes place.

# How to Use the Tau Web Server

When Tau is running you can check that the web server is properly configured and ready to be used by entering the following URL in a web browser:

```
http://localhost:57000/
```

You should see the main web page of the Tau Web Server that confirms that the web server is running. It will also list which Web Request Handlers that are currently registerred.

If you have changed which port number to use, or have more than one instance of Tau running you should use another port number. If you are unsure about which port the web server is using, you can find out by following these steps:

1. Open a new temporary Tcl file (**File - New... - File - Tcl file**)
2. Type the command `std::Output "[std::GetWebServerPort]"`
3. Execute the command (**Tools - Execute Script**). The web server port will be printed in the **Script** tab.

In the examples below we assume that the Tau web server uses port 57000.

## URL Syntax

The general syntax of a URL for the Tau Web Server is

`http://localhost:<port>/<handler>/<data>?<parameters>`

Here, `<port>` is the TCP/IP port used by the web server, `<handler>` is the name of a web request handler, and `<data>` is a string of data intended for that request handler. `<parameters>` are ordinary HTTP request parameters, that is a list of name-value pairs separated by ampersands (&).

See Web Request Handlers for more information about the different web request handlers that are available, and what they can be used for.

Naturally, you can access the web server of an instance of Tau that is running on a remote machine by replacing “localhost” with the name or IP number of that machine.

### Delaying Web Requests

By default the list of name-value pairs in `<parameters>` is made available to the selected web request handler. However, there is one special parameter which is interpreted by the web server itself:

```
_invocationDelay=<delay>
```

If this parameter is used the web server will delay the processing of that particular web request with the specified number of milliseconds. For example, a `<delay>` of 1000 will cause the request to be delayed with 1 second. Delaying web requests is useful when accessing the web server asynchronously (for example from a web page using AJAX). See Example 799 on page 2365.

## Web Request Handlers

The main web page of the Tau Web Server shows a table of web request handlers which are currently registerred. Each handler has a name and a description about the URL syntax it supports. Depending on the state of Tau, different request handlers may be available.

The available web request handlers are described below.

### File

The ‘file’ web request handler makes it possible to retrieve data stored in a file on the file system. The data is typically HTML, or some other form of XML, but in general it can be any textual data.

The URL syntax for the ‘file’ web request handler is:

```
file/<filename>
```

`<filename>` is a path to the file that contains the data to retrieve. The path may contain Tau URNs.

**Example 794: Using the ‘file’ web request handler**

To open an HTML file `index.html`, located relative to the Tau user-addins directory you may use an URL similar to the following:

```
http://localhost:57000/file/urn:u2useraddins:MyAddin/etc/index.html
```

The ‘file’ web request handler is always available.

### Agent

The ‘agent’ web request handler makes it possible to invoke an agent in Tau. The agent will be invoked in the model of the currently active project.

The URL syntax for the ‘agent’ web request handler is:

```
agent/<agent id>(<agent parameters>)
```

`<agent id>` identifies which agent to invoke. It may either be the GUID of the agent, or its fully qualified name. `<agent parameters>` are actual parameters that shall be passed when invoking the agent. Note that the text after the ‘/’ must be a valid U2 call expression. This means that U2 identifiers in this part of the URL may need to be enclosed in single quotes in order to comply with U2 syntax rules.

You may use any U2 expression as an actual argument to the invoked agent, provided it is a supported agent parameter expression (see Agent Parameters). In addition you may refer to HTTP request parameters by preceeding the name of the parameter with a ‘$’ sign. There are also some special parameters which can be used in order to pass other information from the HTTP request to the invoked agent. These special parameters are on the form

```
$context::<variable>
```

where <variable> is one of the following:

| Variable | Description |
|---|---|
| `response` | This is an out parameter which represents the response of the web request. The agent may assign a string to this parameter, which will be used as the resulting response text that is returned to the web client. |
| `user_agent` | Expands to the value of the HTTP server variable HTTP_USER_AGENT. |
| `method` | Expands to the value of the HTTP server variable REQUEST_METHOD. |
| `accept` | Expands to the value of the HTTP server variable HTTP_ACCEPT. |

| | |
|---|---|
| `accept_encoding` | Expands to the value of the HTTP server variable HTTP_ACCEPT_ENCODING. |
| `status` | This is an out parameter which represents the HTTP status code of the web request. The agent may assign an integer to this parameter, which should be a valid HTTP status code. For example, the status code 404 is used to denote the error "Not Found".<br>The default status code is 200, which means that the HTTP request was successful. |
| `content_type` | This is an out parameter which represents the HTTP server variable HTTP_CONTENT_TYPE. The agent should set a content type string to this parameter to indicate how the 'response' data should be treated by the web client. For example, if the response data is a HTML string the content type "text/html" should be used.<br>The default content type is "text/plain". |

**Example 795: Using the 'agent' web request handler**

This URL will invoke an agent with the GUID '@MyAgent'. Note that the GUID must be enclosed in single quotes since it contains the '@' character.

```
http://localhost:57000/agent/'@MyAgent'()
```

This URL will invoke an agent A located in a package P. The agent gets one string parameter as input which is the 'response' parameter. It can use it for returning for example some HTML data to the web client.

```
http://localhost:57000/agent/::P::A('$context::response')
```

This URL will invoke an agent with the GUID '@MyAgent'. The agent receives three actual parameters; the integer value 14, the boolean literal 'false' and finally the string value of the HTTP request parameter called '$par', which is "SomeValue".

```
http://localhost:57000/agent/'@MyGuid'(14, false,
'$par')?par=SomeValue
```

The ‘agent’ web request handler is available when there is at least one project with a UML model loaded in Tau.

### Tcl

The ‘tcl’ web request handler makes it possible to interpret a Tcl script in Tau.

The URL syntax for the ‘tcl’ web request handler is:

```
tcl/<script>
```

`<script>` is the Tcl script to interpret. The result of interpreting the script will be set as the response text of the HTTP request.

**Example 796: Using the ‘tcl’ web request handler**

This URL will interpret a Tcl script for getting the version of Tau:

```
http://localhost:57000/tcl/std::GetApplicationVersion
```

The ‘tcl’ web request handler is always available. However, keep in mind that not all Tcl commands are available at all times. In general, commands prefixed with ‘std::’ are always available, while commands prefixed with ‘u2::’ only are available when at least one project with a UML model is loaded in Tau.

### Variable

The ‘variable’ web request handler is a simple mechanism for storing data in the web server which will be persistent across multiple HTTP requests. By using such variables the behaviour of one HTTP request can depend not only on its current input data, but also on the previously made HTTP requests.

The URL syntax for setting the value of a web server variable is:

```
variable/set <variable name> <variable value>
```

If `<variable value>` is an empty string the variable will be unset.

The URL syntax for getting the value of a web server variable is:

```
variable/get <variable name>
```

The response text will be set to the value of the variable. If `<variable name>` is an empty string the response text will be the list of all defined variables with their current values.

**Example 797: Using the 'variable' web request handler**

This URL will set the variable 'logged_in' to the value "true":

```
http://localhost:57000/variable/set logged_in true
```

Afterwards we can get the value of the variable with this URL:

```
http://localhost:57000/variable/get logged_in
```

The response will be `true`.

We can also get the values of all variables with this URL:

```
http://localhost:57000/variable/get
```

The response will be `logged_in=true`.

Finally we can unset the variable using this URL:

```
http://localhost:57000/variable/set logged_in
```

---

The 'variable' web request handler is always available.

# Examples

Here we present some examples on using the Tau Web Server.

**Example 798: An agent generating an HTML page**

Assume we have defined an agent 'GeneratePage' in Tau (placed in global scope) which is implemented by the following Tcl script:

```
proc GeneratePage { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  set p "<html><head><h1>Page generated by
  agent!</h1></head></html>"
  set ap [lreplace $ap 0 0 $p ]
}
```

Then let us create an HTML file 'MyPage.html' with a simple form:

```
<html>
```

```
<head>
<body>
<form action="/agent/::GeneratePage('$context::response')"
method="get">
<input type="submit" class="button" name="pressme"
value="Press Me"/>
</form>
</body>
</html>
```

Note that we can use a relative 'action' URL if we open the HTML file by typing the following URL in a web browser:

```
http://localhost:57000/file/C:\MyPages\MyPage.html
```

---

**Example 799: Using delayed web requests from an AJAX web page**

By using delayed web requests from web pages using asynchroneous JavaScript (AJAX) it is possible to dynamically update an HTML page based on changes that takes place in Tau. Let's start by defining an agent 'GetSelection' which returns the list of entities currently selected in the Tau Model View browser:

```
proc GetSelection { triggeredBy timing context server
agentParameters } {
  upvar 1 $agentParameters ap
  set response [lindex $ap 0]
  set sel [std::GetSelection]
  set c 0
  foreach s $sel {
    if {$c != 0} {
      set response "$response, "
    }
    incr c
    if {[std::GetKind $s] != "U2" || $s == 0} {
      set r "(Non-U2 entity)"
    } else {
      set r [u2::GetMetaClassName $s]
    }
    set response "$response$r"
  }
  if {$response == ""} {
    set response "Nothing is selected!"
  }
  set ap [lreplace $ap 0 0 $response ]
}
```

As you can see, this agent formats a string with information about which kind of entities that are currently selected. Note that the response string is not an HTML string. It's not even XML, but a plain text string.

Now let's define an HTML page 'GetSelection.html' with some JavaScript which invokes the 'GetSelection' agent at regular intervals (1 second) and prints the result string to the page.

```
<head>
<title>Ajax Selection Observer Agent</title>
<body>
<h1>Ajax Selection Observer</h1>
<script type="text/javascript">
/* Create a new XMLHttpRequest object to talk to the Web
server */
var xmlHttp = false;
/*@cc_on @*/
/*@if (@_jscript_version >= 5)
try {
  xmlHttp = new ActiveXObject("Msxml2.XMLHTTP");
} catch (e) {
  try {
    xmlHttp = new ActiveXObject("Microsoft.XMLHTTP");
  } catch (e2) {
  xmlHttp = false;
  }
}
@end @*/
if (!xmlHttp && typeof XMLHttpRequest != 'undefined') {
  xmlHttp = new XMLHttpRequest();
}
function callServer() {
  // Build the URL to connect to
  var url =
"/agent/::GetSelection('$context::response')?_invocationDel
ay=1000";
  // Open a connection to the server
  xmlHttp.open("GET", url, true);
  // Setup a function for the server to run when it's done
  xmlHttp.onreadystatechange = updatePage;
  // Send the request
  xmlHttp.send(null);
}
function updatePage() {
  if (xmlHttp.readyState == 4) {
    var response = xmlHttp.responseText;
    var resNode = document.getElementById("sel");
    if (resNode != null)
      resNode.innerHTML = response;
    callServer();
  }
}
callServer();
</script>
Selected elements in Tau:<b><p id="sel"></p></b>
</body>
</html>
```

When this page is loaded the JavaScript function ‘callServer’ is called. It builds the URL for invoking the GetSelection agent and sets up the ‘updatePage’ function to be called when the agent has been invoked. This happens after 1 second, because we use the `_invocationDelay` parameter with a value of 1000 milliseconds. When we obtain the result we locate the DOM element with ID ‘sel’ and sets its inner HTML to the response string returned by the agent. Then we immediately call ‘callServer’ again, in order to schedule a new request. As an effect, when the selection is changed in Tau the web page will be updated with information about the current selection within 1 second.

Note that since the used URL is relative we must open the page using an absolute URL like this:

```
http://localhost:57000/file/C:\MyPages\GetSelection.html
```

# Limitations

The Tau Web Server has the following known limitations.

### POST protocol not supported

The web server only supports the HTTP GET protocol.

# *Common Reference*

The reference chapters listed in this section describe functionality that is valid for all types of Tau projects.

# 84

# *Useful Shortcut Keys*

This section lists useful shortcut keys that you can use. Access keys can be used in the same way as other standard applications.

**(UNIX) This only applies to Exceed users**: If you use a non-American keyboard, you must map the ALT button correctly in order to use the access keys.

**To map the ALT button:**

1. Click the **Start** button, point to **Programs**, point to **Exceed** and click **xconfig**.
2. In the dialog that opens, double-click **Input**. The Input dialog opens.
3. In the **Alt key** field, select **To X** or **Right To Window, Left To X**.
4. Close the dialog.

**Note**

*UNIX: Some short-cut sequences (such as ALT-X, CTRL-H) may be intercepted by the X11 window manager and can cause other actions to be taken than those described in this manual. In most cases it is possible to configure the X11 window manager not to intercept specific short-cut sequences. Please refer to the documentation on your window manager for further information regarding short-cuts.*

# Workspace Operations

| Keyboard shortcut | Description |
|---|---|
| CTRL + N<br>Then CTRL + TAB to Workspaces tab | Create a new workspace |
| CTRL + O | Open an existing workspace. |
| MINUS SIGN (-) on the numeric keypad | Contracts the tree of a selected entity. |
| MULTIPLICATION SIGN (*) on the numeric keypad | Expands the model tree one level below the selection. Can be used repeatedly to expand deeper. |
| PLUS SIGN (+) on the numeric keypad | Expands the selection. |
| ALT + 4 | Reconfigure Model View, selection of model filter |

# Project Operations

| Keyboard shortcut | Description |
|---|---|
| CTRL + N<br>Then CTRL + TAB to Project tab | Create a new project |
| CTRL + O | Open project. |

# File Operations

| Keyboard shortcut | Description |
|---|---|
| CTRL + N | Create a new file |
| CTRL + O | Open a file |
| CTRL + P | Print the active document |
| CTRL + S | Save active document |

# Navigate in Files

| Keyboard shortcut | Description |
|---|---|
| CTRL + DOWN ARROW | Scroll down a few rows at a time, without moving the insertion point |
| CTRL + END | Move insertion point to end of file |
| CTRL + SHIFT + G | Opens the **Go to line number** dialog |
| CTRL + HOME | Move insertion point to beginning of file |
| CTRL + LEFT ARROW | Step left one word at a time |
| CTRL + M | Open Navigator tab in Output window |
| CTRL + RIGHT ARROW | Step right one word at a time |
| CTRL + UP ARROW | Scroll up a few rows at a time, without moving the insertion point |
| END | Move insertion point to end of line |
| HOME | Move insertion point to beginning of line |

# Highlight Text

| Keyboard shortcut | Description |
|---|---|
| CTRL + SHIFT + END | Highlight text to the end of the file |
| CTRL + SHIFT + HOME | Highlight text to the beginning of the file |
| CTRL + SHIFT + LEFT ARROW | Highlight one word at a time to the left |
| CTRL + SHIFT + RIGHT ARROW | Highlight one word at a time to the right |
| SHIFT + DOWN ARROW | Highlight one row downwards |

| Keyboard shortcut | Description |
|---|---|
| SHIFT + END | Highlight to the end of the line |
| SHIFT + HOME | Highlight to the beginning of the line |
| SHIFT + LEFT ARROW | Highlight one character at a time to the left |
| SHIFT + RIGHT ARROW | Highlight one character at a time to the right |
| SHIFT + UP ARROW | Highlight one row upwards |

# Edit Text

| Keyboard shortcut | Description |
|---|---|
| CTRL + A | Select all |
| CTRL + C | Copy |
| CTRL + F | Find in active file |
| CTRL + H | Replace |
| CTRL + SPACEBAR<br>SHIFT + SPACEBAR | Name completion, if a definition is found that matches the current name up to the cursor position. If there are multiple matches a Name completion scroll menu will open. |
| CTRL + V | Paste |
| CTRL + X | Cut |
| CTRL + Y | Redo |
| CTRL + Z | Undo |
| F1 | Help with textual syntax on current selection. |
| SHIFT + F8 | Restores text from model, discarding comments or user added formatting. |

| Keyboard shortcut | Description |
|---|---|
| SHIFT + arrow keys | Extends the current text selection. Requires that text is selected |
| SHIFT + END | Selects text from cursor position to end of text row. |
| SHIFT + HOME | Selects text from start of text row to cursor position. |

# Editor Shortcuts

| Keyboard shortcut | Description |
|---|---|
| Arrow key | Selects the symbol in the direction of the arrow, requires current selection |
| CTRL + <click when placing symbols in diagram> | Allows you to place a number of symbols of the same type. Requires that you first select a symbol from the symbol toolbar. |
| CTRL + <click when placing symbols in state machine flow> | Allows you to insert a symbol in the flow. Requires that you first select the preceding symbol or flowline in the flow. |
| CTRL + <click word in diagram> | Navigates to definition. If no diagrams contain the definition, the Model Navigator opens. |
| CTRL + <double-click with a symbol selected in state machine flow> | Selects the entire flow from the selected symbol and downward. Selection will be done on branched flows (multiple signals, decision etc.). |
| CTRL + <Rotate the wheel button> | Scroll the diagram horizontally (requires an IntelliMouse pointing device) |
| CTRL + ALT + END | Diagram navigation, go down in diagram scope |
| CTRL + ALT + Page Down<br>CTRL + ALT + TAB | Diagram navigation, navigate to next diagram in diagram scope |
| CTRL + Arrow key | Moves the selected symbol 5 grid steps in the direction of the arrow. |

| Keyboard shortcut | Description |
|---|---|
| CTRL + DELETE | Delete from Model, deletes the presentation element and its corresponding model element. If other presentation elements are connected to this model element, they will also be deleted. |
| CTRL + DIVISION SIGN (/) on the numeric keypad | Hide all operations. (Valid for signature symbols: class, timer, signal, interface, operation, state machine, datatype, enumeration...) |
| CTRL + F3 | Jumps to the next presentation element of the same model element |
| CTRL + SHIFT + F3 | Jumps to the previous presentation element of the same model element |
| CTRL + MINUS SIGN (-) on the numeric keypad | Hide all attributes, parameters. (Valid for signature symbols: class, timer, signal, interface, operation, state machine, datatype, enumeration...) |
| CTRL + MULTIPLICATION SIGN (*) on the numeric keypad | Show operations. (Valid for signature symbols: class, timer, signal, interface, operation, state machine, datatype, enumeration...) |
| CTRL + PLUS SIGN (+) on the numeric keypad | Show attributes, parameters. (Valid for signature symbols: class, timer, signal, interface, operation, state machine, datatype, enumeration...) |
| CTRL + SHIFT + <click symbol in toolbar> | Interaction overview and Activity diagram: Append a symbol and toggle orientation. Symbol position will be in the currently not selected orientation. (Append requires that a symbol is selected.) |
| CTRL + SHIFT + Arrow key | Moves the selected symbol 1 grid step in the direction of the arrow. |
| CTRL + SHIFT + M | Open Create Presentation dialog |
| CTRL + TAB | Switches to the next open diagrams |
| CTRL + U | Update Model (requires Active Modeler add-in) |
| CTRL+ALT + HOME | Diagram navigation, go up in diagram scope |

| Keyboard shortcut | Description |
|---|---|
| CTRL + ALT + Page Up<br>CTRL + ALT + SHIFT + TAB | Diagram navigation, navigate to previous diagram in diagram scope |
| ESC, DELETE<br><Right-click canvas> | Aborts line creation |
| F2 | Enters the edit mode on a selected symbol |
| F4 | Moves to the next selection in the Output window |
| SHIFT + <click symbol in toolbar> | Create and append a symbol in the diagram. Symbols that cannot be auto-created appear dimmed. (Append requires that a symbol is selected.) |
| SHIFT + Arrow key | Selects the symbol in the direction of the arrow and adds it to the selection. (requires current selection) |
| SHIFT + F4 | Moves to the previous selection in the Output window |
| ALT + UP ARROW | Moves a selected node up in the Model View. |
| ALT + DOWN ARROW | Moves a selected node down in the Model View. |
| SHIFT + ENTER | Shows the model element of the selected diagram element in the Model View. |
| F8 | Check the current selection. |
| CTRL + F8 | Do a check of the entire model. |
| SHIFT + SPACEBAR | Auto creation. All elements that can be auto created on the current selection will be displayed. See Auto placement. |
| CTRL + SPACEBAR | Auto insertion. All elements that can be auto inserted after the current selection will be displayed. See Auto placement. |
| CTRL + R | Route selected lines and assign new endpoints. |

# Compare and Merge

| Keyboard shortcut | Description |
|---|---|
| ALT + EQUAL SIGN (=) | Compare selection |
| ALT + PLUS SIGN (+) on the numeric keypad | Merge selection |
| CTRL + ALT + LEFT ARROW | Select version 1 only on selected differences. |
| CTRL + ALT + RIGHT ARROW | Select version 2 only on selected differences. |
| CTRL + ALT + UP ARROW | Select ancestor version only on selected differences. |
| CTRL + ALT + DOWN | Select both versions on selected differences. |

# Application Builder Shortcuts

| Keyboard shortcut | Description |
|---|---|
| CTRL + SCROLL LOCK | Stops the build process |
| F5 | Launch the current configuration |
| SHIFT + F7 | Generate current configuration |
| F7 | Build the current configuration |
| SHIFT + F5 | Stops the execution |
| CTRL + F7 | Update configuration |

# Model Verifier Shortcuts

| Keyboard shortcut | Description |
|---|---|
| ALT + PAUSE | Break |
| ALT + F10 | Step local |
| CTRL + SHIFT + F5 | Restart the debug session from the beginning |
| F5 | Go |
| F9 | Inserts/removes a selected breakpoint |
| F10 | Step over |
| F11 | Step into |
| SHIFT + F10 | Next transition |
| SHIFT + F11 | Step out |

# Window Navigation

| Keyboard shortcut | Description |
|---|---|
| ALT + 1 | Toggle full screen mode |
| CTRL + F2 | Toggles definition at cursor position as Bookmark in the Model Navigator |
| CTRL + F4 | Close the active window |
| CTRL + SHIFT + TAB<br>CTRL + SHIFT + F6 | Navigate to the previous window |
| CTRL + TAB<br>CTRL + F6 | Navigate to the next window |
| SHIFT + F2 | Displays Model Navigator with context of definition at cursor position. |

# Properties editor

| Keyboard shortcut | Description |
|---|---|
| ALT + ENTER | Display Properties editor |
| CTRL + BACKSPACE | Go to owner, change scope in model tree to the owner of the current selection |
| CTRL + ALT + C | Switch to Control view |
| CTRL + ALT + T | Switch to Text view |

# Show/Hide Windows and Dialogs

| Keyboard shortcut | Description |
|---|---|
| ALT + 0 | Show/ hide workspace window |
| ALT + 2 | Show/ hide Output window |
| ALT + ENTER | Display Properties editor |
| CTRL + Q | Open Query dialog on selection |
| F1 | Display Help |

# Zoom/Pan

| Keyboard shortcut | Description |
|---|---|
| <Rotate the wheel button> | Scroll the diagram vertically (requires an IntelliMouse pointing device) |
| <Double-click middle mouse button> | Zoom to 100% |
| SHIFT + <double-click middle mouse button> | Zoom to fit editor window |
| SHIFT + <rotate wheel button> | Zoom in or zoom out depending on the rotate direction.The zoom in point will be where the mouse pointer is located |

| Keyboard shortcut | Description |
|---|---|
| CTRL + SHIFT + <Rotate the wheel button> | When a single line is selected the diagram will be scrolled along the line until one of the endpoints are centered in view (requires an IntelliMouse pointing device) |
| MINUS SIGN (-) on the numeric keypad | Zoom out 25% (This works when a diagram is active and not in text edit mode for any element) |
| PLUS SIGN (+) on the numeric keypad | Zoom in 25% (This works when a diagram is active and not in text edit mode for any element) |
| LESS-THAN SIGN (<) | When a single line is selected the diagram will be scrolled to the source endpoint of the line. |
| GREATER-THAN SIGN (>) | When a single line is selected the diagram will be scrolled to the destination endpoint of the line. |

# 85

# *Setting Up the Tool Environment*

This section mainly provides information how to integrate Tau with different tools.

Programming tools import:

- Import from UML
- Import from SDL
- Import from XMI

Configuration management tools:

- Telelogic Synergy
- IBM Rational ClearCase

# Import Wizard

Tau supports a number of different possibilities to import model data from other formats than the proprietary Tau model information.

The import scheme is started from the **Import** command in the **File** menu. This will launch the Import Wizard. The basic steps for an import is:

- Select the origin (C/C++, SDL, XMI)
- Add the source files to import from

The import will result in a new package in the current model containing elements and diagrams (when applicable) based on the source file information.

**See also**

"C/C++ Import" on page 545 in Chapter 15, *C/C++ Import*

"XMI import" on page 744 in Chapter 20, *UML 1.x Import*

"SDL Import" on page 629 in Chapter 17, *SDL Import*

# Configuration Management

Tau supports integration schemes with configuration management tools.

- A tight Integration with Telelogic Synergy that provides many of the features from this tool directly from the Tau user interface.
- An integration scheme based on the Microsoft Source Control Integration Interface. This means that as long as the source control system that you use support the Microsoft Source Control Integration Interface it should work with Tau. There is currently support for Integration with IBM Rational ClearCase which is regularly verified to work using Microsoft Source Control Integration Interface.
- A utility to start Tau Compare and Merge operations on u2 file versions selected directly in the configuration management tool.

### Source control provider

The General tab of **Tools->Options** contains the **Source control provider** drop down that enables the user to select source control scheme. Tau must be restarted for this to take effect.

**See also**

"Source control information" on page 2385

"Multiple configuration management tools" on page 2398

"Source control commands" on page 2399

## Source control information

The status of the files in the configuration management tool is displayed by different icons in the File View. The following icons apply:

- **blue check mark**

  This icon indicates that the file or folder is checked out.
- **orange check mark**

  This icon is used for folders that are partially checked out. It indicates that in the folder there are files that are checked in and files that are checked out or that the folder contains files that are not added to source control.

- **red x**

  This icon indicates that the file or folder is checked in.
- **no icon**

  If no icon is available the file or folder is not added to the configuration management tool data base.

The status can also be viewed in the property page of a file.

# Telelogic Synergy Integration

## Integration with Telelogic Synergy

This section describes how to integrate Synergy with Tau. For further details see the Synergy user documentation.

When using the Synergy integration two extra tool bars, dealing with Synergy Tasks and Objects, and one extra menu (called **Synergy**) are available. The tool bars and menu provide access to Synergy functionality from inside the Tau user interface using the commands listed below.

The integration also offers a set of predefined Tau file type definitions in Synergy.

**Project handling**

- Open Managed Project
- Migrate Project
- Project History
- Project Properties
- Project Merge
- Synchronize Project
- Update Project

**Task handling**

- Create Task
- Set Task
- Complete Task
- Task Properties

- Current Task Box

**Object handling**

- Create Object
- Object Properties
- Object History
- Check Out Object
- Check In Object
- Undo Check Out Object

**Version handling**

- Refresh Status

## Tau file type definitions

Two predefined file definitions are available. They define the Tau project and model file types in Synergy. The two files

- tau_project.xml
- tau_model.xml

in the directory “integrations/SYNERGYCM” contain the definitions for Tau project files (.ttp) and Tau model files (.u2).

The definitions should be applied by the Synergy type manager to the relevant data bases.

Further description on how to apply the type definitions can be found in the Synergy help on the `typedef` command.

## Install Synergy integration

### Windows

**Note**

*It is necessary to install the Synergy integration via a separate installer before it is possible to activate the integration.*

The source control system that will be used is specified in a system registry. When Synergy is installed, this source control system registry key should be set automatically. However, if you have more than one configuration management tool installed locally, you may have to edit this value manually. The only additional steps that are needed before starting to use Synergy are the following:

1. Start Tau.
2. On the **Tools** menu, click **Options**.
3. In the **General** tab, select **Synergy Integration** in the Source control provider choice.
4. Click **OK**.

The next time you start Tau you will be prompted to Log in to Synergy. In addition to this a new menu, **Synergy**, will be available together with two new toolbars as described in the previous section. There will also be a Synergy tab in the Output window.

**See also**

"Multiple configuration management tools" on page 2398

## Log in to Synergy

When Synergy is used as the source control system and the Source control provider option is set to **Synergy integration**, you will be prompted to log in to a Synergy server each time you start Tau. The login dialog contains the following fields:

1. User identity: The user identity you use in Synergy
2. Password: Your password to Synergy
3. Database path: The path on the Synergy server where the database is located.
4. Engine host: The name of the computer where the Synergy server is located.
5. Synergy home: The path on the local machine to your Synergy workspace.

If you cancel the login dialog (or have not set the Source control provider option) you can not log in to the Synergy server without restarting Tau.

Cancelling the login also disables the Tau Synergy integration.

**Note**

*If no Synergy functionality is used during a period of time, the connection to the Synergy server is temporarily shut down to save resources. The connection is automatically reestablished when it is needed.*

# Synergy project handling

## Opening an existing Synergy project

It is possible to access a UML project stored in a Synergy database directly from Tau. This is accomplished using the command Open Managed Project in the **Synergy** menu.

An alternative way to open a project that already is part of the local Synergy workspace is to use the standard **Open** command, browse to your local Synergy workspace and select one of the UML projects located here.

**Note**

*Tau only considers Tau projects (.ttp file) with the same name as the Synergy project name as managed projects. If a Synergy project contains more than one Tau project you must open that using the **File->Open** menu and browse.*

**Note**

*The workspace file (`.ttw`) is not managed by Synergy.*

## Creating a new Synergy project

It is possible to create a new UML project and store this on a Synergy project. This is done simply by:

- Creating a project using the **File->New** command.
- In the wizard choose the Synergy Project tab. This will show a list of the available project types.
- Select one of the project types. Fill in the name of the project.
- Fill in the release.
- Fill in project kind info as "**For Purpose**".
- Click **OK**.

The wizard that opens will let you specify more details of the project. When finished you will get a new project. A Synergy task will automatically be created called “Creating project <name of project>”. When you have created an initial version of the model and completed the task the model will be stored in the Synergy repository.

### Automatic Task Handling

The Synergy integration contains a feature that will automate the handling of Synergy tasks. When you try to do an operation that requires a task, for example check out an object, the task creation dialog will be displayed. In this dialog you can define the task name and also cancel the operation.

## Synergy project commands

All commands in this section operates on the currently active project in Tau.

Multiple selection is possible and commands applied will affect the selected files.

### Open Managed Project

This command will display a dialog that lists the managed projects on the Synergy server. When selecting one of the Synergy projects Tau will automatically load any UML project that is found in the Synergy project and that has the same name as the Synergy project. If an existing workspace already exists in Tau the project will be added to this workspace. If no workspace exists a new one will be created.

When the command is finished you will have the files comprising the UML project available in your local Synergy work area and the model will be loaded in Tau. If the Synergy project already is available in your local work area this is the project that will be opened. If the project is not available the files in the project will automatically be copied from the Synergy server to your local work area.

**Note**

*If the project is in a static state (e.g. integrate or released), the “Open Managed Project” operation will copy the project even if the project is available in the users work area. To open a project in a static state, the File->Open or File->Open Workspace must be used.*

### Migrate Project

The **Migrate Project** command will migrate the current project to Synergy. A copy of each file (and directory structure) the Tau project is comprised of will be created in Synergy.

**Note**

*The old project will still be loaded.*

When you choose the command a dialog will pop up where you can specify the Synergy project that will contain the UML files. The command will then perform the following actions:

1. Create a Synergy task called “Migrating <project name> to Synergy”
2. Create the Synergy project
3. Move all files in the UML project into the Synergy database.
4. Complete the task

**Note**

*The name of the Synergy project will by default be that of the Tau project. In cases where this clashes with an existing Synergy project, the user is asked for a new project name. The name of the Tau project will be renamed accordingly.*

### Project History

This command will show a graph describing the configuration management history of the Synergy project that contains the current project. The details of the graph are defined by the Synergy version you currently are using.

### Project Properties

This **Project Properties** command will show a dialog describing the configuration management properties of the current Synergy project. The details of the dialog are defined by the Synergy version you currently are using.

### Project Merge

When performing parallel development it is often necessary to merge models that have been modified by several different developers. When using Synergy the most common way of working is the following:

The project (UML project in this case) is developed based on a specific release.

At regular intervals (like for example after successful builds) a baseline version is created that contains all files used in the project.

Each developer has his own development project version in Synergy that is based on the baselined version of the project and in addition contains his/her recent changes.

A special project version is used for integration testing. This is also based on the baselined version of the project but is set up to contain the new version of all files that have been checked in since the last baseline.

The recommended strategy for merging is that all developers always do a merge where they get all changes from the integration project version *before* they check in their own changes.

To do this merge in Tau it is necessary to have access to the baseline project version and the integration project version.This is specified the following way:

- Choose the **Project Merge** command.
- In the dialog that appears, choose the project version to merge with and the common ancestor(s). In this dialog you can also choose the current task or create a task specific for the merge operation. Click **Ok** to continue.
- The Review differences dialog appears with Ancestor and Version 2 preselected.
- Perform the merge as usual. After the merge is performed and all conflicts resolved you can complete the task and thus check in all modified files

The history links of all files that have been modified by the merge will automatically be updated to reflect the merge that was done.

### Synchronize Project

The **Synchronize Project** command will synchronize your local Synergy workspace with the corresponding workspace on the Synergy server.

### Update Project

The **Update Project** command will cause the resources in the current project to be updated with the latest versions from the Synergy server.

## Synergy task commands

### Create Task

The **Create Task** command will pop up a dialog that makes it possible to create a new Synergy task. The dialog will prompt for the name of the new task.

### Set Task

The **Set Task** command will pop up a dialog that allows you to select between the Synergy tasks that are available for you.

### Complete Task

If you select the **Complete Task** command the current task will be marked as completed in Synergy. This implies that all objects modified for this task automatically will be checked in.

### Task Properties

The **Task Properties** command will show the properties of the current Synergy task in a modal dialog. The actual dialog that is shown is defined by the Synergy version you are running.

### Current Task Box

The **Current Task** box in the tool bar will show the currently selected task. It will also in the pull-down menu make available the Synergy tasks that you recently have used and allow you to select a new task among these.

## Synergy object commands

Multiple selection is possible and commands applied will affect the selected files if possible. For example **Check in** will check in the Synergy objects corresponding to all selected elements. If the corresponding object was not checked out it is of course not checked in either.

**Note**

*Object commands performed in a diagram will affect the file holding the diagram.*

### Create Object

If you choose the **Create Object** command the resource that contains the currently selected UML object will be inserted into the Synergy project that contains the current UML project.

Only objects in the project directory or managed subdirectories in the project directory can be added to Synergy.

### Object Properties

This **Object Properties** command will show a dialog describing the configuration management properties of the resource that contains the currently selected UML object. The details of the dialog are defined by the Synergy version you currently are using.

### Object History

This command will show a graph describing the configuration management history of the resource that contains the currently selected UML object. The details of the graph are defined by the Synergy version you currently are using.

### Check Out Object

If you choose the **Check Out Object** command the resource that contains the currently selected UML object will be checked out from the Synergy server. This implies that it will be available for editing and (depending on its Synergy status) may be locked and thus not available for editing by other users.

### Check In Object

If you choose the **Check In Object** command the resource that contains the currently selected UML object will be stored on the Synergy server. This also implies that it will not be available for editing until it is checked out again.

### Undo Check Out Object

If you choose the **Undo Check Out Object** command the resource that contains the currently selected UML object will be reverted to the latest version available on the Synergy server. This implies that it will not be available for editing until it is checked out again.

**Note**

*This command removes the checked out version of the object.If the checked out version is the initial version of an object the object is removed.*

## Synergy version handling commands

### Refresh Status

The command updates the Synergy status of each object for the active project. This is needed if the status has changed outside of Tau, for example directly in the Synergy user interface.

## Merge UML Projects using Synergy

When performing parallel development it is often necessary to merge models that have been modified by several different developers. When using Synergy the most common way of working is the following:

The project (UML project in this case) is developed based on a specific release.

At regular intervals (like for example after successful builds) a baseline version is created that contains all files used in the project.

Each developer has his own development project version in Synergy that is based on the baselined version of the project and in addition contains his/her recent changes.

A special project version is used for integration testing. This is also based on the baselined version of the project but is set up to contain the new version of all files that have been checked in since the last baseline.

The recommended strategy for merging is that all developers always do a merge where they get all changes from the integration project version *before* they check in their own changes.

To do this merge in Tau it is necessary to have access to the baseline project version and the integration project version. The simplest way to achieve this is that all developers have local work areas corresponding to these project versions. An alternative approach is that all developers share common network directories where the integration and baseline projects are stored.

Once you have access to your own project version, the baseline version and the integration version the merge is done the following way:

- Choose the command **Tools->Merge**
- In the merge dialog specify the baseline version of the project as the **Ancestor** and the integration version of the project as **Version 2**. Note that it is the `.ttp` files that should be selected to do a project merge.
- Perform the merge as usual. After the merge is performed and all conflicts resolved you can complete the task and thus check in all modified files

The history links of all files that have been modified by the merge will automatically be updated to reflect the merge that was done.

**See also**

"Merge versions" on page 145 in Chapter 6, *Working with Models*

"Configuration Management" on page 2385

"Multiple configuration management tools" on page 2398

# Generic Source Code Control Integration

## Integration with IBM Rational ClearCase

This section describes how to integrate Rational ClearCase with Tau. For further details see the Rational ClearCase user documentation.

The integration uses the Microsoft Source Control Interface and the commands that are supported for ClearCase are the same as described in section "Source control commands" on page 2399.

# Install IBM Rational ClearCase integration

### Windows

The source control system that will be used is specified in a system registry When ClearCase is installed, this source control system registry key should be set automatically. However, if you have more than one configuration management tool installed locally, you may have to edit this value manually.

1. Start Tau.
2. On the **Tools** menu, click **Options**.
3. In the general tab, select **Generic Source Control (SCC)** in the Source control provider choice.
4. Click **OK**.

The next time you start Tau, a new menu, Source Control, will be available from the Project menu. A new toolbar is also added as well.

### UNIX

You need to set the register keys and the environment variables. You should be logged on with your normal user identity. If you run on a network with multiple UNIX versions available you have to make a set-up for each UNIX version.

1. Make sure that the ClearCase PATH environment variable is set correctly.
2. To set the register keys, run the script:

   ```
   <installationsdir>/bin/setreg_ClearCase
   ```

   You only have to run the script the first time you will access ClearCase.
3. To set the environment variables, run the script:

   ```
   source <installationsdir>bin/setenv_ClearCase
   ```

   If you do not update your login file with this path, you need to re-run it each time you login.
4. Start Tau.
5. On the **Tools** menu, click **Options**.
6. In the general tab, select **Generic Source Control (SCC)** in the Source control provider choice.

7. Click **OK**.

The next time you start Tau, a new menu, Source Control, will be available from the Project menu. A new toolbar is also added as well.

**Note**

*Before you can use the source control commands, the files in your project must be located in a ClearCase view. See the IBM Rational ClearCase documentation for further instructions.*

**See also**

"Configuration Management" on page 2385

"Multiple configuration management tools" on page 2398

IBM Rational ClearCase user documentation

# Multiple configuration management tools

### Windows

If you have more than one Configuration Management (CM) tool installed, you must select which CM tool that you will use. This is determined by the value of the registry key:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\SourceCodeControlProvider]
```

If you want to change provider, you must change the value of this registry key.

The providers that you have installed are listed as values for the registry key:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\SourceCodeControlProvider\Inst
alledSCCProviders]
```

**Example 800: Registry key settings**

**Microsoft SourceSafe**:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\SourceCodeControlProvider]
   "ProviderRegKey"="Software\Microsoft\SourceSafe\ccm"
```

**IBM Rational ClearCase**:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\SourceCodeControlProvider]
```

```
"ProviderRegKey"="Software\Atria\ClearCase"
```

**UNIX**

If you have more than one Configuration Management (CM) tool installed, you must select which CM tool that you will use.

You need to set the register keys and the environment variables. You should be logged on with your normal user identity.

1. To change the register keys, run the script for your configuration management tool:

   **ClearCase**: `<installationsdir>/bin/setreg_ClearCase`

   You only need to run the script the first time you will access your CM tool.

2. To change the environment variables, run the script for your configuration management tool:

   **ClearCase**:
   `source <installationsdir>bin/setenv_ClearCase`

   If you previously have updated your login file with one of the paths above, just edit that file.

## Source control commands

The basic commands and functions of your CM tool is available in a separate Source Control menu which is available in the Project menu. The commands are also available via the source control toolbar.

For a number of commands, a file dialog opens where you can select which files that the command should apply to. Depending on the command and the file or folder you selected before the command is issued, the listed files in the dialog may vary. For instance if you select the check out command on a folder, only files that are still checked in are listed in the dialog.

**Note**

*The following commands are defined in the Microsoft SCC Interface documentation. All commands may not be available for your configuration management tool and the functionality of the commands may vary. Your configuration tool may also add commands and toolbar buttons that are unique for that configuration management tool. They are not listed here.*

### Get latest version

This command retrieves the latest copy of the file from the source control server and copies it to your computer. It is still read-only. If you are working in on a project with more than one team member, update your local copies frequently to incorporate changes made by others.

1. Click the file or folder in the File View.
2. On the **Project** menu, point to **Source Control** and click **Get Latest Version**. The file selection dialog opens.
3. Select the files you want to update and press **OK**.

### Check out

This command retrieves the latest version of a file or folder from the source control server and reserves the file or folder for you. The file is copied to your computer and the status changes from read-only to read/write. Unless you allow multiple check outs, the file is locked on your source control server.

Use CTRL + ENTER for line breaks in the comment field.

1. Click the file or folder in the File View.
2. On the **Project** menu, point to **Source Control** and click **Check Out**. The file selection dialog opens.
3. Select the files you want to check out and press **OK**.

**Note**

*The file or folder will only be automatically updated to its latest version before check out if the option "Automatically update files" is enabled. If this option is disabled you must yourself update the file or folder before check out. Some CM systems may also allow you to check out a file from a version other than the latest (for example to create a branched version).*

### Check in

This command copies your local version to the source control server as the latest version of the item. The status of the item changes from read/write to read-only. It is now possible for other users to check out the file. If you have not made any changes to the item you should undo the check out rather than check in the item.

Use CTRL + ENTER for line breaks in the comment field.

1. Make sure that you have saved your files.
2. Click the file or folder in the File View.
3. On the **Project** menu, point to **Source Control** and click **Check In**. The file selection dialog opens.
4. Select the files you want to check in and press **OK**.

Elements that are checked in and therefore not editable are marked with a Gray bar between the element symbol and the element name. The bar does not relate to if the file that the element belongs to is writable, it is an internal flag only.

### Undo check out

This command returns the item to the source control server. No changes are saved. This is the command you should use if you have not made any changes to a file that you have checked out.

1. Click the file or folder in the File View.
2. On the **Project** menu, point to **Source Control** and click **Undo Check Out**. The file selection dialog opens.
3. Select the files you want to undo the check out and press **OK**.

### Add to source control

This command adds the item to the source control server.

1. Click the file or folder in the File View.
2. On the **Project** menu, point to **Source Control** and click **Add to source Control**. The file selection dialog opens.
3. Select the files you want to add and press **OK**.

**Note**

*If you selected to add the project file, all files in the project will be listed in the file selection dialog.*

**Note**

*The underlying source control provider may impose restrictions on the files added to source control that can cause the "**Add to source control**" operation to fail. Common restrictions relate for example to the file location, file names and user privileges. Refer to the documentation of your source control provider for detailed information.*

### Remove from source control

This command allows you to remove files and folders from the source control server. You cannot remove entire projects or solutions with this command. See your source control documentation for further instructions.

1. Click the file or folder in the File View.
2. On the **Project** menu, point to **Source Control** and click **Remove from source Control**. The file selection dialog opens.
3. Select the files you want to remove and press **OK**.

### Show history

A configuration management tool keeps a record for all versions of items that you have added to the source control server. This command allows you to list the history record for one file at a time.

1. Click the file or folder in the File View.
2. On the **Project** menu, point to **Source Control** and click **Show History**.

### Show differences

This command allows you to show the differences between your local copy of an item and the latest version on the source control server. This command can only be applied to one file at a time.

1. Click the file or folder in the File View.
2. On the **Project** menu, point to **Source Control** and click **Show Differences**.

### Source control properties

This command displays the properties of the selected item. The dialog that opens is tool dependent.

1. Click the file or folder in the File View.
2. On the **Project** menu, point to **Source Control** and click **Source Control Properties**. Your configuration management tool displays the properties of the item.

### Refresh status

This command updates the status of the selected items from your configuration management tool. This is a useful command if you have done operations on the files directly in your configuration management tool. If you refresh the project file, (*.ttp), that status of all files in the project will be refreshed.

1. Click the file or folder in the File View.
2. On the **Project** menu, point to **Source Control** and click **Refresh Status**.

### Execute CM tool

This command invokes the Configuration Management (CM) tool that is connected to Tau. The CM tool that is connected is dependent on the settings of the registry key:

```
[HKEY_LOCAL_MACHINE\SOFTWARE\SourceCodeControlProvider]
```

1. Click the file or folder in the File View.
2. On the **Project** menu, point to **Source Control** and click **Source Control**.

### Import module

This command allows you to find files that are stored on your source control server but are not part of the project you are working with. The files that you find are added to your local computer and inserted in your project.

1. Click the file or folder in the File View.
2. On the **Project** menu, point to **Source Control** and click **Import from Source Control**.

**Note**

***Import from Source Control*** *is not used with ClearCase.*

# Compare and Merge from a Source Code Control tool

This section describes how to launch Tau for compare and merge operations on versions of an u2 file selected in the version browser of a Source Code Control tool.

Detailed setup instructions are provided for Telelogic Synergy and IBM Rational ClearCase.

**Note**

*There is only support for compare and merge operations on u2 files.*

## Setup Telelogic Synergy

To be able to launch Tau from Synergy you need to:

- Install the file type definition for Tau u2 files see Tau file type definitions.
- Setup compare and merge operations for your Synergy client.
- And if you already use Synergy for source control of Tau u2 files you may want to; change existing elements to the new object type if you haven't used the enclosed definitions when creating the objects.

### Setup compare and merge operations

The compare and merge tool are set by default in the ccm.properties file located in the `etc` directory of your client installation for Windows and in .ccm.user.properties file in your home directory for UNIX.

- Windows

```
windows.tool.compare.tau_model  =
"c:\\Program Files\\Telelogic\\TAU_4.2\\bin\\U2FileUtility.exe"
xcompare "%file2" "%file1"
windows.tool.merge.tau_model  =
"c:\\Program Files\\Telelogic\\TAU_4.2\\bin\\U2FileUtility.exe"
xmerge -base "%ancestor" -out "%outfile" "%file2" "%file1"
```

- UNIX

```
unix.tool.compare.tau_model  = .../bin/U2FileUtility xcompare
%file2 %file1
```

```
unix.tool.merge.tau_model  = .../bin/U2FileUtility xmerge -base
%ancestor -out %outfile %file2 %file1
```

Path can be excluded if the bin directory is in your path.

**Note**

*Synergy requires double back slash separators on Windows.*

### Change existing elements to the new object type

If you already have Tau u2 files stored in Synergy you need to change the object type on these files. This is done with the change_type command, please see Synergy documentation for further information.

Change the object type on any u2 files already stored in the database using the command:

```
ccm change_type file_spec -type tau_model
```

## Setup IBM Rational ClearCase

To be able to launch Tau from ClearCase you need to:

- Configure the ClearCase type manager.
- Create a new element type
- Optionally, setup ClearCase magic files for the new element type
- And if you already use ClearCase for source control of Tau u2 files you may want to; change existing elements to the new element type

### Configure the ClearCase type manager

In ClearCase type managers are setup through a map file in the ClearCase client installation. Please see ‘cleartool man type_manager’ for further information.

To install on Windows add the following lines to the end of the map file (located in the directory `lib\mgrs\map` in the ClearCase home directory):

```
u2_manager      construct_version       ..\..\bin\zmgr.exe
u2_manager      create_branch           ..\..\bin\zmgr.exe
u2_manager      create_element          ..\..\bin\zmgr.exe
u2_manager      create_version          ..\..\bin\zmgr.exe
u2_manager      delete_branches_versions..\..\bin\zmgr.exe
u2_manager      xcompare                C:\Program
Files\Telelogic\TAU_4.2\bin\U2FileUtility.exe
u2_manager      xmerge                  C:\Program
```

```
Files\Telelogic\TAU_4.2\bin\U2FileUtility.exe
u2_manager      get_cont_info           ..\..\bin\zmgr.exe
```

To install on UNIX do as follows:

- Move to `lib\mgrs` in the ClearCase home directory
- Copy the `z_whole_copy` directory to `u2_manager preserving the links, e.g. use tar to copy the directory`
- Move to `u2_manager`
- Remove links named `compare`, `xcompare`, `merge` and `xmerge`
- Create new symbolic links to `xcompare` and `xmerge` in the Tau `bin` directory

### Create a new element type

ClearCase creates new element types using the mkeltype command, please see 'cleartool man mkeltype' for further information.

To create a new element type for Tau u2 files do as follows:

- Move to the VOB you want to create the new element type for
- Create the new element type with the command

```
cleartool mkeltype -supertype file -manager u2_manager
tau_model
```

where tau_model is the name for the new type.

Note: You can use the -global flag to update several VOBs in one operation.

### Setup ClearCase magic files for the new element type

When a new element is created with mkelem, the option -eltype can be used to specify what type the element should be. With the use of magic files new elements get the correct element type automatically (without the use of -eltype). Please see 'cleartool man cc.magic' for further information.

Add the following line to the default.magic file (<clearcase home dir>\config\magic\default.magic)

```
tau_model text_file : -name "*.[uU]2";
```

Note: The line must be inserted above the lines

```
# catch-all, if nothing else matches
compressed_file : -name "*" ;
```

### Change existing elements to the new element type

If you already have Tau u2 files stored in ClearCase you need to change the element type on these files. This is done with the chtype command, please see 'cleartool man chtype' for further information.

Change the element type on any u2 files already stored in the VOB using the command:

```
cleartool chtype -force tau_model <file>
```

# 86

# *Working with links*

This chapter describes how to work with links in Tau. A link is a relation between two elements. Links can be created between any kind of elements, for example between two model elements, between a model element and a requirement in DOORS or from a model element to an external web page.

There are two different kinds of links:

- Hyperlink
- Dependency link

The Managing links section describes how to create, delete and navigate links.

# Hyperlink

A hyperlink relates two elements using a standard Uniform Resource Location (URL), just like a link on a web page. The target of the link can therefore be anything that can be identified by a URL, for example a diagram, a file in the file system or a web page.

The format of a Tau URL is:

```
tlog://<project_path>:plugin_ident:string_ident
```

It consists of the following parts:

- `<project_path>` is the windows path of the project containing the object.
- `plugin_ident` is the Tau internal identification of the Tau add-in which owns the object.
- `String ident` is a string representing the object for a given project of a given plug in.

**Example 801**

This example illustrates a URL obtained from a class in a typical UML model:

```
tlog://<C:\MyModels\LinksProj\LinksProj.ttp>:u2:iOepWITH4FFLCghYEIWspKrL
```

The project is called LinksProj and it contains a class called Class1. The `u2` part of the URL identifies the class as a UML element. The class itself is represented by its GUID, `iOepWITH4FFLCghYEIWspKrL`.

---

## Visualization

Outgoing hyperlinks are indicated by a blue underline on the element name, whether it appears in the workspace window or in a diagram.

Incoming hyperlinks are indicated by a small blue triangle next to the name of the element in the workspace window, and in addition by a dashed blue underline in diagrams.

# Navigating hyperlinks

### Navigating hyperlinks within Tau

Hyperlinks can be navigated in diagrams by holding down **Ctrl** and clicking the text with the dashed blue underline.

### Navigating hyperlinks in external applications

When navigating a URL in an external application, for example a web browser, the expected behavior is to start Tau, load the model containing the element and locate it. Due to technical limitations this is not possible on all platforms.

### Missing targets

Depending on the set of currently loaded project(s) in your workspace, the target of a link can be missing.

If a missing target refers to a Tau element, an error message is issued when navigation fails.

**Note**

*The link source can not be missing, as the source is the carrier of the link information.*

# Creating hyperlinks into a Tau model

Each element in Tau has a unique URL and can be linked to from an external source, such as a web page.

To obtain the URL from an element:

- Select the element in the workspace window or in a diagram
- Right-click and select Copy URL from the context menu
- Paste the URL text in the desired application

# Dependency link

A dependency link is relates two elements using a UML Dependency. It can be used to relate any UML elements. The main advantage of a dependency link is that it is represented using plain UML and can be created, edited, deleted and visualized in diagrams.

**Note**

*All dependencies are **not** considered to be dependency links. Only dependencies with certain stereotypes are. These are described in Requirement relations. It is possible to create any dependency using the link commands, but they will not have link indicators or be present in the Links dialog unless they are requirement relations.*

A dependency link can refer to its source and/or target in two different ways: normal or symbolic.

In the normal case, the client and supplier of the dependency simply refers to the source and target elements. The reference can be set up using names (potentially qualified) or GUIDs.

In the symbolic case, the dependency is annotated with additional information consisting of symbolic references (URLs) to the source and target. The client and/or supplier in this case refers to a predefined element called `ExternalObject`. The symbolic references makes it possible to use dependency links to/from any UML element and to external elements, for example a requirement in DOORS.

## Visualization

Outgoing dependency links are indicated by a small purple triangle next to the name of the element in the workspace window and on symbols.

Incoming dependency links are indicated by a small orange triangle next to the name of the element in the workspace window and on symbols.

**See also**

*Requirement relations*

# Managing links

Links can be created, edited and deleted in several different ways by using the Links menu, the Links toolbar the Links dialog or the Link sub-menu in the context menu. The Link commands section contains a list of all available commands.

Dependency links can also be edited directly in the UML model.

When working with DOORS dependency links can also be created using drag'n'drop between elements in Tau and elements in DOORS. Compare section Creating Links.

## Creating links

Links can be created in several different ways. Using drag & drop is usually the most convenient way, but if there are multiple targets, or the target is not yet located, manual link creation has to be used.

### Creating a link using the toolbar

To create a link using the Links toolbar:

- Select the correct link type from the **Current link kind** list in the Links toolbar.
- Select the source element(s)
- Click **Start Link** in the Links toolbar, or use the shortcut **Ctrl+K**.

  This makes the selected element(s) become active link sources, and they will be remembered until the link is created or until they are cleared.

  If the link is a Hyperlink, the Insert Hyperlink dialog is opened to allow specification of external hyperlinks.
- Select the target element(s)
- Click **Make link from start** in the Links toolbar, or use the shortcut **Ctrl+L**.

Links are now created from the selected source element(s) to the selected target element(s).

If the link type is Dependency link, the Stereotypes dialog is displayed once for each link to allow specification of the link type. The previous choice in the Stereotypes dialog is remembered during a Tau session.

**Note**

*If the By default, make hyperlink to a workspace element option is enabled the Insert Hyperlink dialog is not displayed during link creation.*

### Creating a link using drag & drop

To create a link using drag & drop:

- Select the correct link type from the **Current link kind** list in the Links toolbar.
- Select the source element(s)
- Press the right mouse button and drag the element onto the target element.
- Release the mouse button. Select **Link** from the context menu.

**Note**

*In some views drag and drop using the left mouse button has no semantic meaning. In these views links can also be created using left button drag and drop. The context menu is not need in this case. The tabs in the Output Window are examples of such views.*

### Creating links in diagrams

Dependency links can be drawn in diagrams since they're represented by dependencies. To create a dependency link in a diagram:

- Select the link source and use the appropriate line handle to create a dependency link
- If desired or needed apply a stereotype to the dependency to make it a dependency link. A list of dependency link types can be found in the Requirement relations section.

### Creating multiple links from the same source

The source element(s) of a link can be locked to create many links from the same element(s) without having to reselect it each time.

To lock the link source element(s):

- Select the correct link type from the **Current link kind** list in the Links toolbar.
- Select the source element(s)

- Click **Start Many Links** in the Links toolbar

This makes the selected element(s) become active link sources, and they will be remembered until they are cleared by clicking **Clear start links**.

To create a link from the source element(s) to one or more target elements:

- Select the target element(s) and click **Make link from start**, or use the shortcut **Ctrl+L**.

This can be repeated any number of times to create links to many different targets from the same source. To clear the source element(s), click **Clear start links**.

### Automatic link creation

There is a special mode for link creation called **Automatic link creation**. This mode allows you to automatically create links from a pre-selected source to all modified elements.

When this mode is active a link will be created as soon as an element is modified. The source is the pre-selected source element, and the target is the modified element.

To activate the automatic link creation mode:

- Enable the option Automatically create links between modified objects and active link end
- Select the link source(s) and click Start Many Links

To de-activate the automatic link creation mode:

- Click **Clear start links**

## Deleting links

To delete one or more links from a given element:

- Select the element that is the source or target of the link
- Click **Edit links** in the Links toolbar
- Select the correct link type from the **Active link** drop-down list
- Set the **Direction** to `Outgoing` or `Incoming` depending on if you selected the target element or the source element.

- Select the target element in the list dialog and click **Delete**

Links can also be deleted by using the Links context menu or the Links menu in the main menu bar.

If the link is a dependency link the link can be deleted directly from the model. In the model view just select the dependency representing the link and hit the `Del/Delete` key. If the dependency is visualized in a diagram, select the line and execute the **Delete from Model** command.

## Navigating a link

To navigate a link in the workspace window or in a diagram:

- Right-click the link indicator symbol
- Select the correct link type from the context menu
- Select the link target element or URL

The result of navigating a link is different depending on the link type and the installed integrations. The default behavior is to select the element in the workspace window, and if possible also highlight it in a diagram. Some integrations may launch an external application to show the target.

**See also**

Navigating hyperlinks

## Link commands

This section contains a full list of the commands available on links. Note that add-ins and integrations may add additional commands. The commands can be accessed from the Links menu, the Links toolbar, the Links dialog or the Links sub-menu in the context menu.

### Start Link (CTRL+K)

Starts creation of a link and specify a link source. See Creating links.

### Start Many Links

Starts creation of many links and specify the link sources. See Creating links.

### Make Link from Start (CTRL+L)

Ends link creation and specifies the link target(s). See Creating links.

### Copy URL

Copies the hyperlink URL of the selected element as text to the clipboard. The text can be pasted and used in any application supporting URLs to navigate to the element.

### Display Outgoing Links

This command will produce a list in the link tab of the Output window of the links going out from the selected entity.

### Display Incoming Links

This command will produce a list in the link tab of the Output window of the links coming in to the selected entity.

### Edit Links

Allows manipulation of both in- and outgoing links of the selected element. See Links dialog.

### Link Options

View and/or edit the link options. See Link options and Hyperlink options.

## Links menu

The Links menu is a part of the main menu in Tau, and contains most link commands.

## Links toolbar

The Links toolbar contains most link commands and is often the most convenient way to manipulate links.

The most important use of the link toolbar is to set the link kind using the Current link kind control.

**Current link kind**

Specifies the link kind used during link creation. The correct kind must be selected before specifying the link source(s).

Available link kinds are:

- Hyperlink
- Dependency link

**Note**

*The list of available link kinds depends on the platform and the installed integrations.*

## Links dialog

The links dialog can be used to view or delete the in- and outgoing links of the selected elements. Its primary purpose is to delete links.

To show the Links dialog:

- Select the element
- Click **Edit links** in the Links toolbar.

The Links dialog shows all the objects connected with a selected object. An object that is not loaded into Tau is referred to by its URL instead of its name.

The **Active link** drop-down list controls which type of links that are shown. only one type can be active.

The **Direction** radio-buttons controls if outgoing or incoming links are displayed.

The **Delete** button deletes the selected link(s).

## Insert Hyperlink dialog

This dialog allows you to search for a target for a link. The following is present in the dialog.

- **Link to:** text field allowing a selection if the link is external (web page or file) or within the elements of the current workspace.
- **Text to display** will be visible through the link output tab or the Links dialog, in the **name** column.

- **Unique Resource Location of the hyperlink target:** text field allowing you to specify a web page location directly.
- List for selection of recently used files, web pages and links, with radio buttons for **Recent files**, **Browsed pages** and **Inserted links** to be displayed in the field.

## Link options

Tau offers you the possibility to customize link creation behavior. You can change link options via the Tools menu by selecting Options or clicking on the Display Link Options toolbar button.

### Active link end is an active target, not an active source

If this option is disabled, then when you use automatic creation of links, you will create links from your active link end to the other models.

If this option is enabled, then you will create links to your active link end from the other models.

### Automatically create links between modified objects and active link end

If this option is enabled, then when you select an active link end, all your modifications will be linked to this link end.

### Show link indicators

If this option is enabled, Tau will show the link markers.

### Use requirement as target when creating links by drag-and-drop

This option affects the link direction when creating links using drag-and-drop.

When this option is enabled (default), links will be created form the model element to the requirement when you drag-and-drop a requirement to a model element as this is normally the desired direction.

# Hyperlink options

Tau offers you the possibility to customize link creation behavior via the Tools menu by selecting Options or clicking on the Display Link Options toolbar button.

### By default, make hyperlink to a workspace element

When the Insert Hyperlink dialog is started this option controls the start setting of the **Link to** text field.

When checked, the application allows you to select the target of your hyperlink within your workspace. Otherwise, you are prompted to specify another type of target, such as an existing file or a web page.

# 87

# *Visual Studio Integration*

The Tau Visual Studio Integration provides an integration to the Microsoft Visual Studio 2005 IDE. The integration consists of a number of commands added to both Tau and Visual Studio in order to facilitate features such as seamless two-way navigation between UML models and generated code.

Visual Studio projects can be created from existing UML projects, and populated with generated source files.

The Visual Studio Integration works both for the C++ and C# languages. Most functionality is the same regardless of which of these languages that is used. Functionality that is specific to a certain language is explicitly marked in this document.

# Installing the Integration

The Visual Studio integration consists of two add-ins:

- A Tau add-in called `MSVS8Integration`
- A Visual Studio add-in called `TAUG2IntegrationAddin`

These add-ins have to be correctly activated in order for the integration to work properly.

## Activate the Visual Studio add-in

To activate the `TauG2IntegrationAddin` in Visual Studio it must first be installed:

1. Make sure that Visual Studio is properly installed and not running.
2. Install the add-in from your Start Menu; point to All Programs, select Telelogic, Telelogic Lifecycle Solution Tools, Telelogic Tau 4.2, **Install Microsoft Visual Studio 2005 integration**. Alternatively, execute the specific `Setup.exe` for the add-in typically located in:

   `C:\Program Files\Telelogic\TAU_4.2\integrations\MSVS8`
3. Follow the instructions of the setup program.

The add-in will be automatically activated the next time Visual Studio is started. You can see that the add-in is activated by the presence of a Tau menu in Visual Studio.

**Note**

*Commands connected to Visual Studio menu controls are stored in the environment and not in the add-in. In order to install new versions or recover from a corrupt command environment, do the following:*
*1) Uninstall the TAUG2IntegrationAddin via Add/Remove programs.*
*2) Execute **devenv /setup** from the Visual Studio command prompt.*
*3) Reinstall the TAUG2IntegrationAddin.*

## Activate the Tau add-in

The `MSVS8Integration` add-in must be activated for every project that you want to use with Visual Studio. To activate the Visual Studio support:

1. Select **Customize** from the **Tools** menu.
2. Click the Add-Ins tab and check the MSVS8Integration add-in.

3. Click **Close**.

You can see that the add-in is activated by the presence of a Visual Studio menu in Tau.

The `MSVS8Integration` add-in can also be activated automatically when creating a new Tau project for C++ or C# code generation, by checking the "Enable MS Visual Studio Integration" checkbox in the New Project wizard.

# Using Visual Studio with Tau

## Connecting Tau and Visual Studio

In general you may have more than one instance of both Tau and Visual Studio running on your machine. Most integration commands require that a connection is established between one particular instance of Tau and one particular instance of Visual Studio. The connection procedure takes place automatically the first time you invoke such an integration command from either Tau or Visual Studio.

If the other tool has not been started yet it will be started automatically so that a connection can take place.

If at least one instance of the other tool is running a dialog will pop-up to let you decide which instance of the tool you want to connect to. In this dialog you also have the possibility to connect to a new instance of the tool.

If the connection should fail, make sure that the integration has been properly installed according to the instructions in Installing the Integration.

## Workflow

The Tau Visual Studio Integration enables the following workflow for both C++ and C# application development:

- You develop a UML model in Tau intended for C++ or C# code generation. After source code has been generated from the model the integration lets you create a corresponding Visual Studio project. This project can then be built and debugged in Visual Studio.

- Once a Visual Studio project has been created you can work with the application in either Tau or Visual Studio or in a combination of these environments. Changes you make to the model are generated to the code by the Tau C++ or C# code generator, and changes you make to the code can be propagated to the model using the model update (roundtrip) mechanism of Tau.
- Navigation commands in both Tau and Visual Studio make it possible to navigate between the C++ or C# source code and the UML model in both directions.

The integration also provides certain commands that help you when debugging the generated application in Visual Studio. For example, it is possible to trace the execution in Tau automatically during debugging.

**For C++** the tracing can be done in a Tau UML sequence diagram. It is also possible to switch to the Visual Studio debugger when the application has stopped at a breakpoint in the Tau UML debugger. During debugging navigation to the model can also be automated.

The UML model you develop with Tau does not have to be created from scratch. It is common that parts of the C++ or C# application consists of legacy code that should be reused. You can import such code into Tau to use it from the UML model. You can then choose to either regenerate the code using the C++ or C# code generator, or leave the code as is and just access it from the UML model.

The navigation features of the Visual Studio integration will work both for code that has been imported to Tau or generated from Tau.

## See also

C/C++ Import

Importing Existing C# Code

# Integration Commands

## Tau Commands

### Create/Update Visual Studio project

This command is used both for creating a new Visual Studio project from an existing UML model, and for updating an existing Visual Studio project with changes in the UML model.

To create a new Visual Studio project from an existing UML model:

1. Generate C++ or C# code for the model (or from part of the model).
2. Select an entity in the Model View that is part of the model you have generated code for.
3. Select **Create/Update Visual Studio C++/C# Project** from the **Visual Studio** menu.

The kind of project that is created is different for C++ and C#:

- For C++ a project for building a console application will be created.
- For C# Visual Studio will ask you to specify which kind of project you want to create. Select a project kind that is appropriate for the application you are developing. You may also decide if the project shall be inserted in a new solution, or added to the currently open solution.

The new Visual Studio project will then be populated with all generated C++/C# files in the model. Appropriate project settings will also be set-up for the created project.

**Important!**

*Visual Studio has an option **Projects and Solutions - General - Save new projects when created**. This option must be enabled for this command to work. With this option disabled the created project will not be saved to disk which means Tau does not know where it is located.*

You should not mix C# and C++ development in the same Tau model. Use separate Tau projects for the C# and C++ parts of the application.

**Note**

*The Visual Studio C# project wizard you choose may create files you do not need. You should delete these files from the project. For example, if your model contains a class with a static Main operation, you should delete the file generated by the wizard that contains the corresponding C# Main method.*

The next time you invoke the **Create/Update Visual Studio C++/C# Project** command the project will only be updated to reflect any changes made in the model, such as adding new source files. Note, however, that files will never be removed from the project, even if those files are no longer generated from the model. You have to decide manually which files to remove from the project.

### Open Visual Studio project

This command locates the Visual Studio project to which the Tau model is connected. It can be useful if the Visual Studio solution contains several projects, or if you have multiple instances of Visual Studio running.

### Locate an element

When C#/C++ code has been generated for a model it is possible to navigate to the source code location in Visual Studio for a selected model element.

For C# this command is called **Locate in Visual Studio project**.

For C++ elements often end up in two files, a header file and an implementation file. There are therefore two separate commands; **Locate in header file** and **Locate in implementation file**.

When an element has been located in a source file, that file will be opened in Visual Studio and the cursor will be positioned at the location of the element within the file.

Another way to navigate to generated C#/C++ source is to use the **Go to source** command in the context menu for a selected element in the Model View. That command is enabled if the selected element has at least one representation in the generated code. Normally the source file is opened in the Tau text editor, but when the Visual Studio integration is enabled it will instead open the file in Visual Studio.

### Transfer control to target debugger

This command is available for C++ only.

When debugging a generated C++ application with Tau using the UML debugger, control can be transferred to the Visual Studio debugger when the Tau debugger is in break mode. This command is available in the form of a tool button in the Tau Model Verifier toolbar.

Control is transferred to the Visual Studio debugger by setting a breakpoint on the next executable statement in the generated code, and then issuing a Run command in the Tau debugger.

## Visual Studio Commands

### Locate in Tau

This command can be used in order to navigate to an element in the Tau model from a generated C#/C++ source file that is open in Visual Studio.

1. Select an element in the Visual Studio code editor by placing the cursor on it.
2. Select **Locate in Tau** from the Tau menu or tool button.

When the element has been located in the model, it will be selected in Tau's Model View. If it has a presentation element in a diagram that diagram will also be opened.

It is possible to automate the navigation to Tau when debugging the generated application, so that when hitting a breakpoint or issuing a debugger execution control command (such as Step or Run To Cursor), a relevant model element will be located in Tau. In order to use this mode, turn on Autolocate in Visual Studio**.**

### Connection Status

This command can be used to show which instance of Tau the Visual Studio add-in is connected to. If no connection has been established yet, the dialog can be used to set-up such a connection. See Connecting Tau and Visual Studio for more information about the connection between Tau and Visual Studio.

### Create/Update Tau Project

Use this command to update the Tau project with changes made in C#/C++ source code.

**Note**

*This command currently only supports model update for existing files. If new files have been added to the Visual Studio project these has to be imported in the Tau model as usual.*

### Autolocate

When Autolocate is enabled, every time the Visual Studio debugger enters break mode (after hitting a breakpoint or stopping after a debugger execution control command such as Step or Run To Cursor) Tau will highlight the model element which corresponds to the current position in the C#/C++ source code.

The Autolocate mode works best if the screen area is sufficiently large so that the Tau and Visual Studio applications can be placed side by side.

### Tau Trace

This command is available for C++ only.

The Tau Trace command allows the execution of a generated C++ application to be traced. Tracing can be done either to a log file, or to a Tau sequence diagram. The latter requires that Tau is running during tracing. If you decide to trace to a log file you may later visualize that file as a Tau sequence diagram using the **Import - Import Trace** command in Tau.

Trace functionality requires that instrumentation is enabled for the build artifact before generating C++ code from Tau. See Enable instrumentation for more information on how to generate an instrumented C++ application.

To activate tracing:

1. From the Tau menu select **Tau Trace**.
2. Select if you want to trace in the form of a log file or a sequence diagram.

**Note**

*Since the trace functionality is embedded in the generated executable, and not in the Visual Studio environment, changes to the trace settings can only be done during debugging sessions. Changes must thus be made when the execution is in break mode.*

# 88

# *Printing*

This chapter describes different ways of printing a diagram and how to change print settings.

# Adding and Removing Printers (UNIX)

The MainWin Control Panel allows you to make your printers available in the Tau print dialog.

**To open the MainWin Control Panel:**

- From the terminal window, type:
  `<installation directory>/bin/mwcontrol`

The control panel opens.

**To add a printer:**

1. When the control panel is open, click **Printers**. The Printer window opens.
2. Click **Add New Printer** and follow the instructions in the add printer wizard that opens.
3. When you have completed the wizard, close the printer dialog and the Control Panel.
4. Restart Tau.

**To remove a printer:**

1. When the control panel is open, click **Printers**. The Printer window opens.
2. Right-click the printer you want to remove and click **Delete**.
3. Close the printer dialog and the Control Panel.
4. Restart Tau.

# Printing Diagrams

There are several ways of printing diagrams. You can print single diagrams from:

- The diagram itself.
- The Model View.
- The Print Manager.
- The diagram preview window.

You can print multiple diagrams from:

- The Model View.
- The Print Manager.

**Note**

*Using a white/transparent background for an Icon image may result in a black background when printing. This is related to a Windows postscript driver PS level 2. Changing to PS level 1 may remove the situation. Using a colored background or frame will also prevent this.*

## Adding and setting up printers (UNIX only)

The procedure how to add and set up printers for use from Tau on UNIX hosts is done with the **MainWin Control Panel**, described in detail in the Installation Guide.

## Print settings

**To change print settings:**

1. On the **File** menu, select **Print Setup**.
2. In the Print Setup dialog, select printer, paper size and other properties allowed for the selected printer. The paper size and orientation will be used to determine the default diagram size in the editors.

3. Click **OK**.

1. Print files

**To print a file:**

1. Open the file that you want to print, and place the cursor somewhere in the text.
2. On the **File** menu, click **Print** or click the print icon in the toolbar.
3. In the Print dialog, change settings according to your preferences.
4. Click **OK**.

## Select diagrams to be printed

All diagrams in your model are available in the Model View. The Print Manager allows you to select which diagrams to print. To open the Print Manager, click **Print Manager**, on the **File** menu.

The diagrams that are included in the container that is active in the Model View, are listed in the Print Manager. Use the **Track Selection** button if you want to change container in the Model View. If the button is not pressed in, the contents in the Print Manager is locked to the first selection you made.

*Figure 290: Track selection button, when not selected*

You can also decide which type of diagrams you want to print by checking or clearing the diagram type check boxes in the **Filter** area.

You can calculate the number of pages to print by clicking **Pages** in the **Print** window.

## Preview of diagrams

**To get a preview of a diagram:**

1. Select the diagram in the Model View.
2. Select **Print Preview** on the **File** menu. A preview of the diagram is displayed.
   – You can scroll to other diagrams by using the **Next Page** and **Previous Page** buttons.

## Print a single diagram

**To print a single diagram from the diagram itself:**

1. Open the diagram.
2. Select **Print** on the **File** menu. The standard print dialog is displayed.

**To print a single diagram from the Model View:**

1. Select the diagram in the Model View.
2. Right-click the diagram and select **Print**. The standard print dialog is displayed.

**To print a single diagram from the Print window in the Print Manager:**

1. Select the diagram in the Model View. The diagram icon is displayed in the **Selection** area.
2. Click the **Print** button. The standard print dialog is displayed.

**To print a single diagram from the preview window:**

- Select **Print**. The standard print dialog is displayed

## Print multiple diagrams

**To print multiple diagrams from the Model View:**

1. Select the diagrams in the Model View.
2. Select **Print Manager** on the **File** menu. The **Print** window is displayed.
3. Click the **Print** button or select **Print Preview** on the **File** menu and then **Print**. The standard print dialog is displayed.

You can print diagrams of the same type(s) at the same time if you use the Print window and the Filter functionality.

**To print multiple diagrams from the Print window:**

1. Select **Print Manager** on the **File** menu. The **Print** window is displayed.
2. In the Model View, select the diagram(s) you want to print. The diagrams and page numbers for the diagram type(s) you selected are displayed in the **Selection** area.
3. Click the **Print** button or select **Print Preview** on the **File** menu and then **Print**. The standard print dialog is displayed.

# 89

# *Model Browser*

This section describes how to generate a navigable HTML view of a model or selected part(s) of a model.

HTML generation is performed by an add-in called ModelBrowser.

HTML generation can be performed interactively, see Generating HTML or from the command line, see Command line usage.

# Generating HTML

To generate a HTML view of a model:

- Make sure the ModelBrowser add-in is activated
- In the model view, select the element(s) you want to include in the report
- From the **Tools** menu select **Generate HTML**.

A HTML view is generated and displayed in the built-in web browser. For details, see HTML View.

**Note**

*For large models it can take a long time to generate the HTML view, and Tau is not responsive during this process.*

## Activating the ModelBrowser add-in

To activate the support for HTML generation:

1. From the **Tools** menu select Customize.
2. Click the Add-Ins tab and check the ModelBrowser add-in.
3. Click **OK**.

# HTML View

This section describes the appearance and contents of a HTML view.

## Contents

The HTML view contains the following information:

- A Tree-view corresponding to the Model View
- A set of Properties for each included element
- A set of Diagrams

The main page of the HTML view displays the information in three frames.

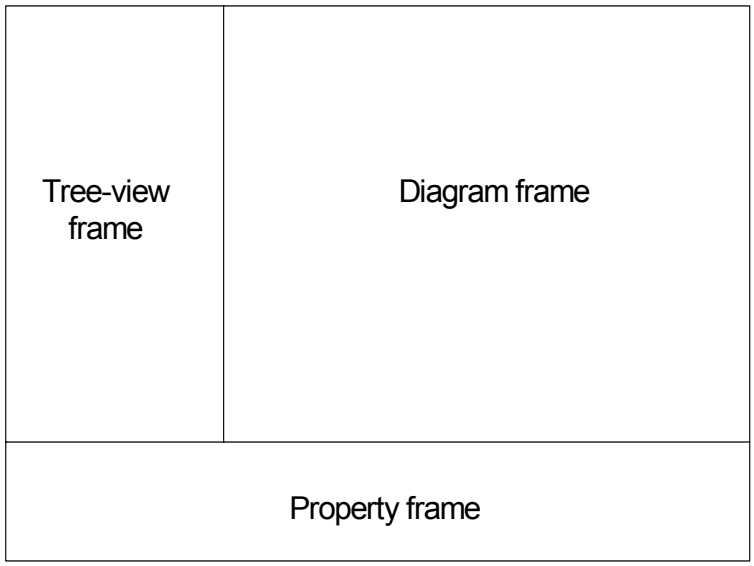



*Figure 291: HTML view appearance.*

**Note**

*The contents of a HTML view is predefined and is not customizable. It does not reflect the active metamodel.*

## Tree-view

The tree-view is a standard tree-view of the model. Each node in the tree represents a UML model element.

Clicking a node in the tree view displays the properties of the element in the property frame. If the element is a diagram, the corresponding diagram is displayed in the diagram frame.

Collapsing and expanding nodes does change the contents of any other frame.

## Properties

The properties of an element is displayed in the property frame. The available set of properties for each element depends on its metaclass.

References to other elements included in the HTML view are navigable.

**Note**

*Each element has a unique set of properties defined by the ModelBrowser addin. The set of properties of an element is not customizable and doesn't neccessarily match the set of properties displayed in Tau.*

## Diagrams

Diagram images are generated as jpeg files. Symbols in some diagrams are clickable if the corresponding model element is included in the report. When clicking a symbol in a diagram, the properties of the model element is opened in the property frame.

# Command line usage

A HTML report can be generated from the command line by passing a script to `vcs.exe` according to the following syntax:

```
vcs -script <etc-path>/HtmlReport.tcl <project-file>
```

where:

`vcs`

The Tau executable in the `bin` folder of the installation

`<etc-path>`

A full or relative path to the `etc` folder of the Tau installation

`<project-file>`

A full or relative path to a project file

The resulting folder is placed in the same folder as the project file.

**Example 802**

This example shows how to generate a HTML report for a project called `MyProject` located in `C:/MyModels/MyProject.ttp`.

First open a command window and make sure your current folder is the `bin` folder of the Tau installation.

Then execute the following command line:

```
vcs -script ../etc/HtmlReport.tcl C:/MyModels/MyProject.ttp
```

A HTML report is generated for the project and placed in the same location as the project file.

---

**Note**

*When using the command line mode a report is generated for the entire project. To generate reports for parts of a project or model use the interactive mode.*

# 90

# *Internationalization Support*

This section describes the internationalization support in Telelogic Tau/Developer and Tau/Architect. The main focus of this document is Chinese, Japanese and Korean (CJK) language handling.

## Supported environments

This section describes specific information for Internationalization support of system environments. The information not described in this section is common through all languages. Please refer to the installation guide for general information.

### Supported platforms

The internationalization support in Tau is available for Windows 2000 and XP. It is assumed that you use a local version of Windows and set the locale to use your local language.

### Configuration Management

Tau does not support CJK environments beyond limitations of each configuration management tool for CJK support.

### DOORS Integration

Tau supports integration with Telelogic DOORS 8.0 and Telelogic DOORS 6.0a for CJK characters.

### IME (Input Method Editor)

Default IMEs bundled in Windows are supported. Using supported IME, you can enter your local characters inline.

## Font settings

By selecting the correct font for your language, your language is displayed correctly.

1. Select **Tools** and then **Options** from the Tau menu bar**.**
2. Select **Format** tab.
3. Choose **Category** and specify font type
   - **Dialog fixed** : the font type setting for dialogs using a fixed width font.
   - **Developer diagram symbol font**: the font type setting for other symbols and diagrams.
   - **Report Windows**: the font type setting for tabs in the Output window.
   - **Output Windows**: the font type setting for Message, Model Verifier and Script tabs in the Output window.
   - **Tcl Files**: the font type setting for Tcl and text files opened in Tau.
   - **C/C++ Header/Source**: the font type setting for C/C++ header and source files opened in Tau.

**Note**

*The instructions presented below should be performed before you start to create elements in your diagrams.*

There is also fonts settings for diagrams elements.

1. Select **Tools** and then **Options** from the Tau menu bar.
2. Select **Font settings** tab.
3. Specify font types. See “Font settings” on page 2460.

**Note**

*You can also change the font style and size for each element from the Diagram element properties toolbar.*

# Modeling with CJK characters

Tau supports modeling with CJK characters. You can use CJK characters for

- names of all elements
- comments
- Charstring literals.

You can type CJK characters in the same way as English characters. No special operation is needed to draw models with CJK characters.

# Code generation with CJK characters

Element names in the model are used as names of identifiers in generated C/C++ files. However, in C/C++ grammar, CJK characters are illegal for identifier names. So, Tau have a mechanism to provide legal names for C/C++ code and it can be done in two ways.

### Automatic UTF-16 naming

If CJK characters are used in an element name, Tau provides the legal name in generated C/C++ code. The provided legal name consists of a UTF-16 big endian encoded hexadecimal string that is prefixed by `tau_` and suffixed by `_tau`

**Example 803: UTF-16 encoded name**

MALMÖ is encoded as

```
tau_004D0041004C004D00D6_tau
```

### Using ansiName stereotype

If you want to use CJK characters for element names and you want to specify legal name for generated C/++ code, you can use the stereotype ansiName.

1. From the **Tools** menu select **Customize**. Go to the Add-Ins tab.
2. Check the **Internationalization** add-in to load ansiName stereotype in the current project. (You need to activate it per project.)
3. Enter the element name in the diagram with CJK characters.
4. In the Model View, right-click the element for which you want to specify a legal name.
5. On the shortcut menu, click **Properties**.
6. From the **Filter** box, select **ansiName**.
7. Type the legal name for the code generation in the **name** field.

When you specify a legal name by ansiName, the specified name is used for C/C++ code.

All UML definitions may be given an alternative name, for the purpose of C or C++ code generation, by means of the ansiName stereotype.

### Names of files and folders used by build tool chain

### Names of class symbols and package symbols

The names of class symbols and package symbols are used as generated file and folder names. If CJK characters are used in symbol names, the encoding mechanism described in "Code generation with CJK characters" on page 2445 is applied to name the generated folders and files. If you want to specify the file and folder names, then use ansiName stereotypes as in "Using ansiName stereotype" on page 2446.

### Comments

Comments will not be presented in generated C/C++ files. This applies to both CJK characters, English characters and any other language.

### Encode type of files used by build tools

The encode type of the files that are processed by the build tool chain (such as the files holding the intermediate format and the generated C/C++ files) are encoded using the **system locale** encoding. For example, on Japanese Windows, the generated files will be encoded using SHIFT-JIS, so that Microsoft Visual C++ can handle and compile the files.

Therefore, you need to set correct locale for your language to generate files with the correct encoding.

1. Open the Windows Control panel
2. Open the **Regional Settings** dialog and select the **General** tab.
3. Make sure that the selected locale is the correct one.

## Handling textual files

Textual files can be opened inside Tau. Tau supports local ANSI encoding and UTF-8 for the textual file. When existing textual files are opened in Tau, Tau saves the files in the original encoding. When the textual file is created in Tau, the file will be saved in UTF-8 by default. You can select encode type from the **Save as** dialog.

## Restrictions

- Single byte Japanese Katakana and Japanese characters defined between 0x80 and 0xFF in Shift-JIS are not supported.
- CJK characters are not supported for Project names.
- Using CJK names for messages and message text parameters may result in displaying UTF-16 big endian encoded names instead of original CJK names, when tracing a Model Verifier in Sequence diagrams. In order to find out the original names, decoding must take place in an external application that supports displaying UTF-16 encoded characters.

**See also**

"Automatic UTF-16 naming" on page 2445.

# 91

# *Dialog Help*

This section lists the help texts that are displayed when you click the help button in dialogs.

# The New Wizard

## Files tab

This dialog provides the possibility to add new files to your design.

- When adding a file you must specify a file name and a location.
- The file can be added to an existing project. The project must be opened in the File View in order for you to add the file to it.
- The new file is opened in the Desktop.

## Projects tab

This dialog provides the possibility to add a new project.

When you add a project, you specify how the project will be used. Depending on your choice, different add-ins will be loaded at start-up, for example:

**UML for AgileC Code Generation**
The AgileCApplication, ModelVerifier and RTUtilities add-ins are loaded.

**UML for C Code Generation**
The CApplication, ModelVerifier and RTUtilities add-ins are loaded.

**UML for C++ Code Generation**
The CppAppGen and CppTypes add-ins are loaded.

**UML for Model Verification**
The ModelVerifier and RTUtilities add-ins are loaded.

**UML for Modeling**
No add-ins are loaded.

Should you later want to change the default add-ins in your project, you can do so using the Add-ins tab (**Tools** menu, select **Customize**).

- When adding a project you must specify a project name and a location.
- The project can be included in the current workspace, or a new workspace can be created for the project.

**See also**

"Working with Projects" on page 35 in Chapter 4, *Introduction to Tau 4.2*

"Add-ins tab" on page 2455

## UML Projects - page 2

This dialog displays a suggested file directory and a suggested name for the file holding the model.

- You can change or confirm the suggestions.
- As an option, an empty package can be added.

## UML Projects - page 3

This dialog displays the name of the project and the name of the related file.

- You can confirm the names by clicking the Finish button or enable changes by clicking the **Back** button.
- The new project appears in the Workspace window.

## Workspaces

This dialog provides the possibility to add a new workspace.

- When adding a workspace you must specify a workspace name and a location.
- The new workspace is loaded in the Workspace window.

**See also**

"Working with Workspaces" on page 33 in Chapter 4, *Introduction to Tau 4.2*

# Customize

## Commands tab

This tab lists the default menus with toolbar buttons, commands and menus that you can add to a toolbar or menu. It allows you to move, delete or add buttons to your toolbars.

1. In the **Categories** box, click the toolbar name that you want to customize.
2. In the **Buttons** area, drag the item from the dialog on to the toolbar. Click the item first to receive information about the specific item.
3. To remove an item from a toolbar, drag the item from the toolbar on to the dialog.

**To add a button to a toolbar:**

1. Make sure that the toolbar you want to change is displayed.
2. In the **Categories** box, the available toolbar buttons or items are grouped. Select the category where the toolbar button or item you want to add is located.
3. Click a button or item to receive information about its functionality.
4. Drag the button or item from the **Buttons** area to the toolbar in the user interface.

**To delete a button from a toolbar:**

1. Make sure that the toolbar you want to change is displayed.
2. Drag the button or item off the toolbar.

When you delete a default button from a toolbar, the button is still available in the Customize dialog box. However, when you delete a toolbar button with a custom appearance, its appearance is permanently lost, although the command is still available (Customize dialog box, Commands tab).

**Hint**

*To save a toolbar button with a custom appearance for later use, create a toolbar for storing unused buttons, move the button to this storage toolbar, and then hide the storage toolbar.*

## Toolbars tab

This tab lists standard and custom toolbars.

Select or clear the check boxes to display or hide the toolbars. Each toolbar appears either in the default location or in the last location that it is moved to. The menu bar cannot be hidden.

**Show Tooltips**
Click the check box to enable tooltips to be displayed when the cursor moves over a button or field in the toolbars.

**Large Buttons**
Click the check box to display larger sized buttons in the toolbars.

**Create a new toolbar:**

1. Click **New**.
2. In the dialog that opens, type the name of the toolbar. The new toolbar appears in the toolbar area of the interface.
3. From the **Commands** tab, select the items that you want to add to the toolbar.

**Restore the default toolbar settings:**

1. Click the toolbar in the list.
2. Click **Reset**.

A user-created toolbar cannot be restored.

**Delete a user-created toolbar:**

1. Click the toolbar in the list.
2. Click **Delete**.

A default toolbar cannot be deleted.

**Rename a user-created toolbar:**

1. Click the toolbar in the list.
2. In the **Toolbar Name** field, type a new name for the toolbar.
3. Click the toolbar again to save the change.

## Create New Toolbar

Type the name of the new custom toolbar. You can use upper or lower case letters, but each name must be unique regardless of case. The name must be unique from other toolbars. If you want to change this name later, you can edit the name in the Toolbar Name box on the Toolbars tab.

## Windows layouts

This tab allows you to customize the appearance of the Windows layout. You can save toolbar positions, visibility and location of docked windows.

**Save a new layout:**

1. Click the New button.
2. Type a name for your layout.
3. Close the window.

**To restore a new layout:**

1. Click the layout you want to restore.
2. Click Restore.

**To delete a layout:**

1. Click the layout you want to delete.
2. Click the Delete button.

## Tools tab

This tab allows you to add commands in the Tools menu. These commands can be associated with any program that runs on your operating system. The information is saved in a file named Tools.dat in the directory:

```
C:\Documents and Settings\<user>\Application Data\Telelogic\Shared
```

**Add a command to the Tools menu:**

1. Click the **New (Insert)** button. A blank line, indicated by an empty rectangle, appears in the **Menu Contents** box.
2. Type the name of the command as it will appear in the Tools menu. Press **ENTER** to save the name.

3. In the **Command** field, type the path to the program. You can also use the browse button to locate the program.
4. In the **Arguments** text box, browse or type any arguments to be passed to the program. Use the drop-down arrow next to the Arguments text box to display a menu of arguments.
5. In the **Initial directory** box, browse or type the file directory where the command will be located.
6. If the program is a console program, for instance the Windows command prompt, you can select to have it run in the Output window. Just select the **Use Output Window** check box.
7. Select the **Prompt for Arguments** check box, if you want to be able to change argument each time you want to use the command.
8. Select the **Use OEM format** check box, if you want to the application's output to be in OEM format.
9. Click **OK**. The command appears in the Tools menu.

**Additional tasks**

- To insert the command in a submenu, separate the menu name and the name with a backslash '\'. For instance, the command Notepad in an editor menu should be typed `editor\Notepad`.
- To insert an access key, type an ampersand '&' before the selected letter in the name.
- Move commands up and down in the menu by using the **Move Up** and **Move Down** buttons.
- To change the name of the command, double-click it and type a new name.

**Delete a command in the Tools menu:**

1. Click the command in the list.
2. Click the **Delete** button.

## Add-ins tab

Add-ins are used to extend the tool functionality. From the Add-ins tab you can load a selection of predefined add-ins.

The different add-ins are optimized for different build situations. This means that you often do not have to change any build settings. When you click an add-in, you will see its usage in the description field.

- To load or unload add-ins, select or clear the check boxes. Close the dialog.
- You can modify the available add-ins. Click the add-in and click **Modify**. The dialog that opens allows you to customize the add-in according to your needs.
- You can also create your own add-ins by clicking **Create**. The dialog that opens allows you to configure the add-in according to your needs.

See also

"Contents and structure of an add-in" on page 1988 in Chapter 73, *Customizing Telelogic Tau*

# Options

## General

This tab allows you to set general options:

**Display status bar**
Allows you to show or hide the status bar that is available at the bottom of the Tau user interface.

**Show output window when receiving content**
When the Output window is closed, information that is normally listed in the different tabs is not displayed. However, when selecting this option, the output window will open automatically when new information is listed, for instance after a manual check.

**Track selection in the Print Manager**
The Print Manager by default tracks the active selection in the Model View. This option can be turned off to disable this tracking.

**Show advanced option page**
Select this option to display an additional tab with all options listed in a tree structure. Some advanced option can only be set from this additional tab.

**Tabbed documents**
Select this option to open documents in a single window as tabs.

**Show welcome page at startup**
This option controls whether or not the welcome page should be opened when starting the tool. This option can also be set from the welcome page itself. If you turn this option off you can open the welcome page manually from the Help menu.

**Source control provider**
If you have a source control system installed, you can use this option to enable using it from Tau. Doing so will enable a source control menu and toolbar for interaction with your source control system. For more information see Configuration Management.

**Automatically update files**
This option can be used if Generic Source Control is selected as source control provider. If the option is enabled files will be automatically updated from the CM system before attempting to do a check out.

**Disable external program launch for these types of file**
In this field you can specify the extension of files, that Tau should attempt to open instead of the external application which otherwise is associated with that file extension. For instance, if you add the `*.txt` extension, text files will be opened in the Tau text editor instead of in your external text editor application.

**Select the default help context**
If there are many Telelogic tools installed, you can choose which help file to use as default by selecting the file in this list.

**URN Map**
Use the **URN Map** (Universal Resource Name) to define shorthand names for file storage locations. For example:

```
home:C:\MyHomeDir;work:C:\MyWorkDir
```

Here "`home`" is shorthand for `C:\MyHomeDir` and "`work`" is shorthand for `C:\MyWorkDir`. Each user may define URNs for his/her environment. These are used by some components for referring to files, bitmaps and other resources.

There are five predefined names, "`u2`", "`u2user`", "`u2useraddins`", "`u2teamaddins`" and "`u2companyaddins`".

- `u2` maps to the Tau installation directory
- `u2user` maps to the user directory
- `u2useraddins` maps to where user add-ins are located
- `u2teamaddins` maps to where the team add-ins are located
- `u2companyaddins` maps to where the company add-ins are located

**Example 804: URN file references (usages of URNs)**

Using the name/directory mappings defined above a URN file reference of:

```
urn:home:mybitmap.bmp
```

would expand to:

```
C:\MyHomeDir\mybitmap.bmp
```

The reference:

```
urn:u2:etc\TTDMetamodel.u2
```

might translate to

```
C:\Program Files\Telelogic\TAU_4.2\etc\TTDMetamodel.u2
```

if the installation directory is:

```
C:\Program Files\Telelogic\TAU_4.2
```

These URNs can for example be used in referring to add-in etc files, images in icon references and for encoding hypertext references.

## Save

This tab allows you to set save options in Tau.

**Save before running tools**
Select this option to automatically save any unsaved work before an external tool is launched.

**Prompt before saving files and projects**
Select this option to be prompted for saving when modified files and projects exist when an editor is closed.

**Automatic reload of externally modified files**
You will by default receive an information message and be prompted to reload an externally modified file. Select this option to avoid this prompting, in order to automatically reload a file that has been modified in another tool than Tau.

**Save project's add-in state in all the loaded projects**
This option will let any loaded add-ins become activated for all projects currently loaded.

**Auto-backup**
Select the **Activate** check box to allow automatic saves of your model in predetermined intervals. Enter the desired number of minutes between the saves, either by typing the number or by clicking the up and down buttons.

## Workspace

This tab allows you to set general options for the workspace that you have opened.

**Reload last workspace at startup**
Select this option to open the workspace that you were working in the last time Tau was running.

**Warn on project file status change**
Select this option to receive a warning if the status of the project file you are working in has been changed to read-only. This will protect you from potentially loosing unsaved work.

**Projects default location**
When you create new projects, you will receive a suggestion where the project file will be stored. In this text field, type a path, or browse to a folder, where the new projects will be stored.

## Format

This tab allows you to format the appearance of text and colors in windows and files.

When you have selected a category, you can select:

- **Font** and **Size** of the text in the file or window.

- Background color and text color for the selected Category. By default, system colors defined in the control panel are used. Clear the **Automatic** check boxes to select text and background colors.

## Font settings

This tab allows you to customize the default fonts used when creating a new diagram.

**Diagram font settings**
These font settings determine the default text appearance of the generic diagram element in a created diagram.

**Fixed font settings**
These font settings determine the default text appearance of symbols that have texts which are better displayed using a fixed width font. An example of a symbol that uses this font setting is the Text symbol. If the **Enabled** check box is checked, any created symbol of this kind will have the fixed font settings applied on it.

**Label font settings**
These font settings determine the default text appearance of text labels that are not the main label of a diagram element. An example of this is the Attribute and Operation labels in a Class symbol. If the **Enabled** check box is checked, any created label element will have the label font settings applied on it.

## Links

This tab allows you to customize link creation behavior.

**Active link end is an active target, not an active source**
If this option is off, then when you use automatic creation of links, you will create links from your active link end to the other models. If this option is on, then you will create links to your active link end from the other models.

**Automatically create links between modified objects and active link end**
If this option is on, then when you select an active link end, all your modifications will be linked to this link end.

**Show link indicators**
If this option is on, Tau will show the link markers.

**Use requirement as target when creating links by drag and drop**
Links can be created using drag and drop. If this option is on when doing this on a requirement the target of the link will be the requirement. If the option is off the requirement will instead be the source of the link.

# UML Basic Editing

This tab allows you to set basic preferences for the diagram editors, the properties editor and the Model View.

### Diagram Editors

**Show grid**
Select this option to make a grid visible in the editors. This option is global, which means that the grid will be available in all diagrams. To show or hide the grid for a single diagram, right-click the open diagram and click **Show Grid**.

**Show page breaks**
Select this option to display the bounds of each page when the diagram is printed. This option is global, which means that the page break indicators will be available in all diagrams. To show or hide the page break indicator grid for a single diagram, right-click the open diagram and click **Show Page Bounds**.

**Delete model element when deleting last symbol/line**
When this option is selected, a model element is automatically deleted when its last presentation element (symbol or line) has been deleted. When de-selected all model elements without presentations must be manually deleted, if so is desired.

**Show dialog when deleting last reference symbol/line for a model element**
When this option is selected, a warning is displayed when a model element is about to be deleted as a result of using the "Delete model element when deleting the last symbol/line" option.

**Remember scroll and zoom**
When this option is selected, Tau will remember zoom level and scroll position of open diagrams between different sessions of using the tool. This information is saved in a file named `projectname_DiagramSettings.u2x`. Removing this file does not affect the model.

**Show stereotypes on attribute and operation labels**
If this option is enabled, stereotypes applied on attributes and operations will be displayed in the attribute and operation labels of class-like symbols (class symbol, interface symbol etc.).

### Default Model View Filters

In this area, you decide what objects that will be displayed by default in the Model View of the workspace window. These settings are global. If you want to change the settings for a specific project, use the **View - Model View Filters** menu.

**Sort definitions**
If this option is enabled, the order of definitions shown in the Model View will be sorted on their names in a lexicographical order (‘A’ to ‘Z’, ‘a’ to ‘z’, etc.).

### Default Model View

This drop down list allows you to switch from the standard view of the Workspace window. This setting is global. If you want to change the view for a specific project, use the **View - Reconfigure Model View** menu.

### Default Property View

This drop down list allows you to specify which property view to use in the Properties Editor by default. This option affects all new Properties Editors that are opened. To switch property view of an already opened Property Editor, press the **Options...** button in that particular Property Editor.

### Default Property Filter

This drop down list allows you to specify which filter to use in the Properties Editor by default. This option affects all new Properties Editors that are opened. To switch filter of an already opened Property Editor, press the **Options...** button in that particular Property Editor.

### Default Class Symbol Appearance

These settings will affect the initial appearance of class-like symbols (class symbol, interface symbol etc.) when they are drawn in editors. This affects both newly created symbols and symbols that you drag from the Model View to a diagram.

**Collapsed**
Show only the top compartment of the symbol.

**Show attributes**
All existing attributes will be shown, if the symbol is not collapsed.

**Show operations**
All existing operations will be shown, if the symbol is not collapsed.

**Show ports**
All existing ports will be shown.

**Show stereotype compartment**
All applied stereotypes will be shown in a stereotype compartment.

**Show constraint compartment**
All constraints will be shown in a constraint compartment.

## UML Advanced Editing

This tab allows you to set advanced preferences for the diagram editors and the Model View.

### Text

**Indent level**
If the **Automatic indention** option is selected, this option defines the number of indent spaces that are inserted after the characters '{' and ')'. It also specifies the number of positions between tab stops.

**Color highlighting in text**
When this option is selected, semantic and syntactic highlights are displayed. The highlighting syntax is described in "Text Highlighting" on page 80 in Chapter 6, *Working with Models*.

Pointing the mouse at a highlighted name indicates the current status of the text in a tool tip, and in particular diagnose the situation in some way. For instance: “This reference is currently not bound, see Autocheck log for details”.

**Syntax error highlighting in text**
When this option is selected, syntactic errors are displayed in different colors indicating the type of situation. The highlighting syntax is described in “Text Highlighting” on page 80 in Chapter 6, *Working with Models*.

Pointing the mouse at a syntax error marker will show more information about the syntax error in a tool tip.

**Automatic indention**
When this option is enabled, indentation spaces are automatically inserted on lines following a ‘{’ or ‘)’. The number of spaces to be inserted is specified in the **Indent level** option.

**Automatic matching of left parentheses and quotes ((["‘)**
Automatically adds closing parenthesis or quote signs immediately after opening parenthesis or quote.

**Disable model update while editing**
When this option is set, the model is updated only when you choose to leave text edit mode, for example when you make a new selection outside the label/text symbol. The text will not be merged into the model until you leave the edit mode, irrespective of for how long you are editing the text.

**Sequence diagrams**

**Message separation**
Determines the vertical distance (in millimeters) between messages.

**Lifeline separation**
Determines the horizontal distance (in millimeters) between lifelines.

**Dock sequence diagram trace window**
Determines if the sequence diagram trace window should open as a docked window or not. This option is enabled by default.

### Activity diagrams

**Autocreate**
Determines the default Flow orientation in Activity diagrams. This option controls how autocreated symbols in an Activity diagram get positioned.

### Symbol appearance

**Default symbol color**
Determines the default color of symbols in diagrams. To change the color, click the color field and select a new color from the palette or choose a custom color.

**Port symbol arrows**
Displays arrows inside the port symbols to indicate the enabled communication flow. Directions are shown with reference to the port location on a vertical or horizontal border line.

### Diagram tooltips

**Show symbol and line tooltips**
Enables tooltips for symbols and lines. The tooltips provide various kinds of information about the model presented by a symbols or line, for example comments and error messages.

**Show edit mode tooltips**
Enables tooltips during editing.

### Name completion

**Display name completion window after typing '.' or '::'**
This option controls if the name completion window should be opened when you type a member access ( . ) or qualifier ( : : ) delimiter. The name completion window will then provide you with the names of possible definitions that can be referred to from the current textual context.

### Automatic update of name-based references

**When moving definitions in Model View**
If this option is enabled all references to a definition will be updated when the definition is moved to a new location in the Model View. This option is

by default enabled, but can be disabled if you prefer to update the references yourself (or if they should not be updated, but instead bind to another definition with the same name as the moved one).

## UML Editing Line Styles

Options in this tab control the default line shapes used for different line types in the editors. There is one option for each available UML line kind. The values have the following meaning:

**Auto-routed (assign endpoints)**
Line vertices are automatically positioned to avoid obstacles and will try to keep line orthogonal. Endpoints are automatically assigned.

**Auto-routed (keep endpoints)**
Line vertices are automatically positioned to avoid obstacles and will try to keep line orthogonal.

**Bezier**
Will give the line a curved layout. When the line is selected two control points are displayed which can be used to shape the curve.

**Orthogonal**
The line is always kept orthogonal and line vertices and segments can be moved. Vertices can be added and removed from the line.

**Non-orthogonal**
Line vertices can be moved, added and removed without restrictions.

**Note**

*To change the layout for a single line, right-click the line and select between the different line styles. The current line style has an active radio button.*

## UML Checking

This tab allows you to enable different types of checks performed during manual and automatic checks.

**AutoCheck**

These options control which checks that should be performed automatically while editing the model.

**Check diagrams**
Enables the reporting of syntactical errors in the diagrams.

**Enable simple semantic checks**
Enables certain simple semantic checks, which can help you find problems in the edited model quickly.

**Show binding errors**
Reports identifiers that cannot be resolved.

**Check**

**Enable checking of diagrams**
When enabled, syntactical errors in diagrams are reported. Other errors where the presentation is inconsistent with the model will also be reported.

**Rebind references**

If this option is turned on (default), Tau will attempt to rebind references in the model automatically during editing of the model. Rebinding takes place when necessary in order to ensure that all bindings are accurate and up-to-date at any time while editing. Turn this option off if you want to postpone rebinding until the **Check All** command is used on the model. Doing so may improve the tool performance significantly when editing large models.

# Hyperlink

This tab allows you to change hyperlink behavior.

**By default, make hyperlinks to a workspace element**
When this option is enabled, Tau allows you to select the target of your hyperlink within your workspace. Otherwise, you are prompted to specify another type of target, such as an existing file or a web page.

# Compare/Merge

This tab allows you to change options for Compare/Merge.

**External text compare/merge**

The default values for these options are compatible with the textual compare/merge tool of Synergy CM if installed.

### External text compare/merge tool path

Path to the external text compare/merge tool that can be used from the Review Differences dialog.

### Command line switches

Depending on the compare/merge usecase, the external compare/merge tool will be called with the corresponding command line switches.

### Save review information

### Save differences list image generation

These options specify the width and height (in pixels) of the generated images.

### Save stereotype instances and comments as unparsed text

If this option is enabled, comments and stereotype instances with no graphical representation are added as text in the generated XML file.

## Advanced

The Advanced tab allows you to change the values for all available options using a tree control. The tree consists of option categories with the actual option as leafs. Here you will find both the options that are available in the other option tabs, as well as some advanced options that are only available in this tab. Expand the tree control to find the option which you want to change. To change the option value, select the value and click F2.

Some of the advanced options are documented below.

### Web server

```
Studio - Settings - WebServer
```

The options **PortRangeBegin** and **PortRangeEnd** define the range of TCP/IP ports used by the Tau Web Server. You may need to change these options if the default port numbers are not available for use on your machine.

### Proxy settings

```
U2 - Options - ProxySettings
```

The options **Host**, **Password** and **User** can be set if you access the web through an HTTP proxy server. They will be used whenever Tau accesses information from an URL, for example when importing a WSDL file from an URL. The syntax of the host option is `<address>:<port>`.

# Editor Shortcut

## Show Elements

This dialog provides the possibility to add multiple elements to a diagram with a selection of symbols from your existing model.

- Elements are selected by checking the check box in the element list.
- The element list contains the elements of the current set scope.
- The Set Scope button is used to add elements from any scope in your model to the element list.

When this dialog is entered as a result of an operation where an element is initially selected this element will be pre-checked in the list.

- The new diagram is opened in the Desktop.

# Models

## Reconfigure ModelView

Select the browser model that you want to use. There are two predefined browser views. The browser view “Standard View”, gives a comprehensive view of the loaded model including design detail. This view is intended for design-oriented users.

The other browser view “Diagram View” gives a simplified view of the loaded model. This view is intended for analysis-oriented users.

**See also**

“Metamodel” on page 386 in Chapter 8, *UML Language Guide*

# Other

## Select Stereotypes

Select the stereotypes that you want to apply to the element. Click each line to see a description of the each stereotype. The number of applicable stereotypes varies depending on the selected element.

**See also**

"Stereotype" on page 387 in Chapter 8, *UML Language Guide*

## Select artifact root

Select the class that you want to use as your Build Root.

**See also**

"Artifact" on page 372 in Chapter 8, *UML Language Guide*

# Model Verifier

## Console Windows

Model Verifier Console Window

## Message Windows

Message Window

## Restart

Restart: Click to restart the execution after that the Break button has been clicked

## Stop Model Verifier

Stop: Click to stop the execution and to shut down the Model Verifier

# 92

# *Additional Resources*

This section list documents that are not part of the help file, but that may help you to extend your knowledge about Tau. Links to useful web resources are also provided.

# Links

## Contacting IBM Rational Software Support

Support and information for Telelogic products is currently being transitioned from the Telelogic Support site to the IBM Rational Software Support site. During this transition phase, your product support location depends on your customer history.

### Product support

- If you are a heritage customer, meaning you were a Telelogic customer prior to November 1, 2008, please visit the Tau Support Web site. Telelogic customers will be redirected automatically to the IBM Rational Software Support site after the product information has been migrated.
- If you are a new Rational customer, meaning you did not have Telelogic-licensed products prior to November 1, 2008, please visit the IBM Rational Software Support site.

Before you contact Support, gather the background information that you will need to describe your problem. When describing a problem to an IBM software support specialist, be as specific as possible and include all relevant background information so that the specialist can help you solve the problem efficiently. To save time, know the answers to these questions:

- What software versions were you running when the problem occurred?
- Do you have logs, traces, or messages that are related to the problem?
- Can you reproduce the problem? If so, what steps do you take to reproduce it?
- Is there a workaround for the problem? If so, be prepared to describe the workaround.

### Other information

For Rational software product news, events, and other information, visit the IBM Rational Software Web site.

## UML documents

- **Java Tutorial**

  This tutorial teaches you the basics of working with the Tau product in a Java coding environment, and introduces the concepts of requirements analysis and project implementation.

  The tutorial is available in your installation in:

  `locale/en/tutorialjava.pdf`
- **UML Tutorial**

  This basic tutorial goes through the design of a UML model. It covers the various type of diagrams and how to use Tau to check and verify the model.

  The tutorial is available in your installation in:

  `locale/en/coffmach.pdf`
- **UML Quick reference guide**

  This document contains common graphical and textual constructs in UML.

  The guide is available in your installation in:

  `locale/en/quickref.pdf`

## Other links

### Borland C/C++

C/C++ dialect supported by the Borland builder.
http://www.borland.com/cbuilder

### Cygwin

For information about the contents of various Cygwin versions, see:
http://www.cygwin.com

### GNU C/C++

C/C++ dialect supported by the GNU Compiler Collection.
http://www.gnu.org/software/gcc

### ITU-T

Formerly CCITT
http://www.itu.int/

### Macrovision

For more information about FLEXnet or Macrovision, please see:
http://www.macrovision.com

### Microsoft Visual C/C++

C/C++ dialect supported by Microsoft Visual C++:
http://msdn.microsoft.com/visualc

### MISRA

The code generated by the AgileC Code Generator is to a large extent compliant with the MISRA coding rules described in the document "MISRA-C:2004 Guidelines for the use of the C language in critical systems" from October 2004. Please see:

http://www.misra.org.uk

### OCL

For more information about OCL (Object Constraint Language), see:
http://www.omg.org

### OMG

For more information about Object Management Group (OMG), see:
http://www.omg.org

### PDF

PDF files are opened and read with Adobe Acrobat Reader:
www.adobe.com

### Tcl

For detailed information refer to the Tcl Developer Site
http://tcl.activestate.com/

### TTCN-3

The TTCN-3 standard can be downloaded from
http://www.etsi.org

### XML

For information about Extensible Markup Language (XML), see:
http://www.w3.org/XML

# Copyrights 1



# Introduction 13

15

## Introduction to Tau 4.2 15

# UML Modeling 73

75

## Working with Models 75

# Working with Diagrams 169

199

## UML Language Guide 199

# UML for Model Verification 441

443

## Verifying an Application 443

# 545

## C/C++ Import 545

711

## Rose Import 711

# 729

## Together Import 729



# 743

## UML 1.x Import 743

763

## UML 1.x Export 763



781

## CORBA IDL Exporter 781

931

## Building Applications Reference 931

961

## Stereotypes for Code Generation 961

# UML for C Code Generation

# 1031

# 1081

## Dynamic Memory Management in C Code Generator 1081



# 1089

## C Code Generator Reference 1089

# 1269

## AgileC Code Generator Reference 1269

1319

## C Compiler Driver 1319



# UML and Java 1327

1329

## Java Support 1329

1469

## Eclipse Integration 1469

# UML and C# 1481

## 1483

## C# support 1483

# UML and C++ 1513

# 1533

## C++ Application Generator Reference 1533

# 1725

# Working together with DOORS 1725

# UML Modeling with System Ar-

# chitect 1793



# UML and Web Services 1809

1823

## WSDL Generator Reference 1823

1843

## WSDL/XSD Importer Reference 1843



# XML Schema Modeling with UML 1891

1893

## Modeling XML Schemas 1893

# Exploring UML Models 1899

## 1901

## The Tau Explorer 1901

# Customizing Tau 1979

## 1981

## Customizing Telelogic Tau 1981

# 2023

## Predefined Stereotypes and Attributes 2023

# 2025

## Agents 2025

# 2061

## COM API 2061

2185

## Tcl API 2185

# 2245

## C++ API 2245



# 2285

## Java API 2285

2337

## Tau Access 2337

# 2349

## XML Framework Library 2349



# 2357

## Tau Web Server 2357

# Common Reference 2369

## 2371

### Useful Shortcut Keys 2371



## 2383

### Setting Up the Tool Environment 2383

# 2421

## Visual Studio Integration 2421



# 2431

## Printing 2431



# 2437

## Model Browser 2437

2471

## Additional Resources 2471

Chapter :

# *Index*

## Symbols



## Numerics



## 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