

Rational® Test RealTime

REFERENCE MANUAL

VERSION: 2002 RELEASE 2 - SR1

IMPORTANT NOTICE

COPYRIGHT

Copyright ©2000-2002, Rational Software Corporation. All rights reserved.

Part Number: 800-025997-000

Version: 2002 Release 2 - SR1

PERMITTED USAGE

THIS DOCUMENT CONTAINS PROPRIETARY INFORMATION WHICH IS THE PROPERTY OF RATIONAL SOFTWARE CORPORATION (“RATIONAL”) AND IS FURNISHED FOR THE SOLE PURPOSE OF THE OPERATION AND THE MAINTENANCE OF PRODUCTS OF RATIONAL. NO PART OF THIS PUBLICATION IS TO BE USED FOR ANY OTHER PURPOSE, AND IS NOT TO BE REPRODUCED, COPIED, ADAPTED, DISCLOSED, DISTRIBUTED, TRANSMITTED, STORED IN A RETRIEVAL SYSTEM OR TRANSLATED INTO ANY HUMAN OR COMPUTER LANGUAGE, IN ANY FORM, BY ANY MEANS, IN WHOLE OR IN PART, WITHOUT THE PRIOR EXPRESS WRITTEN CONSENT OF RATIONAL.

TRADEMARKS

Rational, Rational Software Corporation, Rational the software development company, ClearCase, ClearQuest, Object Testing, Purify, Quantify, Rational Apex, Rational Rose, Rational Suite, among others, are either trademarks or registered trademarks of Rational Software Corporation in the United States and/or in other countries. All other names are used for identification purposes only, and are trademarks or registered trademarks of their respective companies.

Microsoft, the Microsoft logo, Windows, Windows NT, Windows Me and Windows 2000 are trademarks or registered trademarks of Microsoft Corporation in the United States and other countries.

FLEXlm and GLOBEtrotter are trademarks or registered trademarks of GLOBEtrotter Software, Inc. Licensee shall not incorporate any GLOBEtrotter software (FLEXlm libraries and utilities) into any product or application the primary purpose of which is software license management.

PATENT

U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329 and 5,835,701. Additional patents pending.

Purify is licensed under Sun Microsystems, Inc., U.S. Patent No. 5,404,499.

GOVERNMENT RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in the applicable Rational Software Corporation license agreement and as provided in DFARS 277.7202-1(a) and 277.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (Oct. 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 227-14, as applicable.

WARRANTY DISCLAIMER

This document and its associated software may be used as stated in the underlying license agreement. Rational Software Corporation expressly disclaims all other warranties, express or implied, with respect to the media and software product and its documentation, including without limitation, the warranties of merchantability or fitness for a particular purpose or arising from a course of dealing, usage, or trade practice.

Reference Manual

Contents

About Online Documentation.....	11
Documentation Updates and Feedback	11
Technical Support.....	12
Command Line Reference	15
Graphical User Interface	16
Trace Receiver	18
Test Process Monitor	21
Dump File Splitter	25
C and C++ Instrumentor.....	27
C and C++ Instrumentation Launcher.....	41
Java Instrumentor	46
Java Instrumentation Launcher.....	53
Java Instrumentation Launcher for Ant.....	57
JVMPI Agent	61
Ada Instrumentor.....	66
Ada Link File Generator	73
Ada Unit Maker.....	76
Code Coverage Report Generator	79
TDF Splitter	84
C Source Code Parser	86
C Test Script Compiler	90
C Test Report Generator.....	97
C++ Source Code Parser.....	101
C++ Test Script Compiler.....	107

C++ Test Report Generator.....	109
Java Source Code Parser.....	110
Java Test Report Generator	113
Ada Source Code Parser.....	115
Ada Test Script Compiler.....	120
Ada Test Report Generator	127
System Testing Script Compiler	131
System Testing Report Generator.....	138
Probe Processor	142
System Testing Supervisor.....	144
Virtual Tester	147
System Testing Load Report Generator.....	149

Component Testing Script Languages 151

C Test Script Language	151
C Test Script Language Identifiers	152
C Test Script Language Keywords	153
BEGIN.....	155
COMMENT	156
DEFINE STUB	158
ELEMENT...END ELEMENT	160
ENVIRONMENT ... END ENVIRONMENT	162
FAMILY	164
FORMAT.....	166
HEADER.....	169
IF ... ELSE ... END IF.....	170
INCLUDE	172
INITIALIZATION ... END INITIALIZATION	174
NEXT_TEST	175
SERVICE ... END SERVICE	177
SIMUL ... ELSE_SIMUL ... END SIMUL.....	178

STUB.....	180
TERMINATION ... END TERMINATION.....	184
TEST ... END TEST	185
USE	187
VAR, ARRAY and STR	189
VAR, ARRAY and STR <variable> Parameter	191
VAR, ARRAY and STR <initialization> Parameter	193
VAR, ARRAY and STR <expected> Parameter	197
C++ Test Script Language.....	201
C++ Test Script Language Identifiers.....	202
Native Code.....	205
Error Handling	207
C++ Contract Check Scripts.....	208
Template Classes	209
Inheritance.....	210
C++ Test Script Keywords	211
PROLOGUE.....	214
EPILOGUE	216
PROC	218
RUN.....	220
TEST CLASS	222
TEST CASE	224
TEST SUITE.....	226
INCLUDE.....	228
PROPERTY.....	229
STUB.....	230
REQUIRE	233
ENSURE	235
CHECK.....	238
CHECK EXCEPTION.....	239
CHECK METHOD	241

CHECK PROPERTY	243
CHECK STUB.....	244
ON ERROR	246
PRINT	249
COMMENT	250
CALL	251
CLASS and SINGLE CLASS	252
WRAP	255
INVARIANT	257
STATE	259
TRANSITION ... TO	261
Ada Test Script Language	262
Ada Test Script Language Identifiers.....	263
Ada Test Script Language Keywords.....	264
BEGIN.....	266
COMMENT	268
DEFINE STUB ... END DEFINE	270
ELEMENT ... END ELEMENT	272
EXCEPTION	274
FAMILY	276
HEADER.....	278
IF ... ELSE ... END IF.....	279
INCLUDE	281
INITIALIZATION ... END INITIALIZATION	283
NEXT_TEST	284
SERVICE ... END SERVICE	286
SERVICE_TYPE.....	288
SIMUL ... ELSE_SIMUL ... END SIMUL.....	290
STUB	292
TERMINATION ... END TERMINATION	296
TEST ... END TEST	297

VAR, ARRAY, and STR	299
VAR, ARRAY and STR <variable> Parameter	301
VAR, ARRAY and STR <expected> Parameter	302
VAR, ARRAY and STR <initialization> Parameter	307
Java Test Primitives	311
assertEquals()	314
assertNotNull()	317
assertNull()	318
assertSame()	319
assertTrue()	320
fail()	321
verifyEquals()	322
verifyNotNull()	326
verifyNull()	327
verifySame()	328
verifyTrue()	329
verify()	330
verifyApproxEquals()	332
verifyGreaterThan()	334
verifyGreaterThanEquals()	336
verifyLowerThan()	338
verifyLowerThanEquals()	340
verifyLogMessage()	342
verifyLogfail()	343
createTimer()	344
timerStart()	345
timerReportElapsedTime()	346
verifyElapsedTime()	348
System Testing Script Languages.....	351
System Testing Language for C	351

System Testing Language Identifiers.....	352
Test Script Keywords	353
ADD_ID.....	356
CALL	357
CALLBACK ... END CALLBACK	358
CASE ... IS ... WHEN ... WHEN OTHERS ... END CASE.....	361
CHANNEL.....	363
CLEAR_ID	364
COMMENT	365
COMMTYPE	366
DECLARE_INSTANCE.....	367
DEF_MESSAGE.....	368
END	370
ERROR.....	372
EXCEPTION ... END EXCEPTION	373
EXIT	375
FAMILY	376
FLUSH_TRACE.....	377
FORMAT	378
HEADER.....	380
IF...THEN...ELSE.....	381
INCLUDE	383
INITIALIZATION ... END INITIALIZATION	384
INSTANCE ... END INSTANCE.....	385
INTERRECV	387
INTERSEND	389
MATCHED	391
MATCHING.....	393
MESSAGE.....	395
MESSAGE_DATE	396
NIL	397

NO_MESSAGE	398
NONIL	399
NOTMATCHED	400
NOTMATCHING	402
PAUSE	404
PRINT	405
PROC ... END PROC	406
PROCSEND	408
RENDEZVOUS	410
RESET	411
SCENARIO ... LOOP ... END SCENARIO	412
SEND	414
SHARE	415
TERMINATION ... END TERMINATION	416
TIME	417
TIMER	418
TRACE_OFF	419
TRACE_ON	420
VAR	421
VIRTUAL CALLBACK	423
VIRTUAL PROCSEND	426
WAITTIL	428
WHILE ... END WHILE	430
WTIME	431
Macro Keywords	431
ATL_OCCID	431
ATL_TIMEOUT	433
Supervisor Script Keywords	433
CHDIR	435
COPY	437
DELETE	439

DO.....	440
ENDOF	442
ERROR	444
EXECUTE	446
EXIT	448
HOST	449
IF ... THEN ... ELSE ... END IF.....	451
INCLUDE	452
MEMBERS.....	453
MKDIR	454
PAUSE.....	455
PRINT	456
PRINTLN	457
RENDEZVOUS.....	458
RMDIR	459
SET	460
SHELL.....	462
STATUS.....	463
STOP	465
TIMEOUT.....	466
TRACE ... FROM	467
UNSET	468
WHILE.....	469
Variables.....	470
Environment variables	471
Expressions	472
System Testing Probe Macros	473
atl_dump_trace().....	475
atl_end_trace().....	476
atl_format_trace()	477
atl_recv_trace().....	478

atl_select_trace().....	479
atl_send_trace()	480
atl_start_trace()	481

Appendices..... 482

GUI Macro Variables	482
Global Variables	483
Node Attribute Variables.....	485
Instrumentation Pragmas	486
Environment Variables	489
Setting Environment Variables	494
File Types	495

About Online Documentation

1

The entire documentation set for this product is provided as a full-featured online Help system.

Depending on the operating system you are using, this documentation was designed to be viewed with either:

- Microsoft's HTML Help browser for Windows.
- Netscape Navigator 4.5 or later on other operating systems.

Both environments provide contextual-Help from within the application, a full-text search facility, and direct navigation through the Table of Contents and Index panes on the left side of the Help window.

We welcome any feedback regarding this documentation.

Documentation Updates and Feedback

Documentation Updates

For the most recent documentation updates, please visit the Product Support section of the Test RealTime Web site at:

<http://www.rational.com/products/testrt>

Feedback

We do our best to provide you with first-rate user documentation, so your

feedback is essential for us to improve the quality of our products. If you have any comments or suggestions about our online documentation, feel free to contact us at techpubs@rational.com.

Keep in mind that this e-mail address is only for documentation feedback. For technical questions, please contact Technical Support.

Technical Support

When contacting Rational Technical Support, please be prepared to supply the following information:

- **About you:**
Name, title, e-mail address, telephone number
- **About your company:**
Company name and company address
- **About the product:**
Product name and version number (from the **Help** menu, select **About**).
What components of the product you are using
- **About your development environment:**
Operating system and version number (for example, Windows NT 4.0, Solaris 2.5.1/2.6/2.7, or HP-UX 10.20) Target compiler, operating system and microprocessor. If necessary, send the Target Deployment Port file
- **About your problem:**
Your service request number (if you are calling about a previously reported problem)
A summary description of the problem, related errors, and how it was made to occur
Please state how critical your problem is
Any files that can be helpful for the technical support to reproduce the problem (project, workspace, test scripts, source files). Formats accepted

are **.zip** and compressed tar (**.tar.Z** or **.tar.gz**)

If your organization has a designated, on-site support person, please try to contact that person before contacting Rational Technical Support.

You can obtain technical assistance by sending e-mail to just one of the e-mail addresses cited below. E-mail is acknowledged immediately and is usually answered within one working day of its arrival at Rational. When sending an e-mail, place the product name in the subject line, and include a description of your problem in the body of your message.

Note When sending e-mail concerning a previously-reported problem, please include in the subject field: "**[SR#<number>]**", where <number> is the service request number of the issue. For example:

Re: [SR#12176528] New data on Rational Test RealTime
install issue

Sometimes Rational technical support engineers will ask you to fax information to help them diagnose problems. You can also report a technical problem by fax if you prefer. Please mark faxes "**Attention: Technical Support**" and add your fax number to the information requested above.

Location	Contact
North America	Rational Software, 18880 Homestead Road, Cupertino, CA 95014 voice: (800) 433-5444 fax: (408) 863-4001 e-mail: support@rational.com

Europe, Middle East, and Africa

Rational Software,
Beechavenue 30,
1119 PV Schiphol-Rijk,
The Netherlands

voice: +31 20 454 6200

fax: +31 20 454 6201

e-mail: support@europe.rational.com

Asia Pacific

Rational Software Corporation Pty Ltd,
Level 13, Tower A, Zenith Centre,
821 Pacific Highway,
Chatswood NSW 2067,
Australia

voice: +61 2-9419-0111

fax: +61 2-9419-0123

e-mail: support@apac.rational.com

Command Line Reference

2

This section provides reference information to help you run the product's features from a command line. This can be useful in complex development environments to perform software testing tasks in the command line interface under UNIX or Windows operating systems.

Graphical User Interface

Purpose

The Graphical User Interface (GUI) of the product is an integrated environment that provides access to all of the capabilities packaged with the product.

Syntax

```
studio [-r <node>] [<filename>{,<filename>}]
```

where:

- *<filename>* can be an **.rtp** project or **.rtw** workspace file, as well as any text or report file that can be opened by the GUI.
- *<node>* is a project node to be executed.

Description

The studio command launches the GUI.

The **-r** option launches the GUI and automatically executes the specified node. Use the following syntax to indicate the path in the Project Explorer to the specified node:

```
<workspace_node>{[.<child_node>]}
```

Nodes in the path are separated by period ('.') symbols. If no node is specified, the GUI executes the entire project.

When using the **-r** option, an **.rtp** project file must be specified.

Example

The following command opens the **project.rtp** project file in the GUI, and runs the **app_2** node, located in **app_group_1** of **user_workspace**:

```
studio -r user_workspace.app_group_1.app_2 project.rtp  
Trace Receiver
```

Trace Receiver

Purpose

The Trace Receiver is a graphical client that receives and splits trace dump data through a socket.

Syntax

```
trtpd [<options>] [<file> [,<file>]]
```

where:

- *<file>* is one or several dynamic trace output files
- *<options>* is a set of optional parameters

Description

If a set of user-defined I/O functions uses sockets in a customized Target Deployment Port (TDP), the Trace Receiver can be used to receive the dump data and to split the trace files on-the-fly. Use the Target Deployment Port Editor to customize the TDP.

The Trace Receiver uses its own graphical user interface to display reception and split progression.

To use the Trace Receiver, you must:

- Customize the TDP to produce trace buffer output through a socket by setting the `SOCKET_UPLOAD` setting of the TDP to *True*
- Specify a delimiter character in the `SOCKET_UPLOAD_DELIMITER` setting of the TDP

The Runtime Trace Receptor uses the delimiter to find useful trace data and

directs the output to the trace file formats. If no filenames are provided, the following files are produced:

- **testing.rio** for Component Testing output to be processed by a Report Generator
- **purifylt.tpf** for Memory Profiling data
- **quantifylt.tqf** for Performance Profiling data
- **attolcov.tio** for Code Coverage data
- **tracer.tdf** for Runtime Tracing data

Options

```
-p | --port <number>
```

Port number. Specifies the decimal number of the port. The default port number is 7777.

```
-d | --delimiter <delimiter-byte>
```

Delimiter byte. Specified the decimal number of the delimiter byte. The default number is 94 ("^" in ASCII).

```
-o | --oneshot
```

Oneshot. Exits the Trace Receiver when the first client closes.

Example

The following trace dump is sent to the socket, using the "^" character as a delimiter:

```
^..  
TU "ms"  
SF 1 1dch 9527b66bh  
TI 1 1 5  
TM 726h  
HS 95fch
```

```
ME 3 1
NI 6 1
SF 2 10edh 72cbacbch
TM b68h
HS bea4h
^ ...
```

Use the following command line to receive and split the trace dump into the correct output file formats.

```
trtpd --port 7778 --delimiter 95 -o c:\\temp\\coverage.tio
c:\\temp\\trace.tdf c:\\temp\\profiling.tqf
```

You can also launch the Trace Receiver with no parameters. In this case, default parameters are assumed:

```
trtpd
```

Test Process Monitor

Purpose

Use the Test Process Monitor tool (**tpm_add**) to create and update Test Process Monitor databases from a command line.

Syntax

```
tpm_add -metric=<metric> [-file=<filename>] [-user=<user>]
{ [<value_field>] }
```

where:

- *<metric>* is the name of the metric.
- *<filename>* contains the name of the file under test to which the metric applies. This allows metrics for several files to be saved within the same database.
- *<user>* is the name of the product user who performed the measured value.
- *<value_field>* are the values attributed to each field

Description

The Test Process Monitor (TPM) provides an integrated monitoring feature that helps project managers and test engineers obtain a statistical analysis of the progress of their development effort.

Metrics generated by a test or runtime analysis feature are stored in their own database. Each database is actually a three-dimensional table containing:

- **Fields:** Each database contains a fixed number of fields. For example a typical Code Coverage database records.

- **Values:** Each field contains a series of values.
- **Filenames:** Values can be attributed to a filename, such as the name of the file under analysis. This way, the TPM Viewer can display result graphs for any single filename as well as for all files, allowing detailed statistical analysis.

Each field contains a set of values.

Note Although you specify a filename for the file under analysis, the TPM Viewer currently only displays a unique **FileID** number for each file.

The TPM database is made of two files that use the following naming convention:

```
<metric>.<user>.<nb_fields>.idx  
<metric>.<user>.<nb_fields>.tpm
```

where *<nb_fields>* is the number of fields contained in the database.

In the GUI, the Test Process Monitor gathers the statistical data from these database file and generates a graphical chart based on each field.

There are 3 steps to using TPM:

- Creating a database for the metric
- Updating the database
- Viewing the results in the GUI

Creating a Database

Before opening the Test Process Monitor in the product, you must create a database.

Database files are created by using the **tpm_add** command line tool.

If you are using Code Coverage from the GUI, it automatically creates and updates a TPM code-coverage database.

If you are using the product in the command line interface you can invoke **tpm_add** from your own scripts.

To create a new metric database with tpm_add:

- Type the following command:

```
tpm_add -metric=<name> -file=<filename> <value1>[  
{<value2>...}]
```

where *<name>* is the name of the new metric and *<value>* represents the initial value of each field in the database. *<filename>* is the name of the source file to which these values are related.

Updating a Database

The Test Process Monitor adds a record to the database each time it encounters an existing database.

To add a new record to this database:

- Type the **tpm_add** command:

```
tpm_add -metric=<name> <value1> [ {<value2>... } ]
```

where *<name>* is the name of the new metric and *<value>* represents the initial value of each field in the database. The number of values must be the consistent with the number of fields in the table.

Note It is important to remain consistent and supply the correct number of fields for your database. If you run the **tpm_add** command on an existing metric, but with a different number of fields, the feature creates a new database.

```
tpm_add -metric=stats 5 -6 5.4 3 0
```

Viewing TPM Reports

Use the Test Process Monitor menu in the product to display database. Please refer to the User Guide for further information.

Examples

The following command creates a user metric called *stats*, made up of five fields, containing initial values **1**, **0.03**, **0**, **3** and **-4.7**.

```
tpm_add -metric=stats -file=/project/src/myapp.c 1 0.03 0
3 -4.7
```

The new database is contained in the following files:

```
stats.user.5.idx
stats.user.5.tpm
```

The following line adds a new record to the *stats* database, pertaining to the **myapp.c** source file:

```
tpm_add -metric=stats -file=/project/src/myapp.c 5 -6 5.4
3 0
```

The following line adds a new set of values to the *stats* database, this time related to the **mylib.c** source file:

```
tpm_add -metric=stats -file=/project/src/mylib.c 5 -6 5.4
3 0
```

The metrics related to **myapp.c** and **mylib.c** are stored in the same database and can be viewed either jointly or separately in the product Test Process Monitor Viewer.

If the following command is issued:

```
tpm_add -metric=stats -file=myapp.c 5 -6 3 0
```

A new database is created with four fields:

```
stats.user.4.idx
stats.user.4.tpm
```

Dump File Splitter

Purpose

The dump file splitter (**atlsplit**) tool separates the unique multiplexed trace data file generated by the runtime analysis command line tools into specific trace files that can be processed by the runtime analysis and test feature Report Generators.

Syntax

```
atlsplit <trace_file>
```

where:

- *<trace_file>* is the name of the generated trace file (**atlout.spt**)

Description

The dump file splitter actually launches a *perl* script. You must therefore have a working perl interpreter such as the one provided with the product in the **/bin** directory.

Alternatively, you could use the following command line:

```
perl -I<installdir>/lib/perl  
<installdir>/lib/scripts/BatchSplit.pl atlout.spt
```

where *<install_dir>* is the installation directory of the product.

The script automatically detects which test or runtime analysis feature were used to generate the file and produces as many output files.

After the split, depending on the selected runtime analysis feature, the following file types are generated:

- **.rio test result files:** process with C Test Report Generator, Ada Test

Report Generator or System Testing Report Generator

- **.tio Code Coverage report files:** view with Code Coverage Viewer
- **.tdf Dynamic trace files:** view with UML/SD Viewer
- **.tpf Memory Profiling report files:** view with Memory Profiling Viewer
- **.tqf Performance Profiling report files:** view with Performance Profiling Viewer

C and C++ Instrumentor

Purpose

The two SCI Instrumentors for C and C++ insert functions from a Target Deployment Port library into the C or C++ source code under test. The C and C++ Instrumentors are used for:

- Memory Profiling
- Performance Profiling
- Code Coverage
- Runtime Tracing

Syntax

```
attolcc1 <src> <instr> <def> [<options>]
attolccp <src> <instr> <hpp> <opp> [<options>]
```

where:

- *<src>* Preprocessed source file (input)
- *<instr>* Instrumented file (output)
- *<def>* Standard definitions file the C Instrumentor only
- *<hpp>* and *<opp>* are the definition files for the C++ Instrumentor only

The *<src>* input file must have been preprocessed beforehand (with macro definitions expanded, include files included, **#if** and directives processed).

When using the C Instrumentor, all arguments are functions. When using the C++ Instrumentor, arguments are qualified functions, methods, classes, and namespaces, for example: **void C::B::f(int)**.

Description

The SCI Instrumentor builds an output source file from an input source file, by adding special calls to the Target Deployment Port function definitions.

The C Instrumentor (**attolcc1**) supports preprocessed ANSI or K&R C standard source code without distinction.

The C++ Instrumentor (**attolccp**) accepts preprocessed C++ files compliant with the ISO/IEC 14882:1998 standard. Depending on the Target Deployment Port, **attolccp** can also accept the C ISO/IEC 9899:1990 standard and other C++ dialects.

In C++, the following minor restrictions apply:

- **reinterpret_cast** does not allow casting a pointer to a member of one class to a pointer to a member of another class if the classes are unrelated.
- Template **attolccp** parameters are not accepted.

Both C and C++ versions of the Instrumentor accept either C or C++-style comments.

Attol pragmas start with the # character in the first column and end at the next line break.

The `<def>` and `<header>` parameters must not contain absolute or relative paths. The Code Coverage Instrumentor looks for these files in the directory specified by the **ATLTGT** environment variable, which must be set.

You can select one or more types of coverage at the instrumentation stage.

When you generate reports, results from some or all of the subset of selected coverage types are available.

General Options

```
-FILE=<file>[ { , <file> } ] | -EXFILE=<file>[ { , <file> } ]
```

-FILE specifies the only files that are to be explicitly instrumented, where *<file>* is a C/C++ source file. All other source files are ignored. Use this option with multiple */C++files* that can be found in a preprocessed file (*#includes* of files containing the bodies of C/C++ functions, lex and yacc outputs, and so forth).

-EXFILE explicitly specifies the files that are to be excluded from the instrumentation, where *<file>* is a C source file. All other source files are instrumented. You cannot use this option with the option **-FILE**.

-FILE and **-EXFILE** cannot be used together.

```
-UNIT=<name>[ { , <name> } ] | -EXUNIT=<name>[ { , <name> } ]
```

-UNIT specifies code units (functions, procedures, classes or methods) whose bodies are to be instrumented, where *<name>* is a unit which is to be explicitly instrumented. All other functions are ignored.

-EXUNIT specifies the units that are to be excluded from the instrumentation. All other units are instrumented.

-UNIT and **-EXUNIT** cannot be used together.

Note These options replace the **-SERVICE** and **-EXSERVICE** options from previous releases of the product.

```
-RENAME=<function>[ , <function> ]
```

For the C Instrumentor only. The **-RENAME** option allows you to change the name of C functions *<function>* defined in the file to be instrumented. Doing so, the *f* function will be changed to **_atw_stub_f**. Only definitions are changed, not declarations (prototypes) or calls. Component Testing for C can then define stubs to some functions inside the source file under test.

```
-NOINSTRDIR=<directory> [, <directory>]
```

Specifies that any C/C++ function found in a file in any of the *<directories>* or a sub-directory are not instrumented.

Note You can also use the **attol incl_std** pragma with the same effect in the standard definitions file.

```
-INSTANTIATIONMODE=ALL
```

C++ only. When set to **ALL**, this option enables instantiation of unused methods in template classes. By default, these methods are not instantiated by the C++ Instrumentor.

```
-DUMPCALLING=<name> [{ , <name> } ]
```

```
-DUMPINCOMING=<name> [{ , <name> } ]
```

```
-DUMPRETURNING=<name> [{ , <name> } ]
```

These options allow you to explicitly define where trace dump must be done. The **-DUMPCALLING** function is for the C Instrumentor only. See Instrumentation Snapshot Settings in the User Guide.

```
-NOPATH
```

Disables generation of the path to the Target Deployment Package directory in the `#include` directive. This lets you instrument and compile on different computers.

```
-NOINFO
```


Prohibits the Instrumentor from generating the identification header. This header is normally written at the beginning of the instrumented file, to strictly identify the instrument used.

-NODLINE

Prohibits the Instrumentor from generating *#line* statements which are not supported by all compilers. Use this option if you are using such a compiler.

-TSFDIR [=<directory>]

Not applicable to Code Coverage (see **FDCDIR**). Specifies the destination *<directory>* for the **.tsf** static trace file which is generated following instrumentation for each source code file. If *<directory>* is not specified, each **.fdc** file is generated in the corresponding source file's directory. If you do not use this option, the **.tsf** files directory is the working directory (the **attolcl** execution directory). You cannot use this option with the **-FDCNAME** option.

-TSFNAME=<name>

Not applicable to Code Coverage (see **FDCNAME**). Specifies the **.tsf** file name *<name>* to receive the static traces produced by the instrumentation. You cannot use this option with the **-TSFDIR** option.

-NOINCLUDE

This option excludes all included files from the instrumentation process. Use this option if there are too many excluded files to use the **-EXFILE** option.

Code Coverage Options

The following parameters are specific to the Code Coverage runtime analysis feature.

-PROC [=RET]

-**PROC** instruments procedure inputs (C/C++ functions). This is the default setting.

The **-PROC=RET** option instruments procedure inputs, outputs, and terminal instructions.

-CALL

Instruments C/C++ function calls.

-BLOCK=IMPLICIT | DECISION | LOGICAL

The **-BLOCK** option alone instruments simple blocks only.

Use the **IMPLICIT** or **DECISION** (these are equivalent) option to instrument implicit blocks (unwritten else instructions), as well as simple blocks.

Use the **LOGICAL** parameter to instrument logical blocks (loops), as well as the simple and implicit blocks.

By default, the Instrumentor instruments implicit blocks.

-NOTERNARY

This option allows you to abstract the measure from simple blocks. If you select simple blocks coverage, those found in ternary expressions are not considered as branches.

-COND [=MODIFIED | =COMPOUND | =FORCEEVALUATION]

MODIFIED or **COMPOUND** are equivalent settings that allow measuring the modified and compound conditions.

FORCEEVALUATION instruments forced conditions.

When **-COND** is used with no parameter, the Instrumentor instruments basic conditions.

-NOPROC

Specifies no instrumentation of procedure inputs, outputs, or returns, and so forth.

-NOCALL

Specifies no instrumentation of calls.

-NOBLOCK

Specifies no instrumentation of simple, implicit, or logical blocks.

-NOCOND

Specifies no instrumentation of basic conditions.

-COUNT

Specifies count mode.

-COMPACT

Specifies compact mode.

-EXCALL=<file>

For C only. Excludes calls to the C functions whose names are listed in <file> from being instrumented. The names of functions (identifiers) must be separated by space characters, tab characters, or line breaks. No other types of separator can be used.

-FDCDIR=<directory>

Specifies the destination <directory> for the **.fdc** correspondence file, which is generated for Code Coverage after the instrumentation for each source

file. Correspondence files contain static information about each enumerated branch and are used as inputs to the Code Coverage Report Generator. If *<directory>* is not specified, each **.fdc** file is generated in the directory of the corresponding source file. If you do not use this option, the default **.fdc** files directory is the working directory (the **attolccl** execution directory). You cannot use this option with the **-FDCNAME** option.

-FDCNAME=<name>

Specifies the **.fdc** correspondence file name *<name>* to receive correspondence produced by the instrumentation. You cannot use this option with the **-FDCDIR** option.

-NOCVI

Disables generation of a Code Coverage report that can be displayed in the Code Coverage Viewer.

-METRICS

Provides static metric data for compatibility with old versions of the product. Use the static metrics features of the Test Script Compiler tools instead. By default no static metrics are produced by the Instrumentors.

-NOSOURCE

Replaces the generation of the colored viewer source listing by a colored viewer pre-annotated report containing line number references.

-COMMENT=<comment>

Associates the text from either the Instrumentation Launcher (preprocessing command line) or from the source file under analysis and stores it in the **.fdc** correspondence file to be mentioned in Code Coverage reports. In the Code Coverage Viewer, a magnifying glass appears next to the source file, allowing you to display the comments in a separate window.

Memory Profiling Specific Options

The following parameters are specific to the Memory Profiling runtime analysis feature.

```
-MEMPRO
```

Activates instrumentation for the Runtime Tracing analysis feature.

```
-NOINSPECT=<variable> [, <variable>]
```

Specifies global variables that are not to be inspected for memory leaks. This option can be useful to save time and instrumentation overhead on trusted code.

Performance Profiling Specific Options

The following parameters are specific to the Performance Profiling runtime analysis feature.

```
-PERFPRO [=<os> | <process>]
```

Activates instrumentation for the Runtime Tracing analysis feature.

The optional *<os>* parameter allows you to specify a clock type. By default the standard operating system clock is used.

The *<process>* parameter specifies the total CPU time used by the process.

The *<os>* and *<process>* options depend on target availability.

Runtime Tracing Specific Options

The following parameters are specific to the Runtime Tracing analysis feature.

`-TRACE`

Activates instrumentation for the Runtime Tracing analysis feature.

`-NO_UNNAMED_TRACE`

For the C++ Instrumentor only. With this option, unnamed *structs* and *unions* are not instrumented.

`-NO_TEMPLATE_NOTE`

For the C++ Instrumentor only. With this option, the UML/SD Viewer will not display notes for template instances for each template class instance.

`-BEFORE_RETURN_EXPR`

For the C Instrumentor only. With this option, the UML/SD Viewer displays calls located in return expressions as if they were executed sequentially and not in a nested manner.

Component Testing Options for C++

The following parameters are specific to Component Testing for C++.

`-OTIFILE=<file>[{ , <file>}]`

Name of one or several Component Testing **.oti** instrumentation files for C++. These files contain rules for Component Testing instrumentation for C++ (they are generated by the C++ Test Script Compiler).

`-AUTO_OTI`

If this option is specified, Component Testing **.oti** instrumentation files for C++ will be searched and loaded in the directory specified with option **-OTIDIR**, or in current directory if this option is not used. **.oti** files are searched according to the source file names. For instance, if class **A** is found in file **myfile.h**, the **.oti** searched will be **myfile.oti**. An information

message is issued for each **.oti** file loaded automatically.

```
-OTIDIR=[<directory>]
```

This option specifies, when option **-AUTO_OTI** is active, which directory is to be searched. If no directory is specified (i.e. **-OTIDIR=** is specified), **.oti** files will be searched in the same directory as the source file they are matching.

```
-BODY=MAP_FILE | NAME_CONV | INLINE
```

This option specifies where generated methods body should be generated.

Use **INLINE** to generate method bodies in each instrumented source file as inline routines. This is the default, since there is little chance that the generated code cannot be accepted by a compiler, except with template classes on some compilers.

Use **NAME_CONV** to generate routine bodies in the **.cpp**, **.cc** or **.C** file whose name matches the **.h** file that contains the class definition of the generated method.

Use **MAP_FILE** when you provide a map file with the option **-MAPFILE**. This generates method bodies according to the map file.

```
-MAPFILE=<file>
```

When you add the **-BODY=MAP_FILE** option, this option must be provided. The **-MAPFILE** option specifies a user-created map file, describing where the methods of each class are to be generated.

This file must have the following format:

```
<source file>  
  <class name>  
    <class name>  
    ...  
<source file>
```

```
<class name>  
...  
...
```

Note that the character before a class name **MUST** be a tabulation.

For example:

```
a.cpp  
  A  
b.cpp  
  B
```

This specifies that class **A** methods bodies have to be generated in file **a.cpp**, and **B** methods bodies have to be generated in file **b.cpp**.

```
-NO_OTC
```

```
-NO_OTD
```

These options specify that Component Testing instrumentation rules for C++ issued from, respectively, an **.otc** contract check test script, or an **.otd** test driver script should be ignored.

```
-SHOWINFO
```

This option activates a diagnosis for each signature analysis. Usually, analysis of ill-formed signatures is silent. This option allows you to find ignored signatures

Note A signature is a string describing a class, a method, or a function, and is used in **.otc** and **.otd** files.

```
-NOWARNING
```

This option deactivates the warning display for signature analysis. The Instrumentor's signature analyzer accepts any non-ambiguous signature, and more permissive than most compilers. Warnings indicates signatures

that which are accepted by the instrumentor, but would be rejected by compilers.

`- INSTR_CONST`

Usually a C++ *const* method cannot modify any field of the *this* object. That's why the *const* methods are not checked for state changes, and are only evaluated once for invariants. But in some cases, the *this* object may change even if the method is qualified with *const* (by assembler code or by calling another method with casting the *this* parameter to a *non-const* type).

There may also be pointers fields to objects which logically belong to the object, but the C++ compiler does not guarantee that these pointed sub-objects are not modified. Use this option if the source code contains such pointers.

`-MTSUPPORT`

Use this option if your application is multi-threaded and objects are shared by several threads. This will ensure the specificity of each object for state evaluation.

Note To use multi-thread support in the product, you must also compile the Target Deployment Port with multi-thread support.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

C and C++ Instrumentation Launcher

Purpose

The Instrumentation Launcher instruments and compiles C and C++ source files. The Instrumentation Launcher is used by Memory Profiling, Performance Profiling, Runtime Tracing and Code Coverage, as well as the Component Testing Contract Check feature for C++.

Syntax

```
attolcc [<options>] -- <compilation_command>
```

where:

- *<compilation_command>* is the standard compiler command line that you would use to launch the compiler if you are not using the product
- "--" is the command separator preceded and followed by spaces
- *<options>* is a series of optional parameters

Description

The Instrumentation Launcher fits into your compilation sequence with minimal changes.

The Instrumentation Launcher is suitable for use with only one compiler and only one Target Deployment Port. To view information about the driver, run **attolcc** with no parameters.

The **attolcc** binary is located in the **/cmd** directory of the Target Deployment Port.

Note Some Target Deployment Ports do not have an **attolcc** binary. In this case, you must manually run the instrumentor, compiler and linker.

General Options

The Instrumentation Launcher accepts all command line parameters for either the C or C++ Instrumentor, including runtime analysis feature options. This allows the Instrumentation Launcher to automatically compile the selected Target Deployment Port.

In addition to Instrumentor parameters and Code Coverage parameters, the following options are specific to the Instrumentation Launcher. Command line options can be abbreviated to their shortest unambiguous number of characters and are not case-sensitive.

```
-VERBOSE | -#
```

The **-VERBOSE** option shows commands and runs them. The **-#** option shows commands but does not execute them.

```
-TRACE  
-MEMPRO  
-PERFPRO
```

These options activate specific instrumentation for the Runtime Tracing, Memory Profiling and Performance Profiling runtime analysis feature.

```
-OTIFILE=<file>[{ , <file>}]
```

When using the Contract Check capability of Component Testing for C++, the **-OTIFILE** option allows you to specify one or several Component Testing **.oti** instrumentation files for C++. These files are generated by the C++ Test Compiler and contain the Component Testing instrumentation rules for C++.

```
-AUTO_OTI
```

When using the Contract Check capability of Component Testing for C++,

this option specifies that Component Testing instrumentation files (.oti) for C++ are to be searched and loaded from the directory specified with option - **OTIDIR**, or in current directory if this option is not used. **.oti** files are searched according to the source file names. For instance, if class A is found in file **myfile.h**, the **.oti** searched will be **myfile.oti**. An information message is issued for each **.oti** file loaded automatically.\$

-FORCE_TDP_CC

This option forces the Instrumentation Launcher to attempt to compile the Target Deployment Port even if the link phase has not yet been reached before the **TP.o** or **TP.obj** is built.

-NOSTOP

This option forces the initial command to resume when a failure occurs during preprocessing, instrumentation, compilation or link. This means that the build chain is not interrupted by errors, but the resulting binary may not be fully instrumented. Use this option when debugging instrumentation issues on large projects.

Each error is logged in an **attolcc.log** file located in the directory where the error occurred.

Code Coverage Options

The following parameters are specific to the Code Coverage runtime analysis feature.

-PASS | -COUNT | -COMPACT

Pass mode only indicates whether a branch has been hit. The default setting is pass mode.

Count mode keeps track of the number of times each branch is exercised. The results shown in the code coverage report include the number of hits as well as the pass mode information.

Compact mode. Compact mode is equivalent to pass mode, but each branch is stored in one bit, instead of one byte as in pass mode. This reduces the overhead on data size.

`-COMMENT` | `-NOCOMMENT`

The comment option lets the user associate a comment string with the source in the code coverage reports and in Code Coverage Viewer.

By default, the Instrumentation Launcher sends the preprocessing command as a comment. This allows you to distinguish the source file that was preprocessed and compiled more than once with distinct options.

Use `-NOCOMMENT` to disable the comment setting.

Example

```
atolcc -- cc -I../include -o appli appli.c bibli.c -lm
atolcc -TRACE -- cc -I../include -o appli appli.c bibli.c
-lm
```

Return codes

The return code from the Instrumentation Launcher is either the first non-zero code received from one of the commands it has executed, or 0 if all commands ran successfully. Due to this, the Instrumentation Launcher is fully compatible with the *make* mechanism.

If an error occurs while the Instrumentation Launcher - or one of the commands it handles - is running, the following message is generated:

ERROR : Error during C preprocessing

All messages are sent to the standard error output device.

Java Instrumentor

Purpose

The SCI Instrumentor for Java inserts methods from a Target Deployment Port library into the Java source code under test. The Java Instrumentor is used for:

- Performance Profiling
- Code Coverage
- Runtime Tracing

Memory Profiling for Java uses the JVMPI Agent instead of source code insertion (SCI) technology as for other languages.

Syntax

```
javi <src> { [,<src> ] } [<options>]
```

where:

- <src> is one or several Java source files (input)

Description

The SCI Instrumentor builds an output source file from each input source file by adding specific calls to the Target Deployment Port method definitions. These calls are used by the product's runtime analysis features when the Java application is built and executed.

The Java Instrumentor creates the output files in a **javi.jir** directory, which is located inside the current directory. By default, this directory is cleaned and rewritten each time the Instrumentor is executed.

Although the Java Instrumentor can take several input source files on the command line, you only need to provide the file containing a *main* method for the Instrumentor to locate and instrument all dependencies.

When using the Code Coverage feature, you can select one or more types of coverage at the instrumentation stage (see the User Guide for more information). When you generate reports, results from some or all of the subset of selected coverage types are available.

Options

```
-FILE=<file>[,{,<file>}] | -EXFILE=<file>[,{,<file>}]
```

-FILE specifies the only files that are to be explicitly instrumented, where *<file>* is a Java source file. All other source files are ignored.

-EXFILE explicitly specifies the files that are to be excluded from the instrumentation, where *<file>* is a Java source file. All other source files are instrumented.

-FILE and **-EXFILE** cannot be used together.

```
-CLASSPATH=<classpath>
```

The **-CLASSPATH** option overrides the **\$CLASSPATH** and **\$EDG_CLASSPATH** environment variables -in that order- during instrumentation.

In *<classpath>*, each path is separated by a colon (":") on UNIX systems and a semicolon (";") in Windows.

```
-OPP=<file>
```

The **-OPP** option allows you to specify an optional definition file. The *<file>* parameter is a relative or absolute filename.

-DESTDIR=<directory>

The **-DESTDIR** option specifies the location where the **javi.jir** output directory containing the instrumented Java source files is to be created. By default, the output directory is created in the current directory.

-PROC [=RET]

The **-PROC** option alone causes instrumentation of all classes and method entries. This is the default setting.

The **-PROC=RET** option instruments procedure inputs, outputs, and terminal instructions.

-BLOCK=IMPLICIT | DECISION | LOGICAL

The **-BLOCK** option alone instruments simple blocks only.

Use the **IMPLICIT** or **DECISION** (these are equivalent) option to instrument implicit blocks (unwritten else instructions), as well as simple blocks.

Use the **LOGICAL** parameter to instrument logical blocks (loops), as well as the simple and implicit blocks.

By default, the Instrumentor instruments implicit blocks.

-NOTERNARY

This option allows you to abstract the measure from simple blocks. If you select simple block coverage, those found in ternary expressions are not considered as branches.

-NOPROC

Specifies no instrumentation of procedure inputs, outputs, or returns, and

so forth.

```
-NOBLOCK
```

Specifies no instrumentation of simple, implicit, or logical blocks.

```
-COUNT
```

Specifies count mode. By default, the Instrumentor uses pass mode. See the User Guide.

```
-COMPACT
```

Specifies compact mode. By default, the Instrumentor uses pass mode. See the User Guide.

```
-UNIT=<name>[{ , <name>}] | -EXUNIT=<name>[{ , <name>}]
```

-UNIT specifies Java units whose bodies are to be instrumented, where *<name>* is an Java package, class or method which is to be explicitly instrumented. All other units are ignored.

-EXUNIT specifies the units that are to be excluded from the instrumentation. All other Java units are instrumented.

-UNIT and **-EXUNIT** cannot be used together.

```
-DUMPINCOMING=<service>[{ , <service>}]
```

```
-DUMPRETURNING=<service>[{ , <service>}]
```

```
-MAIN=<service>
```

These options allow you to precisely specify where the SCI dump must occur. **-MAIN** is equivalent to **-DUMPRETURNING**.

```
-COMMENT=<comment>
```

Associates the text from either the Code Coverage Launcher (preprocessing

command line) or from you with the source file and stores it in the FDC file to be mentioned in coverage reports. In Code Coverage Viewer, a magnifying glass is put in front of the source file. Clicking this magnifying glass shows this text in a separate window.

-NOCVI

Disables generation of a Code Coverage report that can be displayed in the Code Coverage Viewer.

-METRICS

Provides static metric data for compatibility with old versions of the product. Use the static metrics features of the Test Script Compiler tools instead. By default, no static metrics are produced by the Instrumentors.

-JTEST | -NOJTEST

The **-JTEST** option provides UML sequence diagram output for Component Testing for Java with Test RealTime. **-NOJTEST** disables this output.

-NOCLEAN

When this option is set, the Instrumentor does not clear the **javi.jir** directory before generating new files.

-FDCDIR=<directory>

Specifies the destination <directory> for the **.fdc** correspondence file, which is generated for Code Coverage after the instrumentation for each source file. Correspondence files contain static information about each enumerated branch and are used as inputs to the Code Coverage Report Generator. If <directory> is not specified, each **.fdc** file is generated in the directory of the corresponding source file. If you do not use this option, the default **.fdc** files directory is the current working directory. You cannot use this option with the **-FDCNAME** option.

-FDCNAME=<name>

Specifies the **.fdc** correspondence file name *<name>* to receive correspondence produced by the instrumentation. You cannot use this option with the **-FDCDIR** option.

-NO UNNAMED TRACE

With this option, anonymous classes are not instrumented.

-PERFPRO

This option activates Performance Profiling instrumentation. This produces output for a Performance Profile report.

-TRACE

This option activates Runtime Tracing instrumentation. This produces output for a UML sequence diagram.

-TSFNAME=<file>

Disables generation of a Code Coverage report that can be displayed in the Code Coverage Viewer.

-TSFDIR=<directory>

Specifies the destination *<directory>* for the **.tsf** static trace file, which is generated for Code Coverage after the instrumentation of each source file. If *<directory>* is not specified, each **.tsf** static trace file is generated in the directory of the corresponding source file. If you do not use this option, the default **.tsf** static trace file directory is the current working directory. You cannot use this option with the **-TSFNAME** option.

-TSFNAME=<file>

Specifies the *<name>* of the **.tsf** static trace file that is to be produced by the instrumentation. You cannot use this option with the **-TSFDIR** option.

`-INSTRUMENTATION= [FLOW | COUNT | INLINE]`

Choose specifies the instrumentation mode. By default, count mode is used, which is a compromise between the flow mode (everything is a call to the Target Deployment Package) and the inline mode (when possible, the code is directly inserted into the generated file).

Warning: Inline mode must be used only in pass mode. Do not use this option if you want to know how many times a branch is reached.

`-NOINFO`

Asks the Instrumentor not to generate the identification header. This header is normally written at the beginning of the instrumented file.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

Java Instrumentation Launcher

Purpose

The Instrumentation Launcher instruments and compiles Java source files. The Instrumentation Launcher is used by Performance Profiling, Runtime Tracing and Code Coverage.

Syntax

```
javic [<options>] -- <compilation_command>
```

where:

- *<compilation_command>* is the standard compiler command line that you would use to launch the compiler if you are not using the product
- "--" is the command separator preceded and followed by spaces
- *<options>* is a series of optional parameters for the Java Instrumentor.

Description

The Instrumentation Launcher (**javic**) fits into your compilation sequence with minimal changes.

The Instrumentation Launcher is suitable for use with only one compiler and only one Target Deployment Port. To view information about the driver, run **javic** with no parameters.

The **javic** (or **javic.exe**) binary is located in the **cmd** subdirectory of the Target Deployment Port.

The Java Instrumentation Launcher automatically sets the **SATLTGT** environment variable if it is not already set.

The Instrumentation Launcher accepts all command line options designed for the Java Instrumentor.

Command line options can be abbreviated to their shortest unambiguous number of characters and are not case-sensitive.

Customization

The **javac** (or **javac.exe**) binary is a copy of the **perllauncher** (or **perllauncher.exe**) binary located in `<InstallDir>/bin/<platform>/<os>`.

The launcher runs the **javac.pl** perl script which is located in the **cmd** subdirectory and produces the **products.java** file that contains the default configuration settings. These are copied from `<InstallDir>/lib/scripts/BatchJavaDefault.pl`.

The **javac.pl** included with the product is for the Sun JDK 1.3.1 or 1.4.0 compiler. This script can be changed in the TDP Editor, allowing you to customize the default settings, which are based on the **BatchJavaDefault.pl** script, before the call to **PrepareJavaTargetPackage**.

Options

The Launcher accepts the following settings:

```
--atl_threads_max=<number>
```

Sets the maximum number of threads at the same time. The default value is **64**.

```
--atl_buffer_size=<bytes>
```

Sets the size of the Dump Buffer in bytes. The default value is **16384**.

```
--address=<IPaddress>
```


Address of the Socket Trace Receiver Host. The default address is **127.0.0.1**.

```
--uploader_port=<port number>
```

Port number listened to by the Socket Trace Receiver Host. The default port number is **7777**.

```
--atl_run_gc_at_exit=0|1
```

Set this setting to 1 to run finalizers invoking the Garbage Collector upon exit. **0** disables the option. Default is **1**.

```
--atj_check_stub=0|1
```

Check Component Testing for Java stubs. Default is **0**.

```
--atj_display_stub=0|1
```

Display Component Testing for Java stubs in Runtime Tracing. Default is **0**.

```
--att_on_the_fly=0|1
```

If set to 1, implies that each tdf lines are flushed immediately in order to be read on-the-fly by Runtime Tracing. Default is **1**.

```
--att_partial_dump=0|1
```

Partial Message Dump is on if set to 1 in Runtime Tracing. Default is **0**.

```
--att_timestamp=0|1
```

If 1 record and display Time Stamp in Runtime Tracing. Default is **1**.

```
--att_heap_size=0|1
```

Record and Display Current Heap Size in Runtime Tracing. Default is **1**.

```
--att_thread_info=0|1
```

Record and Display Thread Information in Runtime Tracing. Default is **1**.

```
--att_record_max_stack=0|1
```

Record and Display Max Stack in a note in Runtime Tracing. Default is **1**.

Example

The following command launches Runtime Tracing instrumentation of **program1.java** and its dependencies, then compiles the instrumented classes in the **java.jir** directory.

```
javic -trace -- javac program1.java
```

The following command launches Code Coverage instrumentation of **program2.java** and **program3.java**, as well as their dependencies, and generates the instrumented classes in the **tmpclasses** directory.

```
javic -proc=r -block=1 -- javac program1.java  
program2.java -d tmpclasses
```

In this example, **tmpclasses** will contain the compiled TDP classes only if they are not already in the TDP directory. The **-d** option creates these TDP **.class** files in the same location as the source files. Make sure that you set a correct **CLASSPATH** when running the application.

Java Instrumentation Launcher for Ant

Purpose

The Java Instrumentation Launcher (**javic**) for Ant provides integration of the Java Instrumentor with the Apache Jakarta Ant build utility.

Description

This adapter allows automation of the instrumented build process for Ant users by providing an Ant CompilerAdapter implementation called **com.rational.testrealtime.Javic**.

The Java Instrumentation Launcher for Ant provided with the product supports version 1.4.1 of Ant, but is delivered as source code, so that you can adapt it to any release of Ant. Source code for the Javic class is available at:

```
<InstallDir>/lib/java/ant/com/rational/testrealtime/Javic.java
```

Javic uses the **build.actual.compiler** property to obtain the name of your Java compiler. When using JDK 1.4.0, this name is **modern**. Please refer to Ant documentation for other values.

In some cases **-opp=<file>** and **-destdir=<dir>** can not be set in the **Javi.options** property:

- The **.opp** instrumentation file is automatically set in the **-opp=<file>** option by the Javic class if and only if **\$ATLTGT/ana/atl.opp** exists.
- The instrumented file repository directory, where the **javi.jir** subdirectory is created, is automatically set by the Javic class if the **destdir** attribute is set in the **javac** task.

-classpath=<classpath> cannot be set in the **Javi.options** property.

The *classpath* used by the Java Instrumentor is a merge of the *classpath* attribute of the javac task with the **\$CLASSPATH** and **\$EDG_CLASSPATH** contents.

\$ATLTGT must point to the Java TDP directory, for example: *<InstallDir>/targets/jdk_1.4.0*. On Windows platforms, this path must be provided in short-name DOS format.

Options

The Launcher accepts the following settings:

```
--atl_threads_max=<number>
```

Sets the maximum number of threads at the same time. The default value is **64**.

```
--atl_buffer_size=<bytes>
```

Sets the size of the Dump Buffer in bytes. The default value is **16384**.

```
--address=<IPaddress>
```

Address of the Socket Trace Receiver Host. The default address is **127.0.0.1**.

```
--uploader_port=<port number>
```

Port number listened to by the Socket Trace Receiver Host. The default port number is **7777**.

```
--atl_run_gc_at_exit=0|1
```

Set this setting to 1 to run finalizers invoking the Garbage Collector upon exit. **0** disables the option. Default is **1**.

```
--atj_check_stub=0|1
```

Check Component Testing for Java stubs. Default is **0**.

```
--atj_display_stub=0|1
```

Display Component Testing for Java stubs in Runtime Tracing. Default is **0**.

```
--att_on_the_fly=0|1
```

If set to 1, implies that each tdf lines are flushed immediately in order to be read on-the-fly by Runtime Tracing. Default is **1**.

```
--att_partial_dump=0|1
```

Partial Message Dump is on if set to 1 in Runtime Tracing. Default is **0**.

```
--att_timestamp=0|1
```

If 1 record and display Time Stamp in Runtime Tracing. Default is **1**.

```
--att_heap_size=0|1
```

Record and Display Current Heap Size in Runtime Tracing. Default is **1**.

```
--att_thread_info=0|1
```

Record and Display Thread Information in Runtime Tracing. Default is **1**.

```
--att_record_max_stack=0|1
```

Record and Display Max Stack in a note in Runtime Tracing. Default is **1**.

To install the Javac class for Ant:

1. Download and install Ant v1.4.1 from <http://jakarta.apache.org/ant/>
2. Set **ANT_HOME** to the installation directory, for example:
/usr/local/jakarta-ant-1.4.1.
3. Add **\$ANT_HOME/bin** in your **PATH**

4. Compile and install the **Javic** class. In the ant directory, type:

```
ant
```

This adds the **javic.jar** to the **\$ANT_HOME/lib** directory.

Example

The files for the following example are located in *<InstallDir>/lib/java/ant/example*.

The following command performs a standard build based on the build.xml file

```
ant
```

This produces the following output:

```
Buildfile: build.xml
clean:
cc:
    [javac] Compiling 1 source file
all:
BUILD SUCCESSFUL
Total time: 2 seconds
```

To perform an instrumented build of the same build.xml, without modifying that file:

```
ant -DATLTGT=$ATLTGT -
Dbuild.compiler=com.rational.testrealtime.Javic -
Dbuild.actual.compiler=modern -Djavi.options=-trace -
Djavi.settings=--att_on_the_fly=0
```

This produces the following output:

```
Buildfile: build.xml
clean:
    [delete] Deleting: Sample.class
cc:
    [javac] Compiling 1 source file
    [javi]   Instrumenting 1 source file
    [javac] Compiling 1 source file
all:
BUILD SUCCESSFUL
Total time: 4 seconds
```

JVMPI Agent

Purpose

The JVMPI Agent is a dynamic library that is part of the J2SE and J2ME virtual machine distributions. The Agent ensure the memory profiling functionality when using the Memory Profiling feature for Java.

Syntax

```
java -Xint -Xrunagent[:<options>] <configuration>
```

where:

- *<options>* are the command line options of the JVMPI agent
- *<configuration>* is the configuration required to run the application

Description

Because of the garbage collector concept used in Java, Performance Profiling for Java uses the JVMPI agent facility delivered by the JVM. This differentiates Memory Profiling for Java from the SCI instrumentation technology used with other languages.

To run the JVMPI Agent from the command line, add the **-Xrunagent** option to the Java command line.

The JVMPI Agent analyzes the following internal events of the JVM:

- Method entries and exits
- Object and primitive type allocations

The JVMPI Agent retrieves source code debug information during runtime. When the Agent receives a snapshot trigger request, it can either execute an

instantaneous JVMPI dump of the JVM memory, or wait for the next garbage collection to be performed.

Note Information provided by the instantaneous dump includes actual memory use as well as intermediate and unreferenced objects that are normally freed by the garbage collection.

The actual trigger event can be implemented with any of the following methods:

- A specified method entry or exit used in the Java code
- A message sent from the **Snapshot** button or menu item in the graphical user interface
- Every garbage collection

The JVMPI Agent requires that the Java code is compiled in *debug* mode, and cannot be used with Java in just-in-time (JIT) mode.

Options

The following parameters can be sent to the JVMPI Agent on the command line.

```
-H_Cx=<size>  
-H_Ox=<size>
```

Specifies the size of hashtables for classes (**-H_Cx**) or objects (**-H_Ox**) where *<size>* must be 1, 3, 5 or 7, corresponding respectively to hashtables of 64, 256, 1024 or 4096 values.

```
-JVM <prefix>
```

By default, the Agent waits for the virtual machine (VM) to be fully initialized before it starts collecting data. This usually relates to the

spawning of the first user thread. With the **-JVM** option, data collection starts on the first memory allocation, even if the VM is not fully initialized.

-N_O

With the **-N_O** option, the Agent only counts the number of allocated objects and ignores any further object data. The existence of the objects after garbage collection cannot be verified. Use this option to reduce Performance Profiling overhead or to obtain a quick summary.

-D_O_N

Delete Object No. By default, the Agent only collects and presents method data on the latest call to that method. Any further calls to the method replaces existing call data.

Use the **-D_O_N** option to display all referenced objects.

-D_GC

This option requests a JVMPI dump after each garbage collection

-D_PGC

When using a dump request method, this option makes the Agent wait until the next garbage collection before performing the dump.

-D_M [[<method>, <class>, <mode>] , [, <method>, <class>, <mode>]]

Activates "Dump Method" mode.

Use this option to perform a snapshot on entry or exit of the specified methods, where *<mode>* may be **0** or **1**:

- **0** performs the method dump upon exit
- **1** performs the method dump on entry

<class> must be the fully qualified name of a class, including the entire package name.

```
-O_M [ [<method>, <class>], [<method>, <class>] ]
```

Activates "Observe Method" mode.

Use this option to store the call stack when the specified methods are called. The stack is loaded from 0 to 10 (max).

```
-U_S= [<name>]
```

User name

This option adds the name of the user to the JVMPI dump data. The name must be specified between brackets ("[]").

```
-D_U= [<string>]
```

This option specifies a start date that is used by the JVMPI dump data. The string must be specified between brackets ("[]").

```
-F_M [ [<method>, <class>], [<method>, <class>] ]
```

Filter mode.

Use this option to produce JVMPI data only on the specified method(s). All other methods are ignored.

```
-H_N= [<hostname>]
```

Hostname.

Use this option to specify a hostname for the JVMPI Agent to communicate with the graphical user interface on the local host. The hostname must be specified between brackets ("[]").

```
-P_T= [<port_number>]
```

Port number. Use this option to specify a port number for the JVMPI Agent to communicate with the graphical user interface on the local host. The port number must be specified between brackets ("[]").

```
-OUT= [<filename>]
```

Output filename.

This option specified the name of the trace dump file produced by the JVMPI Agent. Use the Dump File Splitter on this output file to produce a **.tsf** static trace file for the GUI Memory Profiling Viewer.

Examples

The following examples launches the JVMPI Agent by dumping the ***exportvalues*** and ***exportvalues2*** methods of the ***com.rational.Th*** class:

```
java -Xint -xrunpagent:-JVM-  
D_M[[exportvalues,com.rational.Th,0],[exportvalues2,com.rational.T  
h,0]] -classpath $CLASSPATH Th
```

Ada Instrumentor

Purpose

The source code insertion (SCI) Instrumentor for Ada inserts functions from a Target Deployment Port library into the Ada source code under test. The Ada Instrumentor is used for Code Coverage only.

Syntax

```
attolada <src> <instr> [<options>]
```

where:

- <src> is the source file (input)
- <instr> is the instrumented output file

Description

The Instrumentor builds an output source file from an input source file, by adding special calls to the Target Deployment Port function definitions.

The Ada Instrumentor (**attolada**) supports Ada83 and Ada95 standard source code without distinction.

You can select one or more types of coverage at the instrumentation stage (see the User Guide for more information).

When you generate reports, results from some or all of the subset of selected coverage types are available.

Options

```
-PROC [=RET]
```

-PROC alone instruments procedure, function, package, and task entries. This is the default setting.

The **-PROC=RET** option instruments both entries and exits.

-CALL

Instruments Ada functions or procedures.

-BLOCK [=IMPLICIT | DECISION | LOGICAL | ATC]

This option specifies how blocks are to be instrumented.

- The **-BLOCK** option alone instruments simple blocks only.
- Use the **IMPLICIT** or **DECISION** option to instrument implicit blocks (unwritten else instructions), as well as simple blocks.
- Use the **LOGICAL** parameter to instrument logical blocks (loops), as well as the simple and implicit blocks.
- Use the **ATC** parameter to extend the instrumentation to asynchronous transfer control (**ATC**) blocks.

By default, the Instrumentor instruments implicit blocks.

-COND [=MODIFIED | COMPOUND | FORCEEVALUATION]

When **-COND** is used with no parameter, the Instrumentor instruments basic conditions.

- **MODIFIED** or **COMPOUND** are equivalent settings that allow measuring the modified and compound conditions.
- **FORCEEVALUATION** instruments forced conditions.

-NOPROC

Disables instrumentation of procedure inputs, outputs, or returns, etc.

-NOCALL

Disables instrumentation of calls.

-NOBLOCK

Disables instrumentation of simple, implicit, or logical blocks.

-NOCOND

Disables instrumentation of basic conditions.

-UNIT=<name>[{ , <name>}] | **-EXUNIT=<name>[{ , <name>}]**

-UNIT specifies Ada units whose bodies are to be instrumented, where *<name>* is an Ada unit which is to be explicitly instrumented. All other functions are ignored.

-EXUNIT specifies the units that are to be excluded from the instrumentation. All other Ada units are instrumented.

-UNIT and **-EXUNIT** cannot be used together.

-LINK=<filename>[{ , <filename>}]

Provides a set of link files to the Instrumentor.

-STDLINK=<filename>

Provides a standard link file to the Instrumentor.

-FDCDIR=<directory>

Specifies the destination *<directory>* for the **.fdc** correspondence file, which is generated for Code Coverage after the instrumentation for each source file. Correspondence files contain static information about each enumerated branch and are used as inputs to the Code Coverage Report Generator. If *<directory>* is not specified, each **.fdc** file is generated in the directory of the

corresponding source file. If you do not use this option, the default **.fdc** files directory is the working directory (the **attolccl** execution directory). You cannot use this option with the **-FDCNAME** option.

-FDCNAME=<name>

Specifies the **.fdc** correspondence file name *<name>* to receive correspondence produced by the instrumentation. You cannot use this option with the **-FDCDIR** option.

-DUMPINCOMING=<name>[{ , <name>}]

-DUMPRETURNING=<name>[{ , <name>}]

Allow you to tell your list of C function, on whose incoming, or returning, the traces dump must be done explicitly.

-COMMENT=<comment>

Associates the text from either the Code Coverage Launcher (preprocessing command line) or from you with the source file and stores it in the FDC file to be mentioned in coverage reports. In Code Coverage Viewer, a magnifying glass is put in front of the source file. Clicking on this magnifying glass, shows this text in a separate window.

-NOMETRICS

Saves the metrics basic data calculation time.

-RESTRICTION =NOEXCEPTION | NOGENERIC | CSMART

Use this option to set a restriction.

- **NOEXCEPTION** deactivates instrumentation of exception block branches encountered in the source file. When this option is active, no coverage information is available on exception blocks or on instructions contained in exception blocks.

- **NOGENERIC** deactivates the instrumentation using a generic Target Deployment Port call. When this option is active, the generated source code may contain uninstrumentable calls. If used with the **-CALL** option, this can generate compilation errors depending on your application if, for example, you use private packages as well as private sub-packages.
- **CSMART** generates **CSMART** compliant code.

-NOSOURCE

Replaces the generation of the colored viewer source listing by a colored viewer pre-annotated report containing line number references.

-NOCVI

Disables generation of a Code Coverage report that can be displayed in the Code Coverage Viewer.

-METRICS

Provides static metric data for compatibility with old versions of the product. Use the static metrics features of the Test Script Compiler tools instead. By default no static metrics are produced by the Instrumentors.

-GENERATEDNAME = CHECKSUM | <filename>

-USERNAME = <NAME>

Use these options to add a package to the header of the generated file to store coverage traces. You can specify the name of the generated package using one of the following three options:

- **-GENERATEDNAME=CHECKSUM** uses a checksum calculated on the instrumented file to create a package name under the form **ATC_<checksum>**, where **<checksum>** has a maximum of four letters.
- **-GENERATEDNAME=<filename>** uses the name of the file to be

instrumented, this name is transformed into an Ada identifier and prefixed by **ATC_**.

- **-USERNAME**=<username>: A name you choose freely by the user and provide on the command line.

<File> is used without checking whether it is a valid Ada identifier.

By default, the **-GENERATEDNAME**=<FILE> option is used.

-PREFIX=<prefix>

You can prefix some instrumentations (name of the generated package, variables, etc.) if there are any semantic ambiguities. Thus, packages generated by **attolada** can be recognized by giving them a known prefix.

By default, no prefix is used.

Note The prefix you provide is used, without checking whether it is a valid Ada identifier.

-SPECIFICATION

Extends instrumentation of calls and conditions to source code inside package specifications.

-MAIN=<unit>[{ , <unit>}]

Forces a trace dump at the end of the main unit of your application.

-EXCALL=<unit>[{ , <unit>}]

Excludes from call instrumentation the calls to specified units or to functions or procedures inside the specified units.

-ADA83 | **-ADA95**

Choose specifies the Ada language used by the Instrumentor. This language is applied to the analyzed and generated file.

```
-INSTRUMENTATION= [FLOW | COUNT | INLINE]
```

Choose specifies the instrumentation mode. By default, count mode is used, which is a compromise between the flow mode (everything is a call to the Target Deployment Package) and the inline mode (when possible, the code is directly inserted into the generated file).

Note Inline mode must be used only in pass mode. Do not use this option if you want to know how many times a branch is reached.

```
-NOINFO
```

Asks the Instrumentor not to generate the identification header. This header is normally written at the beginning of the instrumented file, to strictly identify the instrument used.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

Ada Link File Generator

Purpose

The Ada Link File Generator (**attolalk**) feature automatically generates link files. It uses file name extensions that you allow or disallow, and on the file list found in the specified directories.

Syntax

```
attolalk [<options>] <link file name> <directory> [<directory> ...  
<directory>]
```

where:

- *<link file name>* is the name of the generated link file. If **attolalk** cannot write this file a fatal error is generated.
- *<directory>* is a directory name. If **attolalk** cannot read file from this directory, a fatal error is generated.
- *<options>* is a set of optional command line parameters as specified in the following section.

Description

The Link File Generator requires that the **LD_LIBRARY_PATH** environment variable is set to the **/lib** directory in the product installation directory.

File Extensions

A file extension is a character string of unconstrained positive length (greater than zero). A file name matches an extension if its length is greater than the length of extension, and if the N last characters of the file name are identical to the characters of the extension (N is the length of the extension).

For example, **source.ada** matches the **.ada** extension but not **.1.ada** extension.

Permitted and Forbidden Extensions:

Permitted and forbidden file extensions for the Link File Generator are specified by the **ATTOLALK_EXT** and **ATTOLALK_NOEXT** environment variables and are separated by the ':' character on UNIX systems and ';' on Windows. For example:

```
ATTOLALK_EXT=".ada:.a:.am"  
ATTOLALK_NOEXT=".1.ada"
```

By default, the allowed extension list is **".ada:ads:adb"** and the forbidden extension list is empty.

Link File Generation

For each given directory, the contained file name list is loaded. Each file name is compared with the allowed extensions. If a match is found, the file name is compared with forbidden extension. If there is no match, the file is taken as an Ada source file. Each Ada source file is analyzed and may produce one or more lines in the generated link file (with the syntax described above).

Command Line Parameters

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

```
-r
```

Relative paths. With the **-r** option, all filenames are generated with relative paths.

```
-s
```

Silent mode. With the **-s** option, only errors are displayed.

-f

Force all Ada files. By default, the Link File Generator only analyzes Ada source files that were changed since the last analysis. Use the **-f** option to force the analysis of all Ada source files, regardless of when they were modified.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

Ada Unit Maker

Purpose

The Instrumentor generates several compilation units in the same file. Some compilers require a separate file for each compilation unit.

To achieve this, the Ada Unit Maker feature generates one file for each compilation unit found in a specified Ada source file as the *gnatchop* command, provided with the GNAT Ada compiler, does. You can choose the name of the generated files from several naming conventions.

Syntax

```
attolchop [<options>] <source file name>
```

where:

- *<source file name>* is the source file name to analyze. If this file cannot be read or contains lexical or syntax errors, a fatal error is generated.
- *<options>* is a set of optional command line parameters as specified in the following section.

Description

The Ada Unit Maker feature can generate file names for Rational Apex or Gnat naming standards. To choose the naming standard, either set the **ATTOLCHOP** environment variable to **GNAT** or **APEX** or use the **-n** command line parameter. By default, the Ada Unit Maker uses the Gnat naming convention.

Gnat Naming

The full compilation unit name is set to lower case and all dot characters (".") are replaced with hyphens ("-"). The feature appends the **.ads** extension

to the name if the unit is an extension or the **.adb** extension if the unit is a body. The Krunch Gnat short name mode is not supported by the Ada Unit Maker. Please refer to your Gnat documentation for further information.

Rational Apex Naming

The full compilation unit name is set to lower case; then the feature appends a **.1.ada** extension to the filename if the unit is a specification, or a **.2.ada** extension if the unit is a body. Please refer to your Apex documentation for further information.

Options

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

-l

This option must be placed first if it is used. This tells the Ada Unit Maker feature to send the name of the generated file, and no other messages, to the standard output.

-w

Overwrite. By default, the Ada Unit Maker produces an error if a filename already exists. Use the **-w** option to overwrite any existing files.

-n APEX | GNAT

Naming standard. Use the **-n** option to select either the Rational Apex or Gnat naming convention. This parameter overrides the default setting (Gnat) as well as the **ATTOLCHOP** environment variable if set.

Return Codes

After execution, the program exits with the following return codes:

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of a fatal error
9	End of execution because of an internal error

All messages are sent to the standard error output device.

Code Coverage Report Generator

Purpose

The Report Generator creates test reports from the Code Coverage data gathered during the execution of the application under analysis.

Syntax

```
attolcov {<fdc file>} {<traces>} [<options>]
```

where:

- <fdc files> The list of correspondence files for the application under test, with one file generated for each source file during instrumentation
- <traces> is a list of trace files. (default name **attolcov.tio**)
- <options> represents a set of options described below.

Parameters can use wild-card characters ('*' and '?') to specify multiple files. They can also contain absolute or relative paths.

Description

Trace files are generated when an instrumented program is run. A trace file contains the list of branches exercised during the run.

You can select one or more coverage types at the instrumentation stage.

All or some of the selected coverage types are then available when reports are generated.

The Report Generator supports the following coverage type options:

```
-PROC [=RET]
```

The **-PROC** option, with no parameter, reports procedure inputs.

Use the **RET** parameter to reports procedure inputs, outputs, and terminal instructions.

-CALL

Reports call coverage.

-BLOCK[=IMPLICIT | DECISION | LOGICAL | ATC]

The **-BLOCK** option, with no parameter, reports statement blocks only.

- **IMPLICIT** or **DECISION** (equivalent) reports implicit blocks (unwritten else and default blocks), as well as statement blocks.
- **LOGICAL** Reports logical blocks (loops, as well as statement and implicit blocks).
- **ATC** Reports asynchronous transfer control (ATC) blocks, as well as statement blocks, implicit blocks, and logical blocks.

-COND [=MODIFIED | COMPOUND]

The **-COND** option, with no parameter, reports basic conditions only.

MODIFIED reports modified conditions as well as basic conditions.

COMPOUND reports compound conditions as well as basic and modified conditions.

Explicitly Excluded Options

Each coverage type can also be explicitly excluded.

-NOPROC

Procedure inputs, outputs, or returns are not reported.

`-NOCALL`

Calls are not reported.

`-NOBLOCK`

Simple, implicit, or logical blocks are not reported.

`-NOCOND`

Basic conditions are not reported.

Additional Options

The following options are also available:

`-FILE=<file>{ [, <file>] }` | `-EXFILE=<file>{ [, <file>] }`

Specifies which files are reported or not. Use **-FILE** to report only the files that are explicitly specified or **-EXFILE** to report all files except those that are explicitly specified. Both **-FILE** and **-EXFILE** cannot be used together.

`-SERVICE=<service>{ [, <service>] }` | `-EXSERVICE=<service>{ [, <service>] }`

Specifies which functions, methods, and procedures are to be reported or not. Use **-SERVICE** to report only the functions, methods and procedures that are explicitly specified or **-EXSERVICE** to report all functions, methods, and procedures except those that are explicitly specified. Both **-SERVICE** and **-EXSERVICE** cannot be used together.

`-TEST=<test>{ [, <test>] }` | `-EXTTEST=<test>{ [, <test>] }`

Specifies which tests are reported or not. Use **-TEST** to report only the tests that are explicitly specified or **-EXTTEST** to report all tests except those that are explicitly specified. Both **-TEST** and **-EXTTEST** cannot be used together.

`-OUTPUT=<file>`

Specifies the name of the report file (*<file>*) to be generated. You can specify any filename extension and can include an absolute or relative path.

-LISTING [=<directory>]

This option requires annotated listings to be generated from the source files. Annotated listings carry the same name as their corresponding source files, but with the extension **.lsc**. The optional parameter *<directory>* is the absolute or relative path to the directory where the listings are to be generated. By default, a listing file is generated in the directory where its corresponding source file is located.

-NOGLOBAL

Reports the results of each test found in the trace file, followed by a conclusion summarizing all the tests. If a test has no name, it is identified as "#" in the report. A test is an execution of an instrumented application, a **TEST** as defined for Component Testing for C and Ada, or a dump-on-signal. By default, the report is not structured in terms of tests.

-BRANCH=COV

Reports branches covered rather than branches not covered. It does not affect listings, where only branches not covered are indicated with the source code line where they appear.

-SUMMARY=CONCLUSION | FILE | SERVICE

This option sets the verbosity of the summary:

- **CONCLUSION** reports only the overall conclusion.
- **FILE** reports only the conclusion for each source file, and the overall conclusion.
- **SERVICE** reports only the levels of coverage for each source file, each C function, and overall. The list of branches covered or not covered is not included.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

TDF Splitter

Purpose

For use with Runtime Tracing. The **.tdf** splitter (**attsplit**) tool allows you to separate large **.tdf** dynamic trace files into smaller more manageable files.

Syntax

```
attsplit [<options>] <tcf file> <tsf_file> <tdf file>
```

where:

- *<tcf_file>* is always **\$TESTRTDIR/lib/tracer.tcf**
- *<tsf_file>* is the name of the generated **.tsf** static trace file
- *<tdf_file>* is the name of the original **.tdf** dynamic trace file

Description

Trace **.tdf** files that contain loops cannot be split.

Options

```
-p <prefix>
```

Specifies the filename prefix for the split **.tdf** files. By default, split **.tdf** filenames start with **att**.

```
-s <bytes>
```

Sets the maximum file size for the split **.tdf** files. By default, the original **.tdf** dynamic trace file is split into 1000 byte split **.tdf** files

Specifies

`-v` | `-vw`

Activates verbose mode (**-v**) or verbose mode for written files only (**-vw**).

`-nt`

Disables the writing of time information. By default, time information is written to the split **.tdf** files.

`-fopt <filename>`

Uses a text file to pass options to the **attsplit** command line.

C Source Code Parser

Purpose

When creating a new Component Testing test campaign for C, the C Source Code Parser creates a C test script template based on the analysis of the source code under test.

Syntax

```
attolstartC <source_under_test > <test_script> [{<-option>}]  
attolstartC @<option file >
```

where:

- *<source under test>* this required parameter is the name of the source file to be tested.
- *<test script>* is the name of the test script that is generated
- *<options>* is a list of options as defined below.
- *<option file>* is the name of a plain-text file containing a list of options.

Description

The C Source Code Parser analyzes the source file to be tested in order to extract global variables and testable functions.

Each global variable is automatically declared as external, if this has not already been done at the beginning of the test script. Then, an environment is created to contain all these variables with default tests. This environment has the name of the file (without the extension).

For each function under test, the generator creates a **SERVICE** which contains the C declaration of the variables to use as parameters of the

function.

Parameters passed by reference are declared according to the following rule:

- *char** *<param>* causes the generation of *char <param>[200]*
- *<type>** *<param>* causes the generation of *<type> <param>* passing by reference

It is sometimes necessary to modify this declaration if it is unsuitable for the tested function, where *<type>** *<param>* can entail the following declarations:

- *<type>** *<param>* passing-by-value,
- *<type> <param>* passing-by-reference,
- *<type> <param>[10]* passing-by-reference.

File names can be related or absolute.

If the generated file name does not have an extension, the C Source Code Parser automatically attaches .ptu or the extension specified by the **ATTOLPTU** environment variable. This name may be specified relatively, in relation to the current directory, or as an absolute path.

If the test script cannot be created, the C Source Code Parser issues a fatal error and stops.

If the test script already exists, the previous version is saved under the name *<generated test script>_bck* and a warning message is generated.

Options

Options can be in any order. They may be upper or lowercase and can be

abbreviated to their shortest unambiguous number of characters.

Included Files

```
-insert
```

With this option the source file under test is included into the test script with an `#include` directive, ensuring that all the internal functions and variables (declared static) are visible to the test script. The C Source Code Parser adds the `#include` directive before the `BEGIN` instruction and after any `#includes` added by the `-use` option.

```
-use=<file used>{[, <file used>]}
```

This option gives the C Source Code Parser a list of header files to include in the test script before the `BEGIN` instruction. This avoids declaring variables or functions that have already been declared in a C header file of the application under test.

The C Source Code Parser adds the `#include` directive before the `BEGIN` instruction. Then, for each file, an environment is created, containing all variables with a default test. This environment has the name of the included file.

By default, no files are included in the test script.

Integrated Files

```
-integrate=<additional file>{[, <additional file>]}
```

This option provides a list of additional source files whose objects are *integrated* into the test program after linking.

The C Source Code Parser analyzes the additional files to extract any global variables that are visible from outside. For each global variable the Parser declares an external variable and creates a default test which is added to an

environment named after the corresponding additional file.

By default, any symbols and types that could be exported from the source file under test are declared again in the test script.

Simulated Files

```
-simulate=<simulated file>{[,<simulated file>]}
```

This option gives the C Source Code Parser a list of source files to simulate upon execution of the test. List elements are separated by commas and may be specified relatively, in relation to the current directory, or as an absolute path.

The Parser analyzes the simulated files to extract the global variables and functions that are visible from outside. For each file, a DEFINE STUB block, which contains the simulation of the file's external global variables and functions, is generated.

By default, no simulation instructions are generated.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

C Test Script Compiler

Purpose

The C Test Script Compiler tool pre-processes a **.ptu** test script and converts it into a native source code test harness.

Syntax

```
attolpreproC <test_script> <generated_file> [ <target_directory> ]  
{ [ <-options> ] }  
attolpreproC @<option_file>
```

where:

- *<test_script>* is a required parameter that specifies the name of the test program to be generated.
- *<generated_file>* is a required parameter that specifies the name of the test harness that is generated from the test script.
- *<target_directory>* is an optional parameter. It specifies the location where Component Testing for C will generate the trace file. By default, the trace file is generated in the workspace directory.
- *<options>* is a set of optional command line parameters as specified in the following section.
- *<option_file>* is the name of a plain-text file containing a list of options.

Description

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

Source File Under Test

```
-source=<source file>
```

This option specifies the name of the source file being tested, allowing the Test Script Compiler to:

- Maintain the source file name in the table of correspondence files so that the Test Report Generator can display this name in the header of the results obtained file.
- Establish the list of include files in the tested source file.

The name of the tested source file may be specified with a relative or absolute directory in a syntax recognized by the operating system, or, in UNIX, by an environment variable.

By default, the list of include files in the tested source file and the source file name are not displayed in the Results Obtained file.

Condition Definition

```
-define=<ident>[=<value>] {[, <ident>[=<value>]}
```

This option specifies conditions to be applied when the Test Script Compiler starts. These conditions allow conditional test harness generation as well as identifier definition for C.

The identifiers specified by the `-define` option apply conditions to the generation of any **IF ... ELSE ... END IF** blocks in the test script.

If the option is used with one of the conditions specified in the **IF** instruction, the **IF ... ELSE** block (if **ELSE** is present) or the **ELSE ... END IF** block (if **ELSE** is not present) is analyzed and generated. The **ELSE ... END IF** block is eliminated.

If the option is not used or if none of the conditions specified in the **IF** instruction are satisfied, the **ELSE ... END IF** block is analyzed and generated.

All symbols defined by this option are equivalent to the following line in C

```
-define <ident> [<value>]
```

By default, the **ELSE ... END IF** blocks are analyzed and generated.

Specifying Tests, Families, and Services

```
-test=<test>{ [, <test>] } | -extest=<test>{ [, <test>] }
```

This option specifies a list of tests to be executed.

Use `-test` to only generate the source code related to the specified tests, and `-extest` to specify the tests for which you do not want to generate source code.

Both `-test` and `-extest` cannot be used together.

By default, all tests are selected.

```
-family=<family>{ , <family> } | -exfamily=<family>{ , <family> }
```

Use `-family` to only generate the source code related to the specified families, and `-exfamily` to specify the families for which you do not want to generate source code.

Both `-family` and `-exfamily` cannot be used together.

By default, all families are selected.

```
-service=<service>{ [, <service>] } | -  
exservice=<service>{ [, <service>] }
```

Use `-service` to only generate the source code related to the specified services, and `-exservice` family to specify the services for which you do not want to generate source code.

Both `-service` and `-exservice` cannot be used together.

By default, all services are selected.

Test Script Parsing

```
-fast | -nofast
```

The `-fast` option tells the C Test Script Compiler to analyze only those tests that you want to generate. This setting considerably speeds up the Test Script Compiler when you use the `-service`, `-exservice`, `-family`, `-exfamily`, `-test`, or `-extest` options.

The `-fast` option is selected by default.

If you want a full test script analysis, this option can be de-selected using the **`-nofast`** option.

```
-noanalyse
```

This option disables the native language parser.

By default, native language lines are analyzed. This option enables you to disable this parsing.

```
-noedit
```

This option limits unit test code generation to the initialization of variables, making it possible to generate tighter code for special purposes such as debugging. If you specify the **`-noedit`** option, you cannot generate a test report.

By default, code is generated normally.

```
-nopath
```

Use this option if you do not want to generate long pathnames on the open and close execution trace file call, and on the Target Deployment Port header file include directive. This can be useful, for example, to preserve memory on embedded targets.

By default, full pathnames are generated.

```
-nosimulation
```

This option determines the conditional generation related to simulation in the source file generated by the Test Script Compiler. Blocks delimited by the keywords **SIMUL ... ELSE_SIMUL ... END SIMUL** can be included in the test scripts.

See SIMUL blocks in the C Test Script Language Reference.

```
-restriction=ANSI | KR | NOEXCEPTION | NOIMAGE | NOPOS | SEPAR
```

This option lets you modify the behavior of test script parser.

- **ANSI** enables C native code to be analyzed according to the ANSI standard (C only).
- **KR** enables C native code to be analyzed according to the KERNIGHAN & RITCHIE version 2 standard (C only).
- **noexception:** tells the Test Script Compiler to skip EXCEPTION blocks when generating a test harness. This allows the use of compilers that do not implement exception handling. By default, EXCEPTION blocks are generated in the test program.
- **noimage:** initialization, expected, and obtained values display as integers instead of character strings. By default, reports are generated with IMAGE attributes.
- **nopos:** modifies the way enumerated variables are displayed in the test report by not generating any POS or IMAGE attributes. Initialization and expected values are displayed as they are written in the test script,

whereas obtained values do not appear (although they are tested). Use this option to save memory on restricted target platforms. By default, reports are generated with IMAGE attributes.

- **separ:** modifies the format of the generated test program. In place of a main procedure including a sub-procedure for each service, the C Test Script Compiler generates one separate procedure for each service. With this restriction, the Test Script Compiler generates several compilation units and avoids overflow errors on compilation. By default, code is generated normally.

Several **-restriction** options can be used on the same command line. The ANSI and KR parameters, however, cannot be used together.

Using an Option File

```
@<parameter file>
```

This syntax allows the compiler to pass options to the preprocessor through a file. The parameter file name can be written in absolute or relative format.

The format of the file must follow these rules:

- One or more options can occur per line.
- Each option must follow the same syntax as the command line version, with the character that usually introduces the option being '-' under UNIX and '/' under Windows.
- You may not use both an option file and command line options.

By default, no file is taken into account.

If the option file is not found, a fatal error is generated and the preprocessor stops.

Examples

```
attolprepro C add.ptu Tadd.cpp -service=add -test=1,2,3 -  
family=nominal  
attolprepro CPP @add.opt
```

In this case, the parameter file **add.opt** would contain:

```
add.ptu Tadd.cpp  
-service=add  
-test=1,2,3  
-family=nominal
```

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

These codes help you decide on a course of action once the Test Script Compiler has finished test execution. For example, if the return code in the command file shows there have been incorrect tests, you can save certain files in order to analyze them later.

All messages are sent to the standard error output device.

C Test Report Generator

Purpose

The C Test Report Generator processes a trace file produced during test execution, and generates a test report.

Syntax

```
attolpostpro <root of trace file> <test report file> [<options>]  
attolpostpro @<option file>
```

where:

- *<root of trace file>* specifies the root of the trace file (filename without extension) that is generated when the program runs.
- *<test report file>* is the name of the test report to which the Test Report Generator will output the test results.
- *<options>* can be any of the optional parameters specified below.
- *<option_file>* is the name of a plain-text file containing a list of options.

Description

The Test Report Generator uses *<root of trace file>* to find out the names of both the **.rio** trace file and the **.tdc** table of correspondence file that are generated by the C Test Script Compiler.

If either *<root of trace file>* or *<test report name>* are omitted, execution terminates and Component Testing for C and Ada produces an error message.

If the trace file or the table of correspondence file does not exist, cannot be read, or contains synchronization errors, Component Testing for C

produces a fatal error message.

If the Test Report Generator cannot create *<test report file>*, generation of the report is terminated. If the file already exists, the newly generated file replaces the existing report.

If no extension for the report file is specified, the Test Report Generator attaches **.ro** or an extension specified in the **ATTOLRO** environment variable.

Options

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

```
-cio=<coverage result file>
```

This option allows you to insert coverage results in the report file. This option must be used only in conjunction with the Code Coverage feature.

```
-compare[=strict]
```

This option lets you compare the results from two test runs. A trace file generated during the first run has a **.rio** extension, and the one generated during the second run has a **.ri2** extension.

When making a comparison, the Test Report Generator generates the test report from:

- The **.tdc** table of correspondence file
- The **.rio** trace file generated during the first run
- The **.ri2** trace file generated during the second run

The same root name is used for the names of the three files.

When comparing values, a variable will only be deemed *correct* if the two obtained values are the same as the expected value, or within the specified validity interval for that variable. With the **compare=strict** option, the two results must have the same value.

```
-ra [=test | error]
```

This option specifies the form of the output report generated by the Test Report Generator.

Use **-ra** with no parameter, to display ALL test variables and mark any variables that are incorrect for a given test. This option is used by default.

Use the **-ra=test** option to display ALL test variables, with incorrect variables marked. This option provides a comprehensive display of variables for an incorrect test, which can prove useful in a complex test environment.

Use **-ra=error** to to display only erroneous test variables.

For both **-ra=test** and **-ra=error**, if no errors are detected in the test, only general information about the test is produced.

```
-va=eval | noeval | combined
```

This option lets you specify the way in which initial and expected values of each variable are displayed in the test report.

Use **-va=eval** if you want the test report to show the initial and expected value of each variable evaluated during execution of the test. This is only relevant for variables whose initial or expected value expressions are not reducible in the test script.

Note: For arrays and structures in which one of the members is an array, the initial values are not evaluated. For the expected values, only incorrect

elements are evaluated.

Use **-va=noeval** if you want the test report to show the initial and expected values described in the test script.

The **-va=combined** option combines both **eval** and **noeval** parameters. For each variable, the Report Generator includes the initial and expected values described in the test script, as well as the initial and expected values evaluated during execution, if these values differ.

By default, the **-va=eval** parameter is used.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

C++ Source Code Parser

Purpose

The C++ Source Code Parser takes a set of C++ source files containing classes to generate template **.otd** C++ Test Driver Scripts and **.otc** C++ Contract Check Scripts to fully cover the application under test.

Syntax

```
atostart { [-i] <tested file> } <options>
```

where:

- *<tested file>* is the list of files containing classes to be tested. If no class is specified with the option **-test_class**, the tested classes will be either the classes defined in a file under test, or the classes for which a method is defined in a tested file.
- *<options>* is a series of command line options. See the section **Options**.

If a tested file is specified with option **-i**, this file will be *included* by the generated **.otd** script. As a consequence everything defined in this tested file will be available in the script (especially types, classes, static variables, and functions). This option is ignored if you choose not to generate an **.otd** C++ Test Driver Script.

Description

The tested files and additional files (see option **-integrate**) are parsed by the integrated C++ analyzer. A candidate classes list is automatically deduced from the content of tested files (this list can be viewed in the header of the generated **.otd** and **.otc** scripts). If no **-test_class** or **-do_not_test_class** option is used, then all the candidate classes will have generated code to test them.

The C++ Source Code Parser generates only one **.otd** C++ Test Driver Script that contains all classes under test. It also generates two files associated to this test script: a **.dcl** declaration file (declaring and including every resource needed to compile the test script) and a **.stb** stub file (containing stub declarations deduced from used but not defined entities found in the parsed files).

The C++ Source Code Parser generates one **.otc** C++ Contract Check Script per encountered **.h** file defining a class.

Options

The C++ Source Code Parser supports the following options:

```
{-integrate <additional file>}
```

Specifies additional files or directories to be analyzed. These files do not contain any classes under test, but they do contain code which is to be linked with the tested application. Basically, this option tells the C++ Source Code Parser which files not to stub.

Three types of additional files or directories are supported :

- **body files:** Only the entities defined within the file are considered defined.
- **header files:** Every declaration within the file is considered as having a matching definition in a non-provided body file or in a library. Use additional header files when linking to code for which the source is not available.
- **directories:** Every declaration found in a file belonging to an additional directory is considered as having a matching definition in a non-provided body file or in a library (an additional directory can be viewed as a collection of additional header files).

Note A header file is recognized as such from its content, and not from its extension. A header file does not contain any definition, other than inline functions, and template functions, or else it is considered as a body file.

This option is ignored when no **.otd** generation is required.

```
-o <test script>
```

Specifies the name of the generated **.otd** script. Two associated files are also generated with the same name, but with extension **.dcl** and **.stb**. If the filename extension of <test script> is not **.otd**, then a warning is issued.

This option is ignored when no **.otd** generation is required.

```
-otcdir <OTC directory>
```

Specifies the directory where **.otc** files are to be generated.

This option is ignored when no **.otc** generation is required.

```
-opp <compiler option file>
```

Specifies the name of the Target Deployment Port C++ parser option file. This file is searched for in **/ana** subdirectory of the current Target Deployment Port (see ATLTGT environment variable), and should not include any path.

If this option is not provided, the default filename **atl.opp** will be searched for.

```
-hpp <compiler configuration file>
```

Specifies the name of the Target Deployment Port C++ parser configuration file. This file is searched for in **/ana** subdirectory of the current Target Deployment Port (see ATLTGT environment variable), and should not

include any path.

If this option is not provided, the default filename **atl.hpp** will be searched for.

```
{-test_class|-tc <class under test>}
```

Specifies the classes to be explicitly tested. The classes must belong to the candidate classes. This option cannot be used simultaneously with the options **-do_not_test_class (-dnc)**.

```
{-do_not_test_class|-dnc} <excluded class>
```

Specifies the classes, among the candidate classes, which should not be tested. This option cannot be used simultaneously with the options **-test_class (-tc)**.

```
-test_struct
```

Specifies whether *structs* and *unions* should be treated as *classes*, and therefore should be considered as potential tested classes. This option is not significant when **-test_class** option is used (you can specify *structs* or *unions* as *classes* to be tested).

```
-test_each_instance
```

By default, a template class is tested as a generic template class. Use this option if you want to generate a specific test for each found instance of a template class.

```
{-force_template <template instance>}
```

This option forces the instantiation of the specified templates classes. Use it if no automatic template instantiation occurs while parsing the code. This option is useful only in conjunction with **-test_each_instance** option.

```
-overwrite
```

By default, the Test Template Generator creates a backup file of every file it overwrites. Use this option if you really intend to overwrite these files without backing up them.

```
-ignore_line_directives
```

Usually the generator parse *#line* directives, so that you can use your own preprocessor instead of the built-in preprocessor. If required, deactivate this setting with the **-ignore_line_directives** option. This can be useful when parsing automatically generated code.

This option is ignored when no **.otd** generation is required.

```
{-I<include directory>}
```

This option specifies directories where included files are to be searched for. You can use the option **-I-** to introduce the system *includes*: only directories specified after **-I-** will be looked up when the include directives use angular brackets (**#include <...>**).

```
{-D<macro> [=<value>]}
```

This option adds a predefinition for *<macro>* to *<value>*.

```
-E
```

This options generates preprocessing output to standard output. This option is mainly for debugging purpose.

```
-include={relative|absolute|copy}
```

This option specifies how *#include* directives should be generated in the test script. When *relative* is chosen, includes use relative path to the directory where the generated script is put. When *absolute* is chosen, absolute paths are generated. When *copy* is chosen, the way files are included in the test script is the same as they are included in the tested files, you should in this case ensure that the test script is generated in the same directory than the

source files.

This option is ignored when no **.otd** generation is required.

```
-no_otc
```

This option deactivates **.otc** script generation. Use this option if you only want an **.otd** test driver script.

```
-no_otd
```

This option deactivates **.otd** script generation. Use this option if you only want an **.otc** Contract-Check script.

Note If no candidate class is found, nothing will be generated.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
1	End of execution with error

All messages are sent to the standard error output device.

C++ Test Script Compiler

Purpose

The C++ Test Script Compiler compiles the **.otd** C++ Test Driver Script and **.otc** C++ Contract Check Scripts into C++ source code.

Syntax

```
atoprepro [<OTD Script>] | [<OTC Scripts>] -G C++ -O <cpp file> -OTI  
<oti file> -TDF <tdf file>
```

where:

- *<otd script>* is an **.otd** C++ Test Driver Script file.
- *<otc scripts>* is a set of one or more **.otc** C++ Contract Check Script files.
- *<cpp file>* is the name of the **.cc** or **.cpp** source file to be generated by Component Testing for C++ and linked to the application under test.
- *<oti file>* is the name of the **.oti** instrumentation file to be generated. This file is used by the C++ Instrumentor.
- *<tdf file>* is the **.tdf** dynamic trace file to be generated during the execution of the application under test.

Options

The C++ Source Code Parser supports the following options:

```
-E <number of errors>
```

Specifies the maximum number of error messages that can be displayed by the C++ Test Script Compiler. The default value is 30.

```
-NODLINE
```

Deactivates the generation of *#line* statements. This can be useful in

environments where the generated source code cannot use the *#line* mechanism. By default *#line* statements are generated.

-NOPATH

This option tells the C++ Test Script Compiler not to use the full path to the TDP from the **\$ATLTGT** environment variable before the name of **TP.h** in the *#include* directive.

This option is useful for embedded targets when compilation of the generated source does not occur on the same host as the C++ test compilation.

Example

```
atoprepro script.otd contract1.otc contract2.otc -G C++ -O
app.cc -OTI foo.oti -TDF bar.tdf -E 60
```

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
1	Abnormal termination

All messages are sent to the standard error output device.

C++ Test Report Generator

Purpose

The C++ Test Report Generator processes a trace file produced during test execution, and generates a test report that can be viewed in the GUI.

Syntax

```
atopospro -ots {<ots files>} -tdf <tdf file> -xrd <xrd file>
```

where:

- *<ots files>* is a list of **.ots** intermediate files generated by the C++ Test Script Compiler.
- *<tdf file>* is the **.tdf** dynamic trace file generated during the execution of the application under test.
- *<xrd file>* is the **.xrd** report file to be generated by the Report Generator.

Example

```
atopospro -ots script.ots contract1.ots contract2.ots -tdf  
bar.tdf -xrd report.xrd
```

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
1	Abnormal termination

All messages are sent to the standard error output device.

Java Source Code Parser

Purpose

The Source Code Parser for Component Testing for Java analyzes a set of Java source files that contain classes, and produces a test harness template and metrics.

Syntax

```
javastart <java source files> <options>
```

where:

- *<java source file>* is the list of files containing classes to be tested.
- *<options>* is a series of command line options. See the section Options.

Description

The Java source files are parsed by the integrated Java analyzer. A candidate classes list is automatically deduced from the content of source files.

The Metric Generator generates one metric file for each source file in the *<java source file>* list.

The Source Code Parser generates only one Java test driver script that contains all classes under test. It also generates stub file containing stub declarations for classes specified in the **-stub** option.

Options

```
-J2SE | -J2ME
```

Specifies the Java target testing framework. The default framework is J2ME.

`-classpath <val>`

Sets the EDGCLASSPATH value to *<val>*. EDGCLASSPATH is the environment variable used by the parser to search for Java classes.

`-typical`

With this option, all used classes are stubbed.

`-pref <prefix>`

Use this option to change the prefix of generated test class names. The default prefix is **Test**.

`-metric`

Metrics will be generated for classes in Java source files list.

`-odir <output directory>`

Specifies the directory where results files are to be generated.

`-test_class | -tc <java classes>`

Specifies the classes to be explicitly tested. List of classes could be specified by giving Java source file or class names.

`-o <file name>`

Specifies the name of the generated test. This option is ignored when no test generation is required (see option `-test_class`).

`-stub_class | -sc <java classes>`

Specifies the classes to be explicitly stubbed. List of classes could be specified by giving Java source file or class names.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
1	Abnormal termination

All messages are sent to the standard error output device.

Java Test Report Generator

Purpose

The Java Test Report Generator processes a trace file produced during test execution, and generates a test report that can be viewed in the Test RealTime GUI.

Syntax

```
javapostpro -tsf<trace files> -tdf<dynamic trace file> -xrd<report file>
```

where:

- *<trace files>* is a series of input **.tsf** and **.tdf** intermediate files produced during execution of the test driver
- *<dynamic trace file>* is the input **.tdf** dynamic trace file produced during execution of the test driver
- *<report file>* is the output **.xrd** test report file to be generated

Description

The **-tsf** option takes a list of generated **.tsf** static trace files. However, in Component Testing for Java, the **.tdf** dynamic trace files may contain structural data as well as trace data. Therefore, the **.tdf** file must be specified both in the **-tsf** option and the **-tdf** option.

If a specified **.tsf** or **.tdf** file does not exist, cannot be read, or contains synchronization errors, the Test Report Generator produces a fatal error message.

If the Test Report Generator cannot create the **.xrd** report file, generation of the report is terminated. If the file already exists, the newly generated file

replaces the existing report.

Example

```
javapostpro -tsf testClass1.tsf testClass2.tsf  
testDriver.tdf -tdf testDriver.tdf -xrd report.xrd
```

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
1	Abnormal termination

All messages are sent to the standard error output device.

Ada Source Code Parser

Purpose

When creating a new Component Testing test campaign for Ada, the Ada Source Code Parser creates an Ada test script template based on the analysis of the source code under test.

Syntax

```
attolstartADA <source_under_test > <test_script> [{<-option>}]  
attolstartADA @<option file >
```

where:

- *<source under test>* this required parameter is the name of the source file to be tested.
- *<test script>* is the name of the test script that is generated
- *<options>* is a list of options as defined below.
- *<option file>* is the name of a plain-text file containing a list of options.

Description

The Ada Source Code Parser analyzes the source file to be tested in order to extract global variables and testable functions.

Each global variable is automatically declared as external, if this has not already been done at the beginning of the test script. Then, an environment is created to contain all these variables with default tests. This environment has the name of the file (without the extension).

For each function under test, the generator creates a SERVICE which contains the Ada declaration of the variables to use as parameters of the

function.

Parameters passed by reference are declared according to the following rule:

- *char** <param> causes the generation of *char* <param>[200]
- <type>* <param> causes the generation of <type> <param> passing by reference

It is sometimes necessary to modify this declaration if it is unsuitable for the tested function, where <type>* <param> can entail the following declarations:

- <type>* <param> passing-by-value,
- <type> <param> passing-by-reference,
- <type> <param>[10] passing-by-reference.

File names can be related or absolute.

If the generated file name does not have an extension, the Ada Source Code Parser automatically attaches **.ptu** or the extension specified by the ATTOLPTU environment variable. This name may be specified relatively, in relation to the current directory, or as an absolute path.

If the test script cannot be created, the Ada Source Code Parser issues a fatal error and stops.

If the test script already exists, the previous version is saved under the name <generated test script>_bck and a warning message is generated.

Options

Options can be in any order. They may be upper or lowercase and can be

abbreviated to their shortest unambiguous number of characters.

Included Files

```
-insert
```

With this option the source file under test is included into the test script with an `#include` directive, ensuring that all the internal functions and variables (declared static) are visible to the test script. The Ada Source Code Parser adds the `#include` directive before the `BEGIN` instruction and after any `#includes` added by the `-use` option.

Additional Files

```
-integrate=<additional file>{[,<additional file>]}
```

This option provides a list of additional source files whose objects are *integrated* into the test program after linking.

The Ada Source Code Parser analyzes the additional files to extract any global variables that are visible from outside. For each global variable the Parser declares an external variable and creates a default test which is added to an environment named after the corresponding additional file.

By default, any symbols and types that could be exported from the source file under test are declared again in the test script.

Simulated Files

```
-simulate=<simulated file>{[,<simulated file>]}
```

This option gives the Ada Source Code Parser a list of source files to simulate upon execution of the test. List elements are separated by commas and may be specified relatively, in relation to the current directory, or as an absolute path.

The Parser analyzes the simulated files to extract the global variables and functions that are visible from outside. For each file, a DEFINE STUB block, which contains the simulation of the file's external global variables and functions, is generated.

By default, no simulation instructions are generated.

Header Files

```
-use=<file used>{[,<file used>]}
```

This option gives the Ada Source Code Parser a list of header files to include in the test script before the BEGIN instruction. This avoids declaring variables or functions that have already been declared in an Ada header file of the application under test.

The Ada Source Code Parser adds the #include directive before the BEGIN instruction. Then, for each file, an environment is created, containing all variables with a default test. This environment has the name of the included file.

By default, no files are included in the test script.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

Ada Test Script Compiler

Purpose

The Ada Test Script Compiler tool pre-processes the Ada test script and converts it into a native source test harness.

Syntax

```
attolpreproADA <test_script> <generated_file> [ <target_directory>
] { [ <-options> ] }
attolpreproADA @<option_file>
```

where:

- *<test_script>* is a required parameter that specifies the name of the test program to be generated.
- *<generated_file>* is a required parameter that specifies the name of the test harness that is generated from the test script.
- *<target_directory>* is an optional parameter. It specifies the location where Component Testing for Ada will generate the trace file. By default, the trace file is generated in the workspace directory.
- *<options>* is a set of optional command line parameters as specified in the following section.
- *<option_file>* is the name of a plain-text file containing a list of options.

Description

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

Source File Under Test

```
-source=<source file>
```

This option specifies the name of the source file being tested, allowing the Test Script Compiler to:

- Maintain the source file name in the table of correspondence files so that the Test Report Generator can display this name in the header of the results obtained file.
- Establish the list of include files in the tested source file.

The name of the tested source file may be specified with a relative or absolute directory in a syntax recognized by the operating system, or, in UNIX, by an environment variable.

By default, the list of include files in the tested source file and the source file name are not displayed in the Results Obtained file.

Condition Definition

```
-define=<ident>[=<value>] {[, <ident>[=<value>]}
```

This option specifies conditions to be applied when the Test Script Compiler starts. These conditions allow conditional test harness generation as well as identifier definition for Ada.

The identifiers specified by the `-define` option apply conditions to the generation of any **IF ... ELSE ... END IF** blocks in the test script.

If the option is used with one of the conditions specified in the **IF** instruction, the **IF ... ELSE** block (if **ELSE** is present) or the **ELSE ... END IF** block (if **ELSE** is not present) is analyzed and generated. The **ELSE ... END IF** block is eliminated.

If the option is not used or if none of the conditions specified in the **IF** instruction are satisfied, the **ELSE ... END IF** block is analyzed and generated.

All symbols defined by this option are equivalent to the following line in Ada

```
-define <ident> [<value>]
```

By default, the **ELSE ... END IF** blocks are analyzed and generated.

Specifying Tests, Families, and Services

```
-test=<test>{ [, <test> ] } | -extest=<test>{ [, <test> ] }
```

This option specifies a list of tests to be executed.

Use `-test` to only generate the source code related to the specified tests, and `-extest` to specify the tests for which you do not want to generate source code.

Both `-test` and `-extest` cannot be used together.

By default, all tests are selected.

```
-family=<family>{ , <family> } | -exfamily=<family>{ , <family> }
```

Use `-family` to only generate the source code related to the specified families, and `-exfamily` to specify the families for which you do not want to generate source code.

Both `-family` and `-exfamily` cannot be used together.

By default, all families are selected.

```
-service=<service>{ [, <service> ] } | -  
exservice=<service>{ [, <service> ] }
```

Use `-service` to only generate the source code related to the specified services, and `-exservice` family to specify the services for which you do not want to generate source code.

Both `-service` and `-exservice` cannot be used together.

By default, all services are selected.

Test Script Parsing

```
-fast | -nofast
```

The `-fast` option tells the Test Script Compiler to analyze only those tests that you want to generate. This setting considerably speeds up the Test Script Compiler when you use the `-service`, `-exservice`, `-family`, `-exfamily`, `-test`, or `-extest` options.

The `-fast` option is selected by default.

If you want a full test script analysis, this option can be de-selected using the **`-nofast`** option.

```
-noanalyse
```

This option disables the native language parser.

By default, native language lines are analyzed. This option enables you to disable this parsing.

```
-noedit
```

This option limits unit test code generation to the initialization of variables, making it possible to generate tighter code for special purposes such as debugging. If you specify the **`-noedit`** option, you cannot generate a test report.

By default, code is generated normally.

```
-nopath
```

Use this option if you do not want to generate long pathnames on the open and close execution trace file call, and on the Target Deployment Port header file include directive. This can be useful, for example, to preserve memory on embedded targets.

By default, full pathnames are generated.

```
-nosimulation
```

This option determines the conditional generation related to simulation in the source file generated by the Test Script Compiler. Blocks delimited by the keywords **SIMUL ... ELSE_SIMUL ... END SIMUL** can be included in the test scripts.

See SIMUL blocks in the Ada Test Script Language.

```
-restriction=ANSI | KR | NOEXCEPTION | NOIMAGE | NOPOS | SEPAR
```

This option lets you modify the behavior of test script parser.

- **noexception:** tells the Test Script Compiler to skip EXCEPTION blocks when generating a test harness. This allows the use of compilers that do not implement exception handling. By default, EXCEPTION blocks are generated in the test program.
- **noimage:** initialization, expected, and obtained values display as integers instead of character strings. By default, reports are generated with IMAGE attributes.
- **noapos:** modifies the way enumerated variables are displayed in the test report by not generating any POS or IMAGE attributes. Initialization and expected values are displayed as they are written in the test script, whereas obtained values do not appear (although they are tested). Use this option to save memory on restricted target platforms. By default, reports are is generated with IMAGE attributes.
- **separ:** modifies the format of the generated test program. In place of a

main procedure including a sub-procedure for each service, the Test Script Compiler generates one separate procedure for each service. With this restriction, the Test Script Compiler generates several compilation units and avoids overflow errors on compilation. By default, code is generated normally.

Several -restriction options can be used on the same command line. The ANSI and KR parameters, however, cannot be used together.

Using an Option File

```
@<parameter file>
```

This syntax allows the compiler to pass options to the preprocessor through a file. The parameter file name can be written in absolute or relative format.

The format of the file must follow these rules:

- One or more options can occur per line.
- Each option must follow the same syntax as the command line version, with the character that usually introduces the option being '-' under UNIX and '/' under Windows.
- You may not use both an option file and command line options.

By default, no file is taken into account.

If the option file is not found, a fatal error is generated and the preprocessor stops.

Examples

```
attolprepro C add.ptu Tadd.cpp -service=add -test=1,2,3 -  
family=nominal  
attolprepro CPP @add.opt
```

In this case, the parameter file **add.opt** would contains:

```
add.ptu Tadd.cpp
-service=add
-test=1,2,3
-family=nominal
```

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

These codes help you decide on a course of action once the Test Script Compiler has finished test execution. For example, if the return code in the command file shows there have been incorrect tests, you can save certain files in order to analyze them later.

All messages are sent to the standard error output device.

Ada Test Report Generator

Purpose

The Ada Test Report Generator processes a trace file produced during test execution, and generates a test report.

Syntax

```
attolpostproADA <root of trace file> <test report file> [<options>]
attolpostproADA @<option file>
```

where:

- *<root of trace file>* specifies the root of the trace file (filename without extension) that is generated when the program runs.
- *<test report file>* is the name of the test report to which the Test Report Generator will output the test results.
- *<options>* can be any of the optional parameters specified below.
- *<option_file>* is the name of a plain-text file containing a list of options.

Description

The Test Report Generator uses *<root of trace file>* to find out the names of both the **.rio** trace file and the **.tdc** table of correspondence file that are generated by the Test Script Compiler.

If either *<root of trace file>* or *<test report name>* are omitted, execution terminates and Component Testing for Ada produces an error message.

If the trace file or the table of correspondence file does not exist, cannot be read, or contains synchronization errors, Component Testing for Ada produces a fatal error message.

If the Test Report Generator cannot create *<test report file>*, generation of the report is terminated. If the file already exists, the newly generated file replaces the existing report.

If no extension for the report file is specified, the Test Report Generator attaches **.ro** or an extension specified in the **ATTOLRO** environment variable.

Options

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

```
-cio=<coverage result file>
```

This option allows you to insert coverage results in the report file. This option must be used only in conjunction with the Code Coverage feature.

```
-ra[=test | error]
```

This option specifies the form of the output report generated by the Test Report Generator.

Use **-ra** with no parameter, to display ALL test variables and mark any variables that are incorrect for a given test. This option is used by default.

Use the **-ra=test** option to display ALL test variables, with incorrect variables marked. This option provides a comprehensive display of variables for an incorrect test, which can prove useful in a complex test environment.

Use **-ra=error** to to display only erroneous test variables.

For both **-ra=test** and **-ra=error**, if no errors are detected in the test, only general information about the test is produced.

This option lets you specify the way in which initial and expected values of each variable are displayed in the test report.

Use **-va=eval** if you want the test report to show the initial and expected value of each variable evaluated during execution of the test. This is only relevant for variables whose initial or expected value expressions are not reducible in the test script.

Note: For arrays and structures in which one of the members is an array, the initial values are not evaluated. For the expected values, only incorrect elements are evaluated.

Use **-va=noeval** if you want the test report to show the initial and expected values described in the test script.

The **-va=combined** option combines both **eval** and **noeval** parameters. For each variable, the Report Generator includes the initial and expected values described in the test script, as well as the initial and expected values evaluated during execution, if these values differ.

By default, the **-va=eval** parameter is used.

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors

7 End of execution because of fatal error

9 End of execution because of internal error

All messages are sent to the standard error output device.

System Testing Script Compiler

The System Testing Script Compiler preprocesses the Test Script and converts it into a native source test harness.

Syntax

```
atsprepro <test_script> <interface_file> {[,<interface_file>]}  
<file_name> [<options>]
```

where:

- *<test_script>* is the test script to be compiled.
- *<interface file>* lists interface files that contain event structure definitions, *includes* for interface prototypes, and their types. These files may have any extension.

Note If read access to these files is denied, System Testing for C produces a fatal error.

- *<file name>* is the name of the C code file generated from the test script. If you do not specify an extension, the system uses the **ATS_SRC** environment variable extension, or the default extension **.c**.
- *<options>* is a set of optional parameters among those described below.

Description

If you do not specify an extension, the system uses the **ATS_PTS** environment variable extension or the default **.pts** extension.

If an input file is absent or read access is denied, System Testing for C produces a fatal error.

After execution, the code is generated in the **code.c** file. If it is not possible to create the file, you will receive a fatal error.

If the Report Generator detects incorrect tests, System Testing for C produces a warning message.

If the report detects a synchronization error between the **.tdc** and the **.rio** file, System Testing for C produces a fatal error.

Optional Parameters

Options can be in any order. They may be upper or lowercase and can be abbreviated to their shortest unambiguous number of characters.

```
-ALLOCATION[ =STACK | =DYNAMIC ]
```

This option allows you to specify the method for allocating the work of the test program in the compiler.

If this option is present, the test program uses only allocated data on the execution stack (=STACK).

By default, the work context is global static data.

```
-BUFSIZE=<size>
```

This option sets the size of the trace buffer in kilobytes. The trace buffer is only used with the **-TRACE** option.

The default buffer size is 10KB.

```
-DEFINE=<list of conditions>
```

This option lets you specify the conditions to apply during test compilation. This option is equivalent to compiler option **-D**.

You can specify particular conditions or give them a value (-define=condition=value). Symbols defined with this option are equivalent to

the following line in C:

```
#define <symbol> [ <value> ]
```

```
-FAMILY=<family> { [, <family>] } | -EXFAMILY=<family> { [, <family>] }
```

-FAMILY specifies the only test families that are to be explicitly executed. Any other test families are ignored.

-EXFAMILY explicitly specifies the families that are to be ignored. All other families are executed.

-FAMILY and **-EXFAMILY** cannot be used together. The test compiler generates a warning message if no scenarios are generated.

By default, all test families are executed.

```
-SCN=<scenario> { [, <scenario>] } | -EXSCN=<scenario> { [, <scenario>] }
```

-SCN specifies the only scenarios that are to be explicitly executed. Any other scenarios are ignored.

-EXSCN explicitly specifies the scenarios that are to be ignored. All other scenarios are executed.

-SCN and **-EXSCN** cannot be used together.

To specify a sub-scenario, name the set of scenarios in which it is included and separate with full stops. If you exclude a scenario that contains sub-scenarios, all its sub-scenarios are also excluded.

The test compiler generates a warning message if no scenarios are generated.

```
-FAST | -NOFAST
```

The **-FAST** option tells the test compiler to analyze only those scenarios that

you want to generate. This option accelerates execution of the test compiler if you use a selection option. The option is useful when using **-SCN**, **-EXSCN**, **-FAMILY**, **-EXFAMILY**.

The **-NOFAST** option disables this behavior.

By default, the **-FAST** option is used.

```
-INCL=<directory> { [, <directory> ] }
```

This option lists directories where included files are located. Using this option enables you to:

- Establish the list of include files in the tested source file
- Execute the **INCLUDE** instructions
- Execute the C #include instruction

The system first searches the current directory, next in the directories specified with the **-INCL** option, and finally the default C system files directory.

```
-LANG=C
```

This option allows you to select the language of the generated code. You can generate C virtual testers.

By default, virtual testers are generated in C.

```
-LOG | -NOLOG
```

With the **-LOG** option, the system displays and stores errors found during the analysis of interface files and test script. The name of the log file is the name of the test script with the **.lis** extension.

If you select **-NOLOG**, these errors are not displayed.

By default, the **-LOG** option is used.

-NOCOMMENT

Use this option to deactivate the processing of **COMMENT** statements in order to improve performance issues.

-NOTSHARED

This option allows you to disable sharing of global static data between instances. When using this option, you must apply different names to all global variables within a test script. No local variable, constant, or function parameter should have the same name as a global static variable in the test script.

This used only by the **-ALLOCATION** and **-THREAD** options.

By default global variables of the test script are shared by all instances.

-STD_DEFINE=<standard definitions file>

This option provides the C parser with a C source file describing the characteristics of the compiler used.

If the specified file cannot be found, the test compiler stops and you will receive a fatal error.

By default, no compiler characteristics are specified.

-THREAD [=<function name>]

This option allows you to create a test function with a name other than **main**.

If <function name> is omitted, the function name becomes the source file name appended with **_start**.

By default, the generated function is called **main**.

```
-TRACE=CIRCULAR | ERROR | SCN | TIME
```

The test compiler uses a buffer to store the result of the test script execution. This buffer is saved on disk each time selected events (**ERROR**, **SCN**, **TIME**) occur. This option reduces the size of the virtual tester execution file. It is most useful during an endurance test.

- **-TRACE=CIRCULAR** tells the virtual tester to use a circular buffer to store execution traces. The circular buffer stores the execution traces in memory. Traces are flushed into the **.rio** file only after virtual tester execution or if explicitly requested in the test script (see the **FLUSH_TRACE** keyword).
- **-TRACE=ERROR** saves the buffer each time a test script error occurs.
- **-TRACE=SCN** has the same functionality as the **ERROR** parameter, and additionally saves scenario begin and end marks.
- **-TRACE=TIME** has the same functionality as the **SCN** parameter; and additionally saves timed events (**WAITTIL** and **PRINT**).

These options generate incomplete reports - some information is filtered - but the report always includes plan test errors.

If the buffer is too small, some traces are lost and the generated report is incomplete. You can change buffer size with the **-BUFSIZE** option.

Examples

```
atsprepro gen.pts interface.h code -  
EXSCN=Main.send.test_1, Main.receive.test_1
```

Return Codes

After execution, the program exits with the following return codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

System Testing Report Generator

Syntax

```
atsmerge <file> {[, <file> ]} [<options>]
```

where:

- *<file>* lists the **.rio** intermediate result files generated during the virtual tester execution phase and the **.tdc** correspondence table files generated during compilation.
- *<options>* is a list of options described below.

Description

The system generates a **.rod** result file for each **.rio** file, which is saved in the **rio** directory. The **.rod** filename uses the **.rio** filename with a **.rod** extension.

If one of the files cannot be found, the Report Generator produces a fatal error.

The Report Generator produces a warning message each time it encounters any incorrect data.

If the report contains any synchronization errors between the **.tdc** and the **.rio** file, the Report Generator produces a fatal error.

Options

The options can be in any order. They may be upper or lower case and written in an incomplete form, provided the selected option is clear.

```
-TIME
```

This option enables you to merge reports that do not contain structure

instructions. Structural instructions are beginning and ending block instructions (scenario, initialization, exception, termination).

If the **.rio** and **.tdc** files come from different test scripts, the **-TIME** option is enabled.

```
-RDD = <RDD report filename>
```

This option enables you to specify the output report filename.

By default, the report is names **atsrdd.rdd** and generated in the current directory.

```
-RA [=ERR | =TEST]
```

This option specifies the form of the report generated.

With **-RA = TEST**, all test variables are displayed and incorrect variables are marked. This option provides a comprehensive display of variables for an incorrect test, which can be useful in a complex debugging environment.

With **-RA = ERR**, only incorrect test variables are displayed.

In both cases, if the test is correct, only general information on this test is displayed.

The default option is **-RA** (with no parameters), which provides a full report of all variables for each test.

```
-VA =EVAL | NOEVAL | COMBINE
```

This option lets you specify the way in which initial and expected values of each variable is displayed in the test report.

- With **-VA = EVAL**, the initial, expected value of each variable evaluated during execution is displayed in the report. This option is only visible

for variables whose initialization or expected value is not reduced in the test script.

Note: For structures in which one of the fields is an array, this evaluation is not given for the initial values. For expected values, it is only given for incorrect elements.

- With **-VA = NOEVAL**, for each variable, the report generator displays in the test report the initial and expected values described in the test script.
- Use **-VA = COMBINE** to combines the previous two options, that is, for each variable, the report generator displays in the test report the initial and expected values described in the test script as well as the initial and expected values evaluated during execution.

By default **-VA = EVAL** is used.

-SUMMARY | NOSUMMARY

This option produces a summary of the test execution in the test report.

This option gives a quick overview of the execution of the set of test scenarios. It only summarizes the execution of the test scenarios.

The default option is **-NOSUMMARY**.

-COMMENT | -NOCOMMENT

In the System Testing Language, the **COMMENT** keyword displays a comment in the test report. You can use **-NOCOMMENT** to disable these comments, and **-COMMENT** to make them visible.

By default comments are displayed.

Log File

-LOG | -NOLOG

With the **-LOG** option, errors found during analysis of **.rio** and **.tdc** files are displayed on screen. Use the **-NOLOG** option to disable this behavior.

By default the **-LOG** option is used.

Example

```
atsmerge fic01.rio fic02.rio fic01.tdc fic02.tdc ...
```

Return Codes

After execution, the program exits with the following return codes:

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error
9	End of execution because of internal error

All messages are sent to the standard error output device.

Probe Processor

When using the Probe runtime analysis feature of the product, the Probe Processor takes the instrumented source files and the message definition files and generates a set of source files containing the definition of the message trace functions called by the probe environment.

Syntax

```
parsecode.pl [<options>] {<msg_files> [, <msg_files>]} {<source>
[, <source>]}
```

where:

- <msg_files> are **.h** message type definition files
- <source> are probed C source files
- <options> is a set of optional parameters among those described below.

Definition

The Probe Processor is tightly related with the System Testing feature for C.

By default, the message traces are written to the **.rio** System Testing output file for C. In this case, the probed binary is produced by compiling the **atlprobe.c** file generated file by the Processor and linking the application object files with the **atlprobe.o** file and the Target Deployment Port.

Optional Parameters

```
[-mode=DEFAULT | CUSTOM]
```

Custom mode allows the probes to write traces to a temporary location, such as memory, tubes or a buffer file. In custom mode, the traces are flushed to the **.rio** file only when an **atl_dump_trace** macro is encountered.

The I/O functions for probe trace output to the temporary location are defined in the **probecst.c** source file delivered with the product. You can modify this file to adapt the probe mechanism to your application and platform.

In custom mode, the compilation and link phase generates write operations from the probed application and the **probecst.c** file, and read operations from the **atprobe.c**, **probecst.c** files and the **TP.o** Target Deployment Port file.

```
[-preopts=-INCL=<include directories>]
```

The **-preopts** option allows you to send a list of include directories specified with a C Test Script Compiler **-INCL** option. See the C Test Script Compiler **-INCL** option.

```
[-outdir=<output directory>]
```

This option allows you to specify the target directory for the **.rio** System Testing output file for C.

System Testing Supervisor

Purpose

The System Testing Supervisor executes **.spv** supervisor script files.

Syntax

```
atsspvs <spv_script> <options>
```

where:

- *<spv_script>* is the **.spv** supervisor script to execute
- *<options>* is a series of command line options. See the section Options.

Description

System Testing manages the simultaneous execution of Virtual Testers distributed over a network. When using System Testing, the job of the Supervisor is to:

- Set up target hosts to run the test
- Launch the Virtual Testers, the system under test and any other tools.
- Synchronize Virtual Testers during execution
- Retrieve the execution traces after test execution

The System Testing Supervisor uses an **.spv** supervisor deployment script to control System Testing Agents installed on each distributed target host. Agents can launch either applications or Virtual Testers.

While the agent-spawned processes are running, their standard and error outputs are redirected to the supervisor.

Note You must install and configure the agents on the target machines

before execution.

The Supervisor generates traces during analysis and execution. These traces are displayed on the screen and written to a log file named as `<spv_script>.lis`.

Options

The options can be in any order. They may be upper or lower case and written in an incomplete form, provided the selected option is clear.

`-CHECK`

This option specifies that the scenario is to be analyzed but not executed. This allows you to check for errors in the `.spv` script.

`-NOLOG`

Disables supervisor output of error messages and warnings to the screen. Traces are still written to the `.lis` log file.

`-STUDIO_MACH=localhost`

By default, the supervisor uses the IP address 127.0.0.1 to connect to the Test RealTime graphical user interface. Use `-STUDIO_MACH= localhost` to resolve problems when the supervisor fails to connect.

Return Codes

Code	Description
0	End of execution with no errors
3	End of execution with one or more warning messages
5	End of execution with one or more errors
7	End of execution because of fatal error

Virtual Tester

Syntax

```
<virtual tester> [-INSTANCE=<instance>] [-OCCID=<id number>] [-RIO=<trace_file>]
```

Description

Virtual testers are multiple contextual incarnations of a single **.pts** System Testing test script.

One virtual tester can be deployed simultaneously on one or several targets, with different test configurations. A same virtual tester can also have multiple clones on the same target host machine.

Deployment of virtual testers is controlled by either the GUI or a System Testing **.spv** supervisor script when running in the command line interface. Do not edit **.spv** scripts when using the GUI.

System Testing for C generates virtual testers from a test script according to the declared instances.

Note A System Testing Agent for C must be installed and running on each target host before deploying virtual testers to those targets.

Following the execution architecture and constraints needed to comply, the System Testing Script Compiler provides several ways to generate the virtual testers.

Options

Virtual testers can take the following command line options:

```
-INSTANCE=<instance>
```

If the **.pts** test script contains **DECLARE_INSTANCE** instructions, this option specifies which behavioral instance the virtual tester is to initiate. By default, the virtual tester generates all behaviors contained in the test script, but on execution, only one instance is adopted.

If no instances are selected even though instances do exist in the test script, the virtual tester stops with a fatal error message.

```
-RIO=<trace_file>
```

This syntax specifies the name of the execution trace file to be generated by the virtual tester.

If you do not define a trace filename, the name *<virtual tester>.rio* will be used.

```
-OCCID=<occurrence_id_number>
```

This allows you to specify the occurrence identification number to use in the virtual tester identifier when using communication between virtual testers. See the **INTERSEND** and **INTERRECV** statements for more information.

System Testing Load Report Generator

Purpose

The Load Report Generator produces a report describing messages and execution time.

Syntax

```
atsload -SEPARATOR='<sep_string>' [-TITLE] <rdm file> {[, <rdm file> ]}
```

where:

- <rdm file> is an **.rdm** output file generated by the Report Generator.
- <sep_string> is the separator string.
- <options> is a set of optional parameters among those described below.

Description

The System Testing Load Report Generator tool processes **.rdm** file of a virtual tester from the Report Generator and produces the following output:

- **TITLE:** The optional header for each column of the report.
- **SCENARIO:** total execution time
- **SEND:** timestamp of a **SEND** message relative to the beginning of the **SCENARIO**
- **MESSAGE:** timestamp of a **WAITTIL** relative to the beginning of the **SCENARIO**
- **PRINT:** value of the numeric parameter

You can use the Load Report Generator to compare between several virtual testers. Data is presented in columns, separated by a separator string. Each

column represents a particular virtual tester.

There must be one **.rdd** file for each virtual tester.

Optional Parameters

-TITLE

This option adds a **TITLE** line to the report, containing the name of the virtual tester for each column.

Example

```
atsload -SEPARATOR=':' vt1.rod vt2.rod vt2.rod
```

Return Codes

After execution, the program exits with the following return codes:

Code	Description
0	End of execution with no errors
7	End of execution due to a fatal error
9	End of execution due to an internal error

All messages are sent to the standard error output device.

Component Testing Script Languages

3

The Component Testing feature of Test RealTime provides a unique, fully automated, and proven solution for C, C++, Java and Ada Component Testing, dramatically increasing test productivity.

This section provides reference information to help you write your own test scripts using Rational Test RealTime.

Each test feature uses its own specialized language. Use the scripting language corresponding to the type of test.

C Test Script Language

Purpose

The Test RealTime Component Testing feature for C uses its own simple language for test scripting: C Test Script Language.

This section describes each keyword of the C Test Script language, including:

- Syntax
- Functionality and rules governing its usage
- Examples of use

Notation Conventions

Throughout this guide, command notation and argument parameters use the following standard convention:

Notation	Example	Meaning
BOLD	BEGIN	Language keyword
<i><italic></i>	<i><filename></i>	Symbolic variables
[]	[<i><option></i>]	Optional items
{ }	{ <i><filenames></i> }	Series of values
{ { }	{ { <i><filenames></i> }	Optional series of variables
	on off	OR operator

Note: C Test Script Language keywords can be entered in upper or lower case. To distinguish C Test Script Language from native code, this guide conventionally uses upper case typography for C Test Script Language keywords.

C Test Script Language Identifiers

A C Test Script Language identifier is a text string used as a label, such as the name of a **TEST** or a **STUB** in a **.ptu** test script.

Identifiers are made of an unlimited sequence of the following characters:

- a-z
- A-Z
- 0-9
- _ (underscore)

Spaces are not valid identifier characters.

Note that identifiers starting with a numeric character are allowed. The following statement, for example, is syntactically correct:

```
TEST 1
...
END TEST
```

Unlike keywords, C Test Script Language identifiers are case sensitive. This means that **LABEL**, **label**, and **Label** are three different identifiers.

C Test Script Language Keywords

The C Test Script Language keywords are not case sensitive. This means that *STUB*, *stub*, and *Stub* are equivalent. For conventional purposes however, this document uses upper-case notation for the C Test Script Language keywords in order to differentiate from native source code.

Block Keywords

- ELEMENT...END ELEMENT
- ENVIRONMENT...END ENVIRONMENT
- INITIALIZATION...END INITIALIZATION
- SERVICE...END SERVICE
- SIMUL...ELSE_SIMUL...END SIMUL
- TERMINATION...END TERMINATION
- TEST...END TEST

Other Keywords

- BEGIN
- COMMENT
- DEFINE STUB
- EV

- FAMILY
- FORMAT
- HEADER
- IF...ELSE...END IF
- INIT
- INCLUDE
- NEXT_TEST
- STUB
- USE
- VAR, ARRAY and STR
 - <initialization> Parameter
 - <expected> Parameter
 - <variable> Parameter

BEGIN

C Test Script Language

Purpose

The **BEGIN** instruction marks the beginning of the test program.

Syntax

```
BEGIN
```

Description

BEGIN marks the beginning of the C code generation.

The **BEGIN** instruction is mandatory and must be located before any other Component Testing instruction for C, except a **HEADER** instruction.

If the **BEGIN** keyword is not found, a warning message is generated and a **BEGIN** instruction is implicitly created before the first occurrence of a **SERVICE** instruction.

COMMENT

C Test Script Language

Purpose

The **COMMENT** instruction adds a textual comment to the test report.

Syntax

```
COMMENT [<text>]
```

Argument

<text> is an optional text string to be displayed.

Description

The **COMMENT** instruction is optional and can be used anywhere in the test script.

The position of the **COMMENT** instruction in the test script defines the position in which the comments appear in the test report.

- Within an **ELEMENT** block: the comment appears just before the variable state descriptions.
- After a **TEST** instruction: the comment appears in the test header, before the variable descriptions.
- After a **SERVICE** instruction: the comment appears in the service header, before the test descriptions.
- Outside a **SERVICE** block: the comment appears in the service header following the declaration, before the test descriptions.

Example

```
TEST 1
FAMILY nominal
COMMENT histogram computation for a black image
ELEMENT
```

DEFINE STUB

Purpose

The **DEFINE STUB** and **END DEFINE** instructions delimit a simulation block consisting of stub definition functions, methods, or procedure declarations.

Syntax

```
DEFINE STUB <stub_name> [ <stub_dim> ]  
END DEFINE
```

<stub_name> is the mandatory name of a simulation block.

<stub_dim> is an optional maximum number of stub call descriptions for a test scenario. By default, its value is 10.

Description

Defining stubs in a test script is optional.

Using the stub definitions, the C Test Script Compiler generates simulation variables and functions for which the interface is identical to that of the stubbed variables and functions.

The purpose of these simulation variables and functions is to store and test input parameters, assign values to output parameters, and if necessary, return appropriate values.

Definitions of functions must be in the form of ANSI prototypes for C.

Stub parameters describe both the type of item used by the calling function and the mode of passing. The mode of passing the parameter is specified by

adding the following before the parameter name:

- **_in** for input parameters
- **_out** for output parameters
- **_inout** for input/output parameters
- **_no** for parameters that you do not want to test

The parameter mode is optional. If no parameter mode is specified, the **_in** mode is assumed by default.

A return parameter is always deemed to be an output parameter.

Global variables defined in **DEFINE STUB** blocks replace the real global variables.

DEFINE STUB / **END DEFINE** blocks must be located after the **BEGIN** instruction and outside any **SERVICE** block.

Example

```
BEGIN
DEFINE STUB fic
  #int open_file(char _in f[100]);
  #int create_file(char _in f[100]);
  #int read_file(int _in fd, char _out l[100]);
  #int write_file(int fd, char _in l[100]);
  #int close_file(int fd);
END DEFINE
```

ELEMENT...END ELEMENT

Purpose

The **ELEMENT** and **END ELEMENT** instructions delimit a test phase or **ELEMENT** block.

Syntax

```
ELEMENT
END ELEMENT
```

Description

The **ELEMENT** instruction is mandatory and can only be located within a **TEST** block. If absent, a warning message is generated and the **ELEMENT** block is implicitly declared before the first occurrence of a **VAR**, **ARRAY**, **STR**, or **STUB** instruction.

The block must end with the instruction **END ELEMENT**. If absent, a warning message is generated and it is implicitly declared before the next **ELEMENT** instruction, or the **END TEST** instruction.

The **ELEMENT** block contains a call to the service under test as well as instructions describing the initializations and checks on test variables.

Positioning of **VAR**, **ARRAY**, **STR** or **STUB** instructions is irrelevant, with respect to the test procedure call since the Test Compiler separates these instructions into two parts:

- The test initializer (described by **INIT**) is generated with the **ELEMENT** instruction.
- The test of the expected value (described by **EV**) is generated with the **END ELEMENT** instruction.

Example

```
TEST 1
FAMILY nominal
ELEMENT
VAR x1, init = 0, ev = init
VAR x2, init = SIZE_IMAGE-1, ev = init
VAR y1, init = 0, ev = init
VAR y2, init = SIZE_IMAGE-1, ev = init
ARRAY image, init = 0, ev = init
VAR histo[0], init = 0, ev = SIZE_IMAGE*SIZE_IMAGE
ARRAY histo[1..SIZE_HISTO-1], init = 0, ev = 0
VAR status, init ==, ev = 0
#status = compute_histo(x1,y1,x2,y2,histo);
END ELEMENT
END TEST
```

ENVIRONMENT ... END ENVIRONMENT

C Test Script Language

Purpose

The **ENVIRONMENT** instruction defines a test environment declaration, that is, a default set of test specifications.

Syntax

```
ENVIRONMENT <name> [ ( <param> { , <param> } ) ]  
END ENVIRONMENT
```

<name> is a mandatory identifier that provides a unique environment name.

<param> is an optional identifier.

Description

Every environment can contain parameters. The declared parameters can be used in initialization and expected value expressions. These parameters are initiated by the **USE** instruction.

The **END ENVIRONMENT** instruction marks the end of an environment declaration.

<name> specifies an environment name that is referenced in the **USE** instruction.

An environment must be defined after the **BEGIN** instruction.

Each environment is visible in the block in which it has been declared and in any blocks included in this block, after its declaration.

An environment can only contain **VAR**, **ARRAY**, **STR**, **FORMAT** or **STUB** instructions and conditional generation instructions. If it is empty, a warning message is generated.

An environment is activated by the **USE** instruction that defines its scope and its priority. After generating the initializations and the tests of an **ELEMENT** block, visible environments are included in order of priority, at every **END ELEMENT** instruction, in order to complete the initializations and tests.

Example

```
ENVIRONMENT compute_histo
  VAR x1, init = 0, ev = init
  VAR x2, init = SIZE_IMAGE-1, ev = init
  VAR y1, init = 0, ev = init
  VAR y2, init = SIZE_IMAGE-1, ev = init
  ARRAY histo, init = 0, ev = 0
  VAR status, init ==, ev = 0
END ENVIRONMENT
```

FAMILY

C Test Script Language

Purpose

The **FAMILY** instruction groups tests by families or classes.

Syntax

```
FAMILY <family_name> { , <family_name> }
```

Argument

<family_name> is a mandatory identifier indicating the name of the test family. Typically, you could specify nominal, structural, or robustness families.

Description

The **FAMILY** instruction appears within **TEST** blocks, where it defines the families to which the test belongs.

When you run the test sequence, you can request that only tests of a given *family* are executed.

A test can belong to several families. In this case, the **FAMILY** instruction contains a <family_name> list, separated by commas.

The **FAMILY** instruction must be located before the first **ELEMENT** block of the **TEST** block and must be unique in the **TEST** block.

The **FAMILY** instruction is optional. If it is omitted, a warning message is generated and the test belongs to every family.

Example

```
TEST 1
FAMILY nominal
COMMENT histogram computation on a black image
ELEMENT
```

FORMAT

C Test Script Language

Syntax

```
FORMAT <variable> = <format>
```

```
FORMAT <type> = <format>
```

```
FORMAT <field> = <format>
```

Description

The **FORMAT** instruction allows you to change the display format of a tested item. This item can be:

- A variable
- A simple C type declared by typedef; in this case, the display mode will change for all variables of this type
- A member of a structure or a C union; in this case, the display mode will change for all the members of variables of this type

The new format of the item is described by an optional abstract C type eventually followed by a printing directive for variables of type integer or floating only. The format of these printing directives is the following:

directive [size]

with the following possibilities for integers:

- **#h** for hexadecimal display,
- **#b** for binary display,
- **#u** for unsigned decimal display,

- **#d** for signed decimal display,

and with the following possibilities for floating variables:

- **#f** to display without an exponent,
- **#e** to display with an exponent.

For integers, *size* is the number of bits to be displayed. For floating variables, *size* is the number of the number of digits after the decimal point.

Associated Rules

The **FORMAT** instruction is optional and must be located after the **BEGIN** instruction.

It is applicable immediately, only in the block in which it is declared.

<variable> follows standard C syntax rules. *<type>* is a C identifier used in *typedef*, *struct* or *union* instructions. *<format>* is an abstract C type.

If the change is to be applied to array elements, you can use an abstract C type to describe the new modified variable, field, or type.

<variable>, *<type>* or *<field>* and *<format>* are mandatory. A format cannot be empty. It must contain either the abstract C type or the printing directive.

In the printing directive, the *size* is optional. The *size* must be a multiple of 8 for the integers. The default values for this *size* are the following:

- For integers, the number of bits of the abstract type if it is given, or if it is not, the number of bits of the type or the variable whose printing format is modified
- For **#f**, 6 digits after the decimal point and for **#e**, 2 digits after the decimal point

Example

```
#char x;
#define char *pointer;
#pointer p;
#char t[10]
FORMAT x = int
FORMAT pointer = void-- modifies p
FORMAT t = int[] - t is an array of integers
FORMAT x = int#h8 -- display in hexa, only 8 bits
FORMAT y = #b -- display in binary without
-- modifying the type
FORMAT z = short#u -- display in unsigned decimal
FORMAT f1 = #f -- displays by example 3.670000
FORMAT f1 = #f4 -- displays by example 3.6700
FORMAT f1 = #e4 -- displays by example 0.36700E1FORMAT
```

HEADER

C Test Script Language

Purpose

The **HEADER** instruction specifies the name and version of the module under test as well as the version number of the test script.

Syntax

```
HEADER <module_name>, <module_version>, <test_plan_version>
```

<module_name>, *<module_version>* and *<test_plan_version>* are character strings with no restrictions, except for versions beginning with a dollar sign (**\$**). These instructions must be followed by an identifier.

Description

This information contained in the **HEADER** keyword is reproduced in the test report header to identify the test sequence.

The module and test script versions can be read from the environment variables if they are identifiers beginning with a dollar sign (**\$**).

The **HEADER** instruction is mandatory, but its arguments are optional. It must be the first instruction in the test program. If it is absent, a warning message is generated.

Example

```
HEADER histo, 01a, 01a
BEGIN
```

IF ... ELSE ... END IF

C Test Script Language

Syntax

```
IF <condition> { , <condition> }  
ELSE  
END IF
```

Description

The **IF**, **ELSE** and **END IF** statements allow conditional generation of the test program.

These statements enclose portions of script that are included depending on the presence of one of the conditions in the list provided to the C Test Compiler by the **-define** option.

The *<condition>* list forms a series of conditions that is equivalent to using an expression of logical **ORs**.

The **IF** instruction starts the conditional generation block.

The **END IF** instruction terminates this block.

The **ELSE** instruction separates the condition block into 2 parts, one being included when the other is not.

Associated Rules

This block of instructions can appear anywhere in the test program.

<condition> is any identifier. You must have at least one condition in an **IF** instruction.

This block can contain any scripting or native language.

IF and **END IF** instructions must appear simultaneously.

The **ELSE** instruction is optional.

The generating rules are as follows:

- If at least one of the conditions specified in the **IF** instruction's list of conditions appears in the list associated with the `-define` option, the first part of the block is included.
- If none of the conditions specified in the **IF** instruction appears in the list associated with the `-define` option, then the second part of the block is included (if **ELSE** is present).

The **IF...ELSE...END IF** block is equivalent to the following block in C:

```
#if defined(<condition>) { || defined(<condition>) } ...  
...  
#else  
...  
#endif
```

Example

```
IF test_on_target  
  VAR register, init == , ev = 0  
ELSE  
  VAR register, init = 0 , ev = 0  
END IF
```

INCLUDE

C Test Script Language

Syntax

```
INCLUDE CODE <file>
INCLUDE PTU <file>
```

Description

The **INCLUDE** specifies an external file for the C Test Compiler to process.

When an **INCLUDE** instruction is encountered, the C Test Compiler leaves the current file, and starts pre-processing the specified file. When this is done, the C Test Compiler returns to the current file at the point where it left.

Including a file with the additional keyword **CODE** lets you include a source file without having to start every line with a hash character ('#').

Including a file with the additional keyword **PTU** lets you include a test script within a test script.

Associated Rules

The name of the included file can be specified with an absolute path or a path relative to the current directory.

If the file is not found in the current directory, all directories specified by the **-incl** option are searched when the preprocessor is started.

If it is still not found or if access is denied, an error is generated.

The instruction **INCLUDE CODE** *<file>* is equivalent to the following line in C

```
##include "<file>".
```

Example

```
INCLUDE CODE file1.c  
INCLUDE CODE ../file2.c  
INCLUDE PTU /usr/foo/test/file3.ptu
```

INITIALIZATION ... END INITIALIZATION

C Test Script Language

Syntax

```
INITIALIZATION  
END INITIALIZATION
```

Description

The **INITIALIZATION** and **END INITIALIZATION** statements let you provide native code that is integrated into the generation as the first native instructions of the test program (first lines of main).

In some environments, such as if you are using a different target machine, this provides a way to initialize the target.

Associated Rules

An **INITIALIZATION** block must appear after the **BEGIN** instruction or between two **SERVICE** blocks.

This block can only contain native code. This code must begin with '#' or '@'.

There is no limit to the number of **INITIALIZATION** blocks. On generation, they are concatenated in the order in which they appeared in the test script.

NEXT_TEST

C Test Script Language

Syntax

```
NEXT_TEST [ LOOP <nb> ]
```

where:

- <nb> is an integer expression strictly greater than 1.

Description

The **NEXT_TEST** instruction starts a **TEST** block linked to the previous one. It contains one more **ELEMENT** block. It does not contain the **FAMILY** instruction.

For this new test, a number of iterations can be specified by the keyword **LOOP**.

The **NEXT_TEST** instructions can only appear in a **TEST ... END TEST** block.

The main difference between a **NEXT_TEST** block and an **ELEMENT** block is when you use an **INIT IN** statement within a test block:

- If the **INIT IN** is in a **TEST** block, there will be a loop over the entire **TEST** block, without consideration of the **ELEMENT** blocks that it might contain.
- If the **INIT IN** is inside a **NEXT_TEST** block however, the loop will not affect the **ELEMENT** blocks within other **TEST** blocks

Example

```
SERVICE COMPUTE_HISTO
```

```
# int x1, x2, y1, y2 ;
# int status ;
# T HISTO histo;
TEST 1
  FAMILY nominal
  ELEMENT
...
END ELEMENT
NEXT TEST LOOP 2
ELEMENT
```

SERVICE ... END SERVICE

C Test Script Language

Syntax

```
SERVICE <service_name>
END SERVICE
```

Description

The **SERVICE** instruction starts a **SERVICE** block. This block contains the description of all the tests relating to a given service of the module to be tested.

The <service_name> parameter flags the tested service in the test report, and is therefore usually the name of this service (although this is not obligatory).

The **END SERVICE** instruction indicates the end of the service block.

Associated Rules

The **SERVICE** instruction must appear after the **BEGIN** instruction.

The <service_name> parameter can be any identifier. It is obligatory.

Example

```
BEGIN
SERVICE COMPUTE_HISTO
# int x1, x2, y1, y2 ;
# int status ;
# T_HISTO histo;
TEST 1
FAMILY nominal
```

SIMUL ... ELSE_SIMUL ... END SIMUL

C Test Script Language

Purpose

The **SIMUL**, **ELSE_SIMUL**, and **END SIMUL** instructions allow conditional generation of test script program.

Syntax

```
SIMUL  
ELSE_SIMUL  
END SIMUL
```

Description

Code enclosed within a **SIMUL** block is conditionally generated depending on the status of the **Simulation** setting in Test RealTime.

The **SIMUL** instruction starts the conditional generation block.

The **END SIMUL** instruction terminates this block.

The **ELSE_SIMUL** instruction separates this block into two parts, one being included when the other is not, and vice versa.

This block of instructions can appear anywhere in the test program and can contain both scripting instructions or native code.

The **SIMUL** and **END SIMUL** instructions must appear as a pair. One cannot be used without the other.

The **ELSE_SIMUL** instruction is optional.

When using Test RealTime in the command line interface, use the **-nosimulation** option to deactivate the simulation setting in the C Test Script Compiler.

When using the Test RealTime user interface, select or clear the **Simulation** option in the **Component Testing for C** tab of the **Configuration Settings** dialog box.

The generating rules are as follows:

- If **Simulation** is enabled => the first part of the **SIMUL** block is included.
- If **Simulation** is disabled => the second part of the block (**ELSE_SIMUL**) is included if it exists. If there is no **ELSE_SIMUL** statement, then the **SIMUL** block is ignored.

Example

```
SIMUL
  #x = 0;
ELSE_SIMUL
  #x = (type_x *) malloc ( sizeof(*x) );
END SIMUL
...
SIMUL
VAR x , INIT = 0 , EV = 1
VAR p , INIT = NIL , EV = NONIL
ELSE_SIMUL
VAR x , INIT = 0 , EV = 0
VAR p , INIT = NIL , EV = NIL
END SIMUL
```

STUB

C Test Script Language

Purpose

The STUB instruction for C describes all calls to a simulated function in a test script.

Syntax

```
STUB [<stub name>.] <function> [<call range> =>] ([<param val> {,  
<param val> }]) [<return_val>] {, [<call_range> =>] ([<param_val>  
{, <param_val> }]) [<return_val>] }
```

Description

The following is described for every parameter of this function and for every expected call:

- For **_in** parameters, the values passed to the function; these values will be stored and then tested during execution,
- For **_out** parameters and, where appropriate, the return value, the values returned by the function; these values will be stored in order to be returned during execution,
- For **_inout** parameters, both the previous two values are required,
- For **_no** parameters, nothing is required.

<call_range> describes one or many calls as follows:

```
<call_num> =>  
<call_num> .. <call_num> =>  
others =>
```

where *<call_num>* is the number of the stub call (begins at 1). The keyword **others** describes all calls that have not been described. Moreover, this key

word allows to not check the sub call number.

<function> is the name of the simulated function. It is obligatory. You must previously have described this function in a **DEFINE STUB ... END DEFINE STUB** block. You can specify in which stub (*<stub_name>*) the declaration was made.

<param_val> is an expression describing the test values for *_in* parameters and the returned values for *_out* parameters. For **_inout** parameters, *<param_val>* is expressed in the following way:

```
(<in_param_val>, <out_param_val>)
```

<return_val> is an expression describing the value returned by the function if its type is not void. Otherwise, no value is provided.

You must give values for every **_in**, **_out** and **_inout** parameter; otherwise, a warning message is generated. The **_no** parameters are ignored.

<param_val> and *<return_val>* are expressions that can contain:

- Numeric (integer or floating-point), character, or character string literal values. Strings can be delimited by single or double inverted commas
- Constants which can be numeric, characters, or character strings
- Constants defined in the test script
- Variables belonging to the test program or the module to be tested
- C functions
- The keyword **NIL** to designate a null pointer
- Pseudo-variables **I**, **I1**, **I2** ..., **J**, **J1**, **J2** ..., where **I_n** is the current index of the *m*th dimension of the parameter and **J_m** the current number of the subtest generated by the test scenario's *m*th **INIT IN**, **INIT FROM** or **LOOP**; the **I** and **I1** variables are therefore equivalent as are **J** and **J1**; the

subtest numbers begin at **1** and are incremented by **1** at each iteration

- An expression with one or more of the above elements combined using any of the C operators (+, -, *, /, %, &, |, ^, &&, ||, <<, >>) and casting, with all required levels of parentheses, and conforming to C rules of syntax and semantics, the + operator being allowed to concatenate character string variables
- For arrays and structures, a list of expressions between braces ('{' and '}') or brackets ('[' and ']') with, where appropriate:
 - For an array element, part of an array or a structure field, its index, interval or name followed by '=' and by the value of the array element, common to all elements of the array portion or structure field
 - The keyword **OTHERS** followed by '=' and the default value of any array elements or structure fields not yet mentioned.

You must describe at least one call in the **STUB** instruction. There can be several descriptions, separated by commas (','). **STUB** instructions can appear in **ELEMENT** or **ENVIRONMENT** blocks.

Type Modifier '@' Syntax

In a STUB definition you can use a @ before a type modifier to indicate that this type modifier should be used when generating variable that test the correct execution of STUBs. For example:

```
DEFINE STUB Example
#void ConstParam (@const int _in *a);
END DEFINE
```

Without the @ symbol, the variables are of *const int* type and therefore are not modified by the test harness.

Example

```
STUB open_file ("file1")3
STUB create_file ("file2")4
STUB read_file (3,"line 1")1, (3,"line 2")1, (3,"")0
STUB write_file (4,"line 1")1, (4,"line 2")1
STUB close_file 1=>(3)1,2=>(4)1
```

TERMINATION ... END TERMINATION

C Test Script Language

Syntax

```
TERMINATION  
END TERMINATION
```

Description

The **TERMINATION** and **END TERMINATION** instructions delimit a block of native code that is integrated into the generation process as the last instructions to be executed (last lines of main).

In certain environments (for example, a different target machine), these instructions terminate execution on the target machine.

Associated Rules

A **TERMINATION/END TERMINATION** block must appear after the **BEGIN** instruction and outside any **SERVICE** block.

This block can only contain native code. This code must begin with '#' or '@'.

There is no limit to the number of **TERMINATION** blocks. They are concatenated at generation.

TEST ... END TEST

Syntax

```
TEST <test_name> [ LOOP <nb>]  
END TEST
```

Description

The **TEST** instruction starts a **TEST** block. This block describes the test case for a service. It contains one more **ELEMENT** blocks specifying the test.

In the test report, the *<test_name>* parameter flags the test within the **SERVICE** block. Tests are usually given numbers in ascending order.

A number of iterations can be specified for each test with the optional **LOOP** keyword.

The **TEST LOOP** statement can generate graph metric results in a **.rtx** file. To do this, you must set the environment variable **ATURTX** to *True*. The produced **.rtx** graph can be viewed in the Test RealTime Graphic Viewer.

The **END TEST** instruction marks the end of the **TEST** block.

Associated Rules

The **TEST** and **END TEST** instructions can only appear in a **SERVICE** block.

<test_name> is obligatory. If it is absent, the Test Compiler generates an error message.

<nb> is an integer expression strictly greater than 1.

Example

```
SERVICE COMPUTE_HISTO
# int x1, x2, y1, y2 ;
# int status ;
# T_HISTO histo;
TEST 1
FAMILY nominal
ELEMENT
```

USE

C Test Script Language

Purpose

The **USE** instruction activates a test environment that is defined using the **ENVIRONMENT** instruction.

Syntax

```
USE <name> [ ( <expression> { , <expression> } ) ]
```

Description

The position of the **USE** instruction determines which tests are affected by the environment used:

- If **USE** occurs outside a **SERVICE** block, the instructions contained in this environment are applied to all subsequent **ELEMENT** blocks.
- If **USE** occurs within a **SERVICE** block and outside a **TEST** block, the instructions contained in this environment are applied to all subsequent **ELEMENT** blocks of this **SERVICE** block.
- If **USE** occurs within a **TEST** block and outside an **ELEMENT** block, the instructions contained in this environment are applied to all subsequent **ELEMENT** blocks of this **TEST** block.
- If **USE** occurs within an **ELEMENT** block, the instructions contained in this environment will only be applied to this block.

Because the **USE** instruction can appear at these four different levels, four priority levels are created from "outside a **SERVICE** block" (the lowest priority) to "inside an **ELEMENT** block" (the highest priority).

Within the same priority level, the last **USE** instruction is the one with the highest priority.

Testing is completed according to these priority rules, and on the basis that variables tested several times are included in the environment with the highest priority.

This is also true for every element of arrays described in extended mode.

If the environment it references takes parameters, the **USE** instruction must initialize these parameters using C expressions.

Associated Rules

The **USE** instruction can appear after **BEGIN** and outside an **ENVIRONMENT** block, after the definition of the environment it references.

<name> is the name of an environment declared by the **ENVIRONMENT** instruction.

<expression> must be an expression that conforms to C syntax and semantics.

Example

```
ENVIRONMENT compute_histo
  VAR x1, init = 0, ev = init
  VAR x2, init = SIZE_IMAGE-1, ev = init
  VAR y1, init = 0, ev = init
  VAR y2, init = SIZE_IMAGE-1, ev = init
  ARRAY histo, init = 0, ev = 0
  VAR status, init ==, ev = 0
END ENVIRONMENT
USE compute_histo
```

VAR, ARRAY and STR

Purpose

The **VAR**, **ARRAY**, and **STR** instructions declare the test of a simple variable, a variable array or a variable structure.

Syntax

```
VAR <variable>, <initialization>, <expected_value>
ARRAY <variable>, <initialization>, <expected_value>
STR <variable>, <initialization>, <expected_value>
VAR <expression>, <expected_value>
ARRAY <expression>, <expected_value>
STR <expression>, <expected_value>
```

where:

- *<variable>* is a C variable
- *<expression>* is a valid expression
- *<initialization>* is a Component Testing initialization parameter for C
- *<expected value>* is a Component Testing expected_value parameter for C

Description

Use the **VAR**, **ARRAY**, and **STR** instructions to declare a variable test. During test execution, if the value of the variable is out of the bounds specified in the *<expected_value>* expression, the test is Failed.

Use the **VAR**, **ARRAY**, and **STR** instructions to declare a variable test. During test execution, if the value of the variable is out of the bounds specified in the *<expected>* expression, the test is Failed.

VAR, **ARRAY** or **STR** are synonymous and do not change the way in which the result displayed in the test report.

- **VAR**: This statement is for simple variables.
- **ARRAY**: This statement is for variable arrays.
- **STR**: This statement is for variable structures.

If you use a **VAR** statement to test an array or structure, the report lists each element of the array or structure.

The **VAR**, **ARRAY**, and **STR** instructions must be located in an **ELEMENT** or an **ENVIRONMENT** block.

If you use a **VAR** statement to test an array or structure, the report lists each element of the array or structure.

The **VAR**, **ARRAY**, and **STR** instructions must be located in an **ELEMENT** or an **ENVIRONMENT** block.

VAR, ARRAY and STR <variable> Parameter

C Test Script Language

Description

In the current documentation, the Component Testing <variable> parameter for C is a conventional notation name for a C variable under test. The syntax of the <variable> parameter allows you to specify the upper and lower boundaries of the range of the test for each dimension of the array:

```
[ <lower> .. <upper> ]
```

where:

- <lower> is lower boundary for acceptable values of <variable>
- <upper> is the upper boundary for acceptable values of <variable>

Associated Rules

<variable> can be a simple variable (integer, floating-point number, character, pointer or character string), an element of an array or structure, part of an array, an entire array, or a complete structure.

If no test boundaries have been specified for a variable array, all array elements are tested. Similarly, if one of the fields of a variable structure is an array, all elements of this field are tested.

The variable must have been declared in advance.

Example

```
VAR x, ...
VAR y[4], ...
VAR z.field, ...
VAR p->value, ...
```

```
ARRAY y[0..100], ...  
ARRAY y, ...  
STR z, ...  
STR *p, ...
```

VAR, ARRAY and STR <initialization> Parameter

C Test Script Language

Purpose

In this documentation, the Component Testing <initialization> parameters for C specify the initial value of the variable.

Syntax

```
INIT = <exp>
INIT IN { <exp>, <exp>, ... }
INIT ( <variable> ) WITH { <exp>, <exp>, ... }
INIT FROM <exp> TO <exp> [STEP <exp> / NB TIMES <nb> / NB RANDOM
<nb>[+ BOUNDS]]
INIT ==
```

where:

- <exp> is an expression as described below.
- <nb> is an integer constant that is either literal or derived from an expression containing native constants or constants defined with a **CONST** instruction
- <variable> is a C variable

Description

The <initialization> expressions are used to assign an initial value to a variable. The initial value is displayed in the Component Testing report for C.

Initializations can be expressed in the following ways:

- **INIT** = *<exp>* initializes a variable before the test with the value *<expression>*.
- **INIT IN** { *<exp>* , *<exp>* , ...} declares a list of initial values. This is a condensed form of writing that enables several tests to be contained within a single instruction.
- **INIT (<variable>) WITH** { *<exp>* , *<exp>* , ...} declares a list of initial values that is assigned in correlation with those of the variable initialized by an **INIT IN** instruction. There must be the same number of initial values.

The **INIT IN** and **INIT (<variable>) WITH** expressions cannot be used with for arrays that were initialized in extended mode or for structures.

- **INIT FROM** *<lower>* **TO** *<upper>* allows the initial value of a numeric variable (integer or floating-point) to vary between lower and upper boundary limits:
 - **STEP**: the value varies by successive steps
 - **NB_TIMES** *<nb>*: the value varies by a number *<nb>* of values that are equidistant between the two boundaries, where *<nb>* ≥ 2
 - **NB_RANDOM** *<nb>*: the value varies by generating random values between the 2 boundaries, including, when appropriate, the boundaries, where *<nb>* ≥ 1

The **INIT FROM** expression can only be used for numeric variables.

- **INIT ==** allows the variable to be left un-initialized. You can thus control the values of variables that are dynamically created by the service under test. The initial value is displayed in the test report as a question mark (?).

An initialization expression can still be used (**INIT == <expression>**) to

include of expected value expression when using the **INIT** pseudo-variable is used. See Expected_Value Expressions.

Expressions

The initialization expressions *<exp>* can be among any of the following values:

- Numeric (integer or floating-point), character, or character string literal values. Strings can be delimited by single or double quotes
- Native constants, which can be numeric, characters, or character strings
- Constants, previously defined with the **CONST** instruction
- Variables belonging to the test program or the module to be tested
- C or Ada functions
- The keyword **NIL** to designate a null pointer
- Pseudo-variables **I**, **I1**, **I2** ..., **J**, **J1**, **J2** ..., where **I_n** is the current index of the *n*th dimension of the parameter and **J_m** the current number of the subtest generated by the test scenario's *m*th **INIT IN**, **INIT FROM** or **LOOP**; the **I** and **I1** variables are therefore equivalent as are **J** and **J1**; the subtest numbers begin at **1** and are incremented by **1** at each iteration
- A C or Ada expression with one or more of the above elements combined using any operators and casting, with all required levels of parentheses, the + operator being allowed to concatenate character string variables
- For arrays and structures, any of the above-mentioned expressions between braces ('**{}**') for C or brackets ('**[]**') for Ada, including when appropriate:
 - For an array element, part of an array or a structure field, its index, interval or name followed by '='>' and by the value of the array element, common to all elements of the array portion or

structure field

- For structures you can test some fields only, by using the following syntax:
- For C: {<value>,,<value>}
- For all languages: [<fieldname> => <value>, <fieldname> => <value>]
- The keyword **OTHERS** followed by '=' and the default value of any array elements or structure fields not yet mentioned
- For **INIT IN** and **INIT WITH** only, a list of values delimited by braces ('}') for C composed of any of the previously defined expressions

Additional Rules

Any integers contained in an expression must be written either in accordance with native lexical rules, or under the form:

- *<hex_integer>***H** for hexadecimal values. In this case, the integer must be preceded by **0** if it begins with a letter
- *<binary_integer>***B** for binary values

Example

```
VAR x, INIT = pi/4-1, ...
VAR y[4], INIT IN { 0, 1, 2, 3 }, ...
VAR y[5], INIT(y[4]) WITH { 10, 11, 12, 13 }, ...
VAR z.field, INIT FROM 0 TO 100 NB_RANDOM 3, ...
VAR p->value, INIT ==, ...
ARRAY y[0..100], INIT = sin(I), ...
ARRAY y, INIT = {50=>10,others=>0}, ...
STR z, INIT = {0, "", NIL}, ...
STR *p, INIT = {value=>4.9, valid=>1}, ...
```

VAR, ARRAY and STR <expected> Parameter

C Test Script Language

Purpose

In this documentation, the Component Testing <expected value> parameters for C specify the expected value of a variable.

Syntax

```
EV = <exp>
EV = <exp> , DELTA = <delta>
MIN = <exp>, MAX = <exp>
EV IN { <exp>, <exp>, ... }
EV ( <variable> ) IN { <exp>, <exp>, ... }
EV ==
```

where:

- <exp> can be any of the expressions of the Initialization Parameters, plus the following expressions:
- <delta> is the acceptable tolerance of the expected value and can be expressed:
- <variable> is a C variable

Description

The <expected value> expressions are used to specify a test criteria by comparison with the value of a variable. The test is considered Passed when the actual value matches the <expected value> expression.

An acceptable tolerance <delta> can be expressed:

- As an absolute value, by a numerical expression in the form described above
- As a percentage of the expected value. Tolerance is then written in the form <exp>%.

Expected values can be expressed in the following ways:

- **EV** = <exp> specifies the expected value of the variable when it is known in advance. The value of variable is considered correct if it is equal to <exp>.
- **EV** = <exp>, **DELTA** = <tolerance> allows a tolerance for the expected value. The value of variable is considered correct if it lies between <exp> - <tolerance> and <exp> + <tolerance>.
- **MIN** = <exp> and **MAX** = <exp> specify an interval delimited by an upper and lower limit. The value of the variable is considered correct if it lies between the two expressions. Characters and character strings are treated in dictionary order.
- **EV IN** { <exp>, <exp>, ... } specifies the values expected successively, in accordance with the initial values, for a variable that is declared in **INIT IN**. It is therefore essential that the two lists have an identical number of values.
- **EV (<variable>) IN** is identical to **EV IN**, but the expected values are a function of another variable that has previously been declared in **INIT IN**. As for **EV IN**, the two lists must have an identical number of values.
- **EV ==** allows the value of <variable> not to be checked at the end of the test. Instead, this value is read and displayed. The value of <variable> is always considered correct.

Expressions

The initialization expressions *<exp>* can be among any of the following values:

- Numeric (integer or floating-point), character, or character string literal values. Strings can be delimited by single or double quotes
- Native constants, which can be numeric, characters, or character strings
- Constants, previously defined with the **CONST** instruction
- Variables belonging to the test program or the module to be tested
- C or Ada functions
- The keyword **NIL** to designate a null pointer
- The keyword **NONIL**, which tests if a pointer is non-null
- Pseudo-variables **I**, **I1**, **I2** ..., **J**, **J1**, **J2** ..., where **I_n** is the current index of the *n*th dimension of the parameter and **J_m** the current number of the subtest generated by the test scenario's *m*th **INIT IN**, **INIT FROM** or **LOOP**; the **I** and **I1** variables are therefore equivalent as are **J** and **J1**; the subtest numbers begin at **1** and are incremented by **1** at each iteration
- A C or Ada expression with one or more of the above elements combined using any operators and casting, with all required levels of parentheses, the + operator being allowed to concatenate character string variables
- For arrays and structures, any of the above-mentioned expressions between braces ({}) for C, including when appropriate:
 - For an array element, part of an array or a structure field, its index, interval or name followed by '=>' and by the value of the array element, common to all elements of the array portion or structure field

- For structures you can test some fields only, by using the following syntax:
- {<value>,,<value>}
- The keyword **OTHERS** followed by '=' and the default value of any array elements or structure fields not yet mentioned
- The pseudo-variable **INIT**, which copies the initialization expression

Additional Rules

EV with **DELTA** is only allowed for numeric variables.

MIN = <exp> and **MAX** = <exp> are only allowed for alphanumeric variables that use lexicographical order for characters and character strings.

MIN = <exp> and **MAX** = <exp> are not allowed for pointers.

Only **EV** = and **EV** == are allowed for structured variables.

In some cases, in order to avoid generated code compilation warnings, the word **CAST** must be inserted before the **NIL** or **NONIL** keywords.

Example

```

VAR x, ..., EV = pi/4-1
VAR y[4], ..., EV IN { 0, 1, 2, 3 }
VAR y[5], ..., EV(y[4]) IN { 10, 11, 12, 13 }
VAR z.field, ..., MIN = 0, MAX = 100
VAR p->value, ..., EV ==
ARRAY y[0..100], ..., EV = cos(I)
ARRAY y, ..., EV = {50=>10,others=>0}
STR z, ..., EV = {0, "", NIL}
STR *p, ..., EV = {value=>4.9, valid=>1}

```

C++ Test Script Language

Purpose

The Test RealTime Component Testing feature for C++ uses its own simple language for test scripting: C++ Test Script Language.

This language is designed to bring object-oriented programming techniques to software testing and is applicable to both object-oriented and procedural source code.

- File extensions and C++ Test Script Language usage
- C++ Test Script Language Keywords
- C++ Test Script Language Identifiers
- Mixing C++ Test Script Language and native code
- Inheritance
- Template classes
- Target Deployment Port options

Notation conventions

Throughout this guide, command notation and argument parameters use the following standard convention:

Notation	Example	Meaning
BOLD	INCLUDE	Language keyword
<i><italic></i>	<i><filename></i>	Symbolic variables
[]	[<option>]	Optional items
{ }	{<filenames>}	Series of values

[*]	{{<filenames>}}	Optional series of variables
	on off	OR operator

Note: C++ Test Script Language keywords can be entered in upper or lower case. To distinguish C++ Test Script Language from native code, this guide conventionally uses upper case typography for C++ Test Script Language keywords.

C++ Test Script Language Identifiers

A C++ Test Script Language identifier is a text string used as a label, such as the name of a test case in this kind of script.

Identifiers are made of an unlimited sequence of the following characters:

- a-z
- A-Z
- 0-9
- _ (underscore)

Spaces are not valid identifier characters.

Note that, as opposed to the C++ language, identifiers starting with a numeric character are allowed. For example, **PROC 1** is syntactically correct.

Unlike keywords, these identifiers are case sensitive. This means that **LABEL**, **label**, and **Label** are three different identifiers. C++ Test Driver Scripts

A Component Testing for C++ Test Driver Script (**.otd** script) describes a test

driver. Its purpose is to stimulate the tested classes by creating objects and calling their methods. It provides different ways to check that the objects behavior is the one that was expected.

When executed, the script is translated into a C++ source by Component Testing for C++. Furthermore, it instruments the source code under test whenever the STUB, CHECK STUB, or CHECK METHOD statements are used.

<C++ Test Driver script item> may be one of the following entities:

- INCLUDE
- STUB
- TEST CLASS
- TEST SUITE
- ON ERROR
- RUN
- Native statement

Order is meaningful for INCLUDE and native statements. RUN may appear only once in a C++ Test Driver script. Other entities are not ordered: for instance, a TEST CLASS can forward-reference a STUB.

Note A C++ Test Driver script is made both of statements and instructions. Instructions are ordered: their relative position is meaningful. Statements have no order: they have a declarative nature.

Example

```
INCLUDE "test_a.otl";
ON ERROR BYPASS;
#include "vectors.h"
TEST CLASS B : A {
    TEST CASE clear {
```

```
                                #while (!vect.empty ())
vect[vect.last()].delete;
                                CHECK (vect.count() == 0);
                                }
                                RUN {
                                clear;
                                }
                                }
RUN { B; }
```

Native Code

Syntax

```
# <single-line C++ code>
C++# <single-line C++ code>
#{ <multiple-line C++ code> }#
C++#{ <multiple-line C++ code> }#
```

Location

C++ Test Driver Script, PROLOGUE, EPILOGUE, TEST CASE, PROC, STUB, CHECK EXCEPTION

Description

<single-line C++ code> and <multiple-line C++ code> are made of one or several C++ statements. They must conform to the syntax expected by the host compiler and must be relevant to the current context.

Macros may be used, but it is recommended to define them only at the root-level of the C++ Test Driver Script.

Native code is copied as is in the generated test driver source.

Only global declarations are allowed inside the C++ Test Driver Script.

Inside a PROLOGUE statement, the declaration's scope is that of the surrounding structure (TEST CLASS or TEST SUITE). Elsewhere, the scope is local (visible from the declaration to the end of the C++ Test Script Language block).

the sequence `#}` is different from `}#`:

- }# ends a multiple-line native code block started by #{.
- #} is a single-line native code made with the character "closing brace."

Warning: The use of **return**, **goto**, or any other jump instruction is not allowed in native code. If jump instructions are used, unexpected results will occur.

Native code may generate an error when it raises an unexpected exception. Use the CHECK EXCEPTION statement to specify exceptions.

Example

```

#include <myclass.h>
#{
    static int counter;
    extern void initialize (MyClass &);
    static const int MAX=200;
}#
TEST CLASS A {
    ON ERROR EXIT;
    PROLOGUE {
        #MyClass mc;
        #initialize (mc);
        #for (counter=0; counter < MAX; counter++) {
    }
    TEST CASE 1 {
        #void *temp;
        #temp = mc.create ();
        #mc.unref (temp);
        CHECK mc.empty ();
    }
    TEST CASE 2 {
        #{
            void *temp[MAX];
            for (int i = 0; i<counter; i++)
            {
                temp[i] = mc.create ();
            }
            for (int i = 0; i<counter; i++)
            {
                mc.unref (temp[i]);
            }
            CHECK mc.empty ();
        }#
    }
}

```



```

        EPILOGUE {
            #} //end of for
        }
    }
    RUN {
        A.1;
        A.2;
    }

```

In this example, a loop is defined around the components of the test class A. The loop starts in **PROLOGUE**, and ends in **EPILOGUE**. The execution of test class A will run nothing, because there is no **RUN** statement in this test class. However, the two test cases may be run separately, as it is shown in the above example.

The execution sequence is: *A.PROLOGUE*, *A.1* (200 times), *A.EPILOGUE*, *A.PROLOGUE*, *A.2* (200 times), *A.EPILOGUE*.

Error Handling

An error may be generated by either native code or any of the following instructions in a test script:

- CHECK
- CHECK PROPERTY
- CHECK EXCEPTION
- CHECK STUB
- CHECK METHOD
- REQUIRE
- ENSURE
- Native statement

Refer to each of these keywords to see when the instructions generate an error.

Error handling behavior is specified with the keyword `ON ERROR`. According to the choice specified by `ON ERROR`, the script may continue normal execution, skip the current block, or exit.

Test Results

When no errors occur during execution of a C++ Test Script Language script, the script receives *Passed* status. Otherwise, it is considered *Failed*.

When the test is completed, the errors appear in the Report Viewer or in the UML/SD Viewer as red notes.

C++ Contract Check Scripts

A C++ Contract Check Script (**.otc** script) describes assertions for a set of classes. Each C++ class can be associated to a Contract Check CLASS block. When executed, the script instruments the source code under test.

The scripts, being descriptions, are made of statements only. As a consequence, the order of execution is irrelevant.

`<class>` is a Contract Check CLASS block, describing assertions for a C++ class.

Note The evaluation of the contract should not have any side effects. The contract evaluation does not alter the state of the corresponding system. For more specific information refer to REQUIRE, ENSURE, INVARIANT and STATE sections.

Example

```
CLASS StringArray {  
    INVARIANT (count() >= 0);  
    WRAP add (String s)
```

```

        REQUIRE (s.length() > 0)
        ENSURE (count () == _ATO_old.count () + 1)
    }

    CLASS VCR {
        STATE Empty {
            (media_present() == false)
        }

        STATE Loaded {
            (media_present() == true)
            (mode () == m_stop;)
        }

        STATE Playing {
            (media_present() == true)
            (mode() == m_play || mode() == m_pause)
        }

        TRANSITION Empty TO Loaded;
        TRANSITION Loaded TO Playing;
        TRANSITION Playing TO Loaded;
        TRANSITION Playing TO Empty;
        TRANSITION Loaded TO Empty;
    }
}

```

Template Classes

Component Testing for C++ supports assertions only for fully generic and fully specialized template classes. Partial specializations are not supported.

A contract referring to a generic template class is applied to every instance of this template class, unless a specific contract has been defined for an instance of this template class.

There may be a state machine description associated with the template class, and another with a template specialization. In such a case, the latter applies to the specific template instance, and the first applies to any other instance.

Same mechanism for invariant definition (There may be invariants associated with the template class, and other invariants with a template specialization. In such a case, the latter ones apply to the specific template instance, while the first one apply to any other instance.)

A wrap defined within a generic template class contract does not apply to specialization of the associated method. If you want to test a method specialization, you must define a WRAP into the contract associated to the class instance the method specialization belongs to.

It is not possible to define WRAPs for template methods within a non-template class.

Specialization

Specialized templates are templates for which some of the parameters are real. Full-specialization of a template is an instance of the template (all parameters are real).

Example

```
template <class T,int N> class C; // generic template, not
a specialization
template <class T> class C<T,2>; // partial specialization
(not supported by Component Testing for C++)
template <> class C<char *,2>; // full-specialization
```

Note When using full-specializations, latest ISO/IEC C++ standards suggest using the template prefix **template<>**.

Inheritance

Contracts are divided into several semantic parts:

- state machine
- list of invariants
- pre-conditions
- post-conditions

Each of these parts can be inherited in separate ways in derived classes, unless a matching part has been found in the derived class.

If you specify invariants for a class, they override the invariants defined for any base class. Similarly, a state machine description for a class overrides any state machine definition inherited from a base class.

If a class inherits from several base classes for which a class contract is defined, but does not define any invariant, the base class's invariants are merged. Similarly, if no state transition is defined, a state-transition is maintained for every sub-object inheriting a tested base class.

If you want to define a contract for a class, but not all of its base classes are associated to a contract, then you should use invariants and state transitions with care, because the methods inherited from the non-tested classes are not instrumented. In this situation, define a contract, even empty, for every base class of the class you want to test. A warning is generated during the instrumentation if such a case is encountered.

C++ Test Script Keywords

C++ Test Script Language keywords are not case sensitive. This means that STUB, stub, and Stub are equivalent. For conventional purposes however, this document uses upper-case notation for C++ Test Script Language keywords in order to differentiate from native source code.

OTD Test Driver Definition Keywords

Structure-related Keywords

- EPILOGUE
- INCLUDE
- TEST CLASS
- TEST SUITE

- TEST CASE
- PROLOGUE
- RUN
- PROPERTY
- PROC
- REQUIRE
- ENSURE

Verification Keywords

- CHECK
- CHECK EXCEPTION
- CHECK METHOD
- CHECK PROPERTY
- CHECK STUB

Error-handling Keyword

- ON ERROR

Stubbing Keyword

- STUB
- REQUIRE
- ENSURE

Instructions

- COMMENT
- PRINT
- CALL

OTC Contract Check Keywords

- CLASS and SINGLE CLASS
- WRAP
- REQUIRE
- ENSURE
- INVARIANT
- STATE
- TRANSITION ... TO

PROLOGUE

Syntax

```
PROLOGUE { <prologue item>* }
```

Location

TEST CLASS | TEST SUITE

Description

The PROLOGUE statement defines native code that is to be executed whenever the surrounding test class execution begins. This code is executed before any other of the test class' components.

The PROLOGUE statement may appear at most once in a test class. In an object-context, a prologue can be compared to a constructor.

<*prologue item*> may be one of the following entities:

- COMMENT
- PRINT
- Native statement

Order is meaningful. The native code can be made of declarations and instructions. Variables declared in prologue are visible from every component of the surrounding test class.

Note If the native code raises an exception, the prologue generates an error, handled by the ON ERROR local block. Even if the ON ERROR statement is CONTINUE, the whole TEST CLASS or TEST SUITE is skipped, including its EPILOGUE.

Example

```
TEST CLASS ATest
{
    PROLOGUE {
        #Stack s(20);
        #s.fill ();
    }
    TEST CASE tc1 {
        CHECK (!s.full ());
    }
    RUN {
        tc1;
    }
}
```

EPILOGUE

Syntax

```
EPILOGUE { <epilogue item>* }
```

Location

TEST CLASS | TEST SUITE

Description

The EPILOGUE structure defines native code that is to be executed whenever the execution of the surrounding test class ends. This code is executed after other test class components.

An EPILOGUE statement may appear at most once in a test class. In an object-context, an EPILOGUE can be compared to a destructor.

<epilogue item> may be one of the following entities:

- COMMENT
- PRINT
- Native statement

Order is meaningful.

Example

```
TEST CLASS ATest
{
    PROLOGUE {
        #Stack *s = new Stack(20);
        #s->fill ();
    }
    EPILOGUE {
        #delete s;
    }
}
```

```
    }  
    TEST CASE tc1 {  
        CHECK (!s->full ());  
    }  
    RUN {  
        tc1;  
    }  
}
```

PROC

Syntax

```
PROC <procedure name> [ ( <formal parameter> [ ( , <formal  
parameter> ) ] ) ]  
[ REQUIRE ( <native expression> ) ]  
{ <procedure item>}  
[ ENSURE ( <native expression> ) ]
```

Location

TEST CLASS, TEST SUITE

Description

The PROC statement defines a procedure.

Note Procedures can be called with the CALL statement.

<procedure name> is a C++ Test Script Language identifier. It is visible in the surrounding test class or test suite, in sub-test classes or sub-test suites, and in inheriting test classes.

<formal parameter> is a C++ Test Script Language identifier. It has no type: it is replaced into the procedure by an actual parameter. Thus it can refer to a C++ type as well a C++ constant or a C++ variable.

<native expression> is a C++ expression that can be evaluated to a Boolean. The REQUIRE expression is evaluated before execution of the procedure. The ENSURE expression is evaluated after execution of the procedure. If any of these optional expressions is *False*, the evaluation leads to an error in the caller's context.

<procedure item> may be one the following entities:

- CHECK EXCEPTION
- CHECK
- CHECK PROPERTY
- CALL
- CHECK STUB
- CHECK METHOD
- ON ERROR
- COMMENT
- PRINT
- Native statement

Order is meaningful, except for CHECK STUB, CHECK METHOD, and ON ERROR statements.

The ON ERROR statement may only appear once.

Example

```
TEST CLASS TestA {
    PROC InitArray (array, length)
        REQUIRE (length>30 && length<array.length())
        {
            #{
                for (int i = 0; i<length; i++)
                    array[i].init ();
            }#
        }
        ENSURE (array[0].initialized())
    }
}
```

RUN

Syntax

```
RUN { <run item>* }
```

Location

TEST CLASS, OTD Script

Description

The RUN statement defines the behavior of the surrounding test class.

<run item> may be one of the following entities:

- Test class name
- Test suite name
- Test case name

These names refer to a component defined in the surrounding test class or in an inherited test class. Order is meaningful. They can refer to a nested item (the nesting sequence is specified with the list of identifiers, from the most-surrounding to the most-nested one, separated by a dot).

The RUN statement can be located either within a **TEST CLASS** or at the root level of a C++ Test Driver Script:

- When used in a TEST CLASS, the RUN statement defines the behavior of the surrounding TEST CLASS.
- When used at the root level of a script, the RUN statement defines which entities are to be run when the script is executed. The RUN items can refer to any entity of the script.

Only one RUN statement is allowed at the root of a script or within each TEST CLASS.

The RUN statement is not allowed in included scripts.

RUN items are executed sequentially.

Example

```
TEST CLASS ATest
{
  TEST CASE tc1 {
    #s.push (i);
    CHECK (!s.full ());
  }
  TEST CASE tc2 {
    #s.pop ();
    CHECK (!s.empty ());
  }
  RUN {
    tc1; tc2; tc2; tc1;
  }
}
```

TEST CLASS

Syntax

```
TEST CLASS <test class name> [ ( <formal parameter> [ , <formal parameter> ] ) ] [ : <parent class> ] { <test class item> }
```

Location

C++ Text Driver Script, TEST CLASS

Description

The TEST CLASS statement describes an object test class, which is one of the structuring entities of a C++ Test Driver Script. Test classes can appear at the root-level of a C++ Test Driver Script and in test classes.

<test class name> is a C++ Test Script Language identifier.

<formal parameter> is a C++ Test Script Language identifier. It has no type: it is replaced into the test class by an actual parameter. Thus it can refer to a C++ type as well a C++ constant or a C++ variable.

<parent class> is a valid test class that is defined in the same scope that contains the TEST CLASS. All entities of a parent class are inherited. This mean that they are available just as if they were defined in **<test class name>** itself. The entities defined in the current test class with the same name as in the parent class are said to override, or replace, the entities defined in the parent class.

<test class item> may be one of the following entities:

- TEST CLASS
- TEST SUITE

- TEST CASE
- ON ERROR
- PROPERTY
- PROC
- PROLOGUE
- EPILOGUE
- RUN

A test class scope has no order, so these entities can appear in any order. However ON ERROR, EPILOGUE, PROLOGUE, and RUN may appear only once. The execution of a TEST CLASS without a RUN statement will execute the class' PROLOGUE and EPILOGUE only.

Example

```

TEST CLASS AdvancedTest (T) : BasicTest
{
    PROLOGUE {
        #Stack s (20);
    }
    PROPERTY Initial { (s.count == 0) }
    PROPERTY Final { (s.count == 1) }
    TEST CASE tc1 {
        CHECK PROPERTY Initial;
        #s.push (1);
    }
    CHECK PROPERTY Final;
}
RUN {
    tc1;
}

```

TEST CASE

Syntax

```
TEST CASE <test case name> { <test case item> }
```

Location

TEST CLASS, TEST SUITE

Description

The TEST CASE statement describes an object test case, which is the smallest testing structure in a hierarchical C++ Test Driver Script. Test cases appear in test classes and test suites.

<*test case name*> is a C++ Test Script Language identifier.

<*test case item*> may be one of the following entities:

- ON ERROR
- CHECK EXCEPTION
- CHECK
- CHECK PROPERTY
- CHECK METHOD
- CHECK STUB
- CALL
- COMMENT
- PRINT
- Native statement

CALL, CHECK, CHECK PROPERTY, COMMENT, PRINT as well as Native statements are ordered (they are executed sequentially). Other entities are not (they have a global effect on the test case).

ON ERROR and CHECK EXCEPTION may appear only once.

Example

```
TEST CLASS A {  
    TEST CASE 1 {  
        CHECK (x == 1);  
        #do_something ();  
        CHECK PROPERTY ok;  
    }  
    RUN {  
        1;  
    }  
}
```

TEST SUITE

Syntax

```
TEST SUITE <test suite name> { <test suite item> }
```

Location

OTD script, TEST CLASS, TEST SUITE

Description

The TEST SUITE statement describes an Object test suite, which is one of the structuring entities of an C++ Test Driver Script. Test suites can appear at the root-level of a C++ Test Driver Script, in test classes, and in test suites.

<*test suite name*> is a C++ Test Script Language identifier.

<*test suite item*> may be one of the following entities:

- TEST SUITE
- TEST CASE
- ON ERROR
- PROPERTY
- PROC
- PROLOGUE
- EPILOGUE

All entities but TEST CASE are not ordered in a test suite scope. However, ON ERROR, EPILOGUE, and PROLOGUE may appear only once. The test cases and test suites of a test suite are executed sequentially.

Example

```
TEST SUITE ChargeTest {
  TEST CASE Test1
  {
    /... */
  }
  TEST SUITE Test2
  {
    TEST CASE SubTest2a
    {
      /... */
    }
    TEST CASE SubTest2b
    {
      /... */
    }
  }
}
```

INCLUDE

Syntax

```
INCLUDE "<file name>";
```

Location

C++ Test Driver Script

Description

The INCLUDE statement opens the file *<file name>* and inserts its code into the C++ Test Driver Script.

A file cannot include itself, directly or indirectly.

An included file must not have a RUN statement at the script level. A RUN statement at script level is only allowed in the main test script.

Example

```
INCLUDE "test1.otd";
```

PROPERTY

Syntax

```
PROPERTY <property name> [ ( <parameter> [ (, <parameter> ) ] ) ]  
{ ( (<native expression> ) ) }
```

Location

TEST CLASS, TEST SUITE

Description

The PROPERTY statement associates a global state, defined by the conjunction of *<native expression>*, to a name. This name is visible in the TEST CLASS where the property is defined.

Note: The occurrence of a property may be checked with the keyword CHECK PROPERTY.

<native expression> is a valid C++ expression that may be evaluated to a Boolean.

Example

```
TEST CLASS TestA {  
    PROPERTY Empty { ( s.count() == 0 ) }  
    TEST CASE tc1 {  
        CHECK PROPERTY Empty;  
    }  
    RUN { tc1; }  
}
```

STUB

Syntax

```
STUB <stub name> : <native routine signature>
[ REQUIRE ( <require native expression> ) ]
{<stub item>... <stub item>}
[ ENSURE ( <ensure native expression> ) ]
```

Location

C++ Test Driver Script

Description

The **STUB** statement defines a stub for a function or method. A stub defines or replaces the initial routine.

Note The use of stubs requires instrumentation.

<stub name> is a unique C++ Test Script Language identifier.

<native routine signature> is a C++ signature matching the routine to stub. Unlike WRAP signatures, the signature must be complete; the return type and parameters (type and name) must be specified. If the routine is a class member or belongs to a namespace, its name must be qualified. If the routine is a template function or a template class member, the usual `template<...>` prefix must be used. If it is a generic template, any instance of this template is stubbed. If it is a template specialization, only the corresponding instance is stubbed.

<require native expression> is a C++ expression that can be evaluated to a Boolean. It is evaluated before the stub execution. It can refer to:

- The global variables defined in the test script.
- The stubbed routine's parameters.

<ensure native expression> is a C++ expression which can be evaluated to a Boolean. It is evaluated after the stub execution. It can refer to:

- The global variables defined in the test script.
- The stubbed routine's parameters.
- The **_ATO_result** variable that contains the routine return value, if any. Its type is that of the routine return type. Its value may be undefined if no value is returned (because an exception was thrown, or a return without a value is executed, or the function implicitly returns).
- The **_ATO_in_exception** Boolean variable, which is *True* if the post-condition is executed because an exception has been thrown. This variable is available only if the Target Deployment Package is configured to support exceptions.

If one of these expressions is *False*, the stub is failed but not the **CHECK STUB**, which could still have been defined to ensure the stub is called.

<stub item> may be one the following entities:

- CHECK
- COMMENT
- PRINT
- Native statement

The "..." zone is optional and is replaced by the code provided through the CHECK STUB statement. If not specified, it is implicitly defined at the end of the **STUB** block.

You cannot define several stubs for the same method. However you can

define a stub for each instance of a template function or a template class member.

If a statement of the **STUB** generates an error, the stub is declared failed, but its execution continues (there is always an implicit **ON ERROR CONTINUE** in stubs).

An error in a **STUB** does not imply an error in the **TEST CASE** containing the corresponding **CHECK STUB**. The **CHECK STUB** statement only checks that the stub is called, not that its execution is correct.

Example

```
STUB ModifyCell : int IntArray::Modify (int Cell)
  REQUIRE (Cell != 128)
  {
    #int Nb = random(10000);
    ... // this part is completed by the code of CHECK
  STUB
    #return (Nb);
  }
```

In this example, a number *Nb* is randomly chosen. If no additional code is provided by a **CHECK STUB**, then this number is returned. If a **CHECK STUB** is provided, assign the expected return value to *Nb* on a case-by-case basis.

REQUIRE

Syntax

```
REQUIRE <native expression>
```

Location

WRAP, STUB, PROC

Description

The REQUIRE statement describes a method pre-condition. It can be used in a WRAP, STUB or PROC block.

Note The information below pertains to the use of REQUIRE within a WRAP block. For more information about using the REQUIRE and ENSURE statement within a STUB or PROC block, please refer to the STUB and PROC.

<native expression> is a C++ Boolean expression (or an expression that can be converted to a Boolean).

The following symbols can be used in <native expression>:

- Any of the public or protected class members.
- The method parameters (with the names used in the signature or in the method definition).
- Any of the global variables declared in the file where the method is defined.
- The following symbols cannot be used in <native expression>:
 - Local variables
 - Macros

Evaluation

<*native expression*> is evaluated before any code of the method is executed (local variables are not pushed yet).

Warning: you can call methods in <*native expression*>, but you must ensure that these calls do not modify the object's state by writing to any field. You can ensure this by calling *const* methods only.

Example

C++ source code example:

```
class Stack {
    int count;
    Stack () : count(0) {}
    void push (void *);
    void *pop ();
};
```

OTC code example:

```
CLASS Stack {
    WRAP pop
    REQUIRE (count > 0)
}
```

ENSURE

Syntax

```
ENSURE <native expression>
```

Location

WRAP, STUB, PROC

Description

The ENSURE statement describes a method post-condition. It can be used in a WRAP, STUB or PROC block.

Note The information below pertains to the use of ENSURE within a WRAP block. For more information about using the REQUIRE and ENSURE statement within a STUB or PROC block, please refer to the STUB and PROC.

<native expression> is a C++ Boolean expression (or an expression that can be converted to a Boolean).

The following symbols can be used in <native expression>:

- Any of the public or protected class members.
- The method parameters (with the names used in the signature or in the definition).
- Any of the global variables declared in the file where the method is defined.
- The `_ATO_result`, `_ATO_old` and `_ATO_in_exception` variables (see below).

The following symbols cannot be accessed in *<native expression>*:

- Local variables
- Macros

Variables

`_ATO_result`

This variable contains the method return value, if any. Its type is that of the method return type. Its value may be undefined if no value is returned (because an exception was thrown, or a return without a value is executed, or the function implicitly returns). **`_ATO_result`** is not available when the option **`--postcondition_before_return`** is activated (see Target Deployment Port options).

`_ATO_old`

This variable contains a copy of the object as it was before the method call. The **`_ATO_old`** object is generated by the class copy constructor. If the class copy constructor is explicitly defined, you should remember that **`_ATO_old`** is not a *raw copy* of the current object, but a copy as defined by the copy constructor. **`_ATO_old`** is not available in constructors.

For performance purposes, the **`_ATO_old`** variable is generated only if it is used in the ENSURE expression.

`_ATO_in_exception`

This is a Boolean variable that is TRUE if the post-condition is executed because an exception has been thrown. This variable is available only if the Target Deployment Package is configured to support exceptions.

Evaluation

When **`--postcondition_before_return`** option is set in the Target

Deployment Package **.opp** file, *<native expression>* is evaluated before the return expression. If the return expression evaluation causes side-effects, they are not taken into account at the time the post-condition is checked. This option is provided for compatibility with limited C++ compilers, and its use should be avoided as much as possible.

Otherwise, *<native expression>* is evaluated after any code of the method (local variables are already popped).

Warning: you can call methods in *<native expression>*, but you must make sure that these calls do not modify the object's state (that is, they do not write to any field). You can ensure this by calling *const* methods only. If you want the compiler to check this, use the **ATO_AC_STRICT_CHECKING** Target Deployment Port option.

Example

C++ source code example:

```
class Stack {
    public:
        int count;
        Stack () : count(0) {}
        void push (void *);
        void *pop ();
};
```

C++ Contract Check Script code example:

```
CLASS Stack {
    WRAP push
    ENSURE (count == _ATO_old.count + 1)
}
```

CHECK

Purpose

The CHECK instruction evaluates the Boolean value of a native expression.

Syntax

```
CHECK [ <comment> ] ( <native expression> );
```

Arguments

<comment> is an optional string that appears in the test results.

<native expression> is a valid C++ expression, which may be converted into a Boolean.

Location

TEST CASE, PROC, STUB, CHECK STUB

Description

The CHECK instruction evaluates the native expression. If the result of the check is TRUE, the result of the corresponding test is Passed. Otherwise, an error is generated. The result of the error handling is specified with the ON ERROR keyword.

Example

```
TEST CLASS TestA {
    TEST CASE tc1 {
        CHECK (s.empty ());
    }
    RUN { tc1; }
}
```


CHECK EXCEPTION

Purpose

The CHECK EXCEPTION statement checks that an exception is raised within a block.

Syntax

```
CHECK EXCEPTION ( <native type> [ <native parameter name> ] ) ( ( {  
<on exception item>} ) / ; )
```

Arguments

<native type> is the C++ type of the expected exception.

<native parameter name> is the optional name of the exception. It may also be used in *<on exception item>*.

<on exception item> may be a **COMMENT**, a **PRINT** or a native-code statement.

Description

The **CHECK EXCEPTION** statement specifies that the exception of type *<native type>* is expected to be raised in the current C++ Text Script Language block (test case or proc). If this exception is not raised in the block, an error is generated.

Only one **CHECK EXCEPTION** may occur per block. A **CHECK EXCEPTION** can be located in a **TEST CASE** or **PROC** block.

Example

```
TEST CASE TC1 {  
    CHECK EXCEPTION (DivideByZeroException) {
```

```
        PRINT "Ok";  
    }  
    #b = 1; c = 0;  
    #a = b / c;  
}
```

CHECK METHOD

Syntax

```
CHECK METHOD <native routine signature>;
```

Location

TEST CASE

Description

The **CHECK METHOD** checks that a routine (function or class member) is called during the execution of the surrounding test case. If the routine is not called, an error is generated. Error-handling behavior is specified with the **ON ERROR** keyword.

Note The use of **CHECK METHOD** requires instrumentation of the source code under test.

<native routine signature> refers to an existing routine.

- If it is a class or namespace member, its name must be qualified but the return type may be omitted.
- If it is a class member, and if it not overloaded, the parameters may be omitted.

If parameters are specified, their names may be omitted.

Only one **CHECK METHOD** referring to each routine may occur in each **TEST CASE**.

Example

```
TEST SUITE A {  
    TEST CASE 1 {
```

```

        CHECK METHOD IntArray::ModifyCell (int);
        #IntArray ia;
        #InitializeArray (ia); // this function
calls IntArray::ModifyCell
        // ia is filled with random numbers
    }
    TEST CASE 2 {
        CHECK METHOD IntArray::ModifyCell; // you
can omit parameters
        #IntArray ia;
        // IntArray::ModifyCell was not called =>
error
    }
}
RUN { A; }

```

CHECK PROPERTY

Syntax

```
CHECK PROPERTY [ "<comment>" ] <property name> [ ( <actual  
parameter> [ ( , <actual parameter> )* ) ]
```

Location

TEST CASE, PROC, STUB, CHECK STUB

Description

The CHECK PROPERTY instruction evaluates the property *<property name>*. If the result is TRUE, the test is passed. Otherwise, it generates an error. Error handling behavior is specified with the ON ERROR keyword.

<comment> is an optional string that appears in test results.

<property name> is a valid property defined in the current test class, in a nesting test class or in an inherited test class.

Note: properties are defined with the keyword [PROPERTY](#).

Example

```
TEST CLASS TestA {  
    PROPERTY Empty { ( s.count() == 0 ) }  
    TEST CASE tc1 {  
        CHECK PROPERTY Empty;  
    }  
    RUN { tc1; }  
}
```

CHECK STUB

Syntax

```
CHECK STUB <stub name> { <stub item>}  
CHECK STUB <stub name>;
```

Location

TEST CASE

Description

The CHECK STUB instruction checks that a stub is called at least once during the TEST CASE execution.

If a block is provided, it specifies that *<stub item>* should be executed instead of the stub's "..." zone. If no block is provided, the execution of the stub's "..." does nothing.

If the stub is not called, an error is generated. Error handling behavior is specified with the ON ERROR keyword.

Note: The use of stubs requires instrumentation of the source code under test.

<stub name> is a valid stub identifier.

<stub item> may be one the following entities:

- CHECK
- COMMENT
- PRINT

- Native statement

Only one CHECK STUB may refer to the same STUB in a TEST CASE.

Note The CHECK STUB statement may be used before the corresponding STUB is defined.

Example

```

STUB ModifyCell : int IntArray::Modify (int Cell)
REQUIRE (Cell != 128)
{
    #int Nb = random(10000);
    ... // this part is completed by the code of CHECK
STUB
    this.array[Cell] = Nb;
    #return (Nb);
}
TEST SUITE A {
    TEST CASE 1 {
        CHECK STUB ModifyCell;
        #IntArray ia;
        #InitializeArray (ia); // this function calls
IntArray::ModifyCell
        // ia is filled with random numbers
    }
    TEST CASE 2 {
        CHECK STUB ModifyCell {
            #Nb = 0;
        }
        #IntArray ia;
        #InitializeArray (ia); // this function calls
IntArray::ModifyCell
        // ia is filled with 0
    }
}
RUN { A; }

```

ON ERROR

Syntax

```
ON ERROR [ { <error item> } ] <error action>;
```

Location

C++ Test Driver Script, TEST CLASS, TEST SUITE, TEST CASE, PROC

Description

The ON ERROR statement defines the behavior of the test driver when an error occurs.

ON ERROR applies to the current scope level, and to nested scopes, unless another ON ERROR statement has been defined. The general rule is that the most nested ON ERROR statement is applied.

Note An error can be raised by any instruction of a TEST CASE or a PROC, and by native code from a PROLOGUE or EPILOGUE.

ON ERROR does not apply to stubs. There is always an implicit ON ERROR CONTINUE behavior in stubs

<error item> may be one of the following entities:

- COMMENT
- PRINT
- Native statement

This block is executed when an error occurs.

<error action> is a keyword which defines the behavior of the test driver

when an error occurs:

- **CONTINUE** : The execution continues just as if no error occurred. If the error comes from an unexpected exception raised by native code, the execution continues after the native code, except for an error in a PROLOGUE block. Since it is the default behavior, this on-error action should only be specified to override another on-error action.
- **EXIT** : The execution of the test driver stops at the error point.
- **BYPASS** : The execution of the rest of the current test case or procedure is skipped.
- **BYPASS** *<test class name>* / *<test suite name>* / *<test case name>* / *<proc name>* : The execution of the rest of the referred entity is skipped.

Example

```
ON ERROR CONTINUE;
TEST CLASS A {
    ON ERROR EXIT;
    TEST SUITE A1 {
        ON ERROR BYPASS A;
        TEST CASE A1a {
            ON ERROR CONTINUE;
            CHECK (false); // this leads to an
error but execution continues
            PRINT "ok"; // this instruction is
executed
        }
        TEST CASE A1b {
            CHECK (false); // this leads to an
error
// execution resumes
            PRINT "ko"; // this instruction is
never executed
        }
    }
    TEST CASE A2 {
        CHECK (false); // this leads to an error --
the test driver exits
        PRINT "ko"; // this instruction is never
executed
    }
}
RUN { A1; A2; }
```

```

}
TEST CLASS B {
    TEST CASE B1 {
        ON ERROR BYPASS;
        CHECK (false); // this leads to an error --
execution resumes after B1
        PRINT "ko"; // this instruction is never
executed
    }
    TEST CASE B2 {
        CHECK (false); // this leads to an error but
execution continues
        PRINT "ok"; // this instruction is executed
    }
    RUN { B1; B2; B1; }
}
RUN { B; A; A.A2; }

```

In this example, the execution is: B1 (aborted), B2, B1 (aborted), A1a, A1b (A is aborted), A2 (exited).

PRINT

Syntax

```
PRINT (<expression> [ ( , <expression>* ) ] );
```

Location

C++ Test Driver Script, TEST CASE, PROC, STUB, CHECK STUB, PROLOGUE, EPILOGUE, ON ERROR, CHECK EXCEPTION

Description

The PRINT instruction outputs dynamic comments to the traces file. These comments can be visualized through the UML/SD viewer in the GUI.

<expression> is a valid C++ expression whose type must be handled by the Target Deployment Port (handled types are scalar types, floating types, strings and pointers). <expression> is evaluated on execution.

A PRINT instruction may generate an error when the evaluation of one of the arguments raises an unexpected exception. Use the CHECK EXCEPTION statement to specify exceptions.

Example

```
TEST CASE tc1 {
    #s.push (i);
    PRINT ("Adding an element", i);
    CHECK (!s.full ());
}
```

COMMENT

Syntax

```
COMMENT <one-line text>
COMMENT { <multiple-line text> }
```

Location

TEST CASE, PROC, STUB, CHECK STUB, PROLOGUE, EPILOGUE, ON
ERROR, CHECK EXCEPTION

Description

The **COMMENT** instruction allows the output of static comments to a trace file. These comments can be visualized through the UML/SD Viewer in the GUI.

Example

```
TEST CASE tcl {
#s.push (i);
    COMMENT An element was added to the stack.
CHECK (!s.full ());
}
```

CALL

Purpose

The CALL instruction calls a procedure defined with a PROC statement.

Syntax

```
CALL <procedure_name> [ ( <actual_parameter> [ ( , <actual  
parameter> ) ] ) ] ;
```

Arguments

<procedure_name> is a valid procedure name, defined within a test class or test suite, or in an inherited test class.

<actual_parameter> is an optional list of parameters that must conform to the expected procedure parameter list.

Description

A CALL instruction can be located within a TEST CASE or PROC block.

Example

```
TEST CLASS TestA {  
    PROC InitArray (array, length)  
    {  
        #{  
            for (int i = 0; i<length; i++)  
                array[i].init ();  
        }#  
    }  
    TEST CASE tc1 {  
        #Array<int> ia (50);  
        CALL InitArray (ia, 49);  
    }  
}
```

CLASS and SINGLE CLASS

Syntax

```
[SINGLE] CLASS <native class signature> { <class assertions>* }
```

Location

C++ Contract Check Script

Description

The CLASS statement introduces a block describing assertions for a specific C++ class. This block is called a *class contract*. Assertions described in a CLASS statement also apply to derived classes, unless the SINGLE keyword is used.

Use SINGLE CLASS to describe assertions only for the <native class signature> class.

<native class signature> is a qualified C++ class name. It can refer to template classes.

For instances of template classes, the signature must follow the pattern:

template<> *class_name*<*actual_parameters*>

Note: The template<> sequence may be omitted in order to comply with deprecated usage, but it is best to specify it.

For template classes with generic parameters, the signature must follow the pattern:

template<*formal_parameters*> *class_name*

Note: The `template<...>` sequence may be omitted, but in that case it is not possible to use the formal parameters in the nested WRAP signatures).

`<class assertions>` can be one of the following:

- WRAP
- INVARIANT
- STATE
- TRANSITION

Examples

C++ source code example:

```
class A {
    class B {
        // ...
    };
    // ...
};
template<class T,int N> class C
{
    // ...
};
```

C++ Contract Check Script example:

```
CLASS A
{
    INVARIANT (/*...*/);
    // ...
}
CLASS A::B
{
    // ...
}
CLASS template<class T,int N> C
{
    // ...
}
CLASS template<> C<char*,255>
{
    // ...
}
```

}
}

WRAP

Syntax

```
WRAP <native method signature> <WRAP assertions>
```

Location

CLASS

Description

The WRAP statement describes pre- and post-conditions for a method.

<native method signature> refers to an existing method within the class. The return type may be omitted. The parameters names may be omitted if the WRAP assertions do not refer to the parameters. The parameters list may be omitted if the method is not overloaded and if the WRAP assertions do not refer to the parameters.

<WRAP assertions> is made of these two optional blocks:

- REQUIRE
- ENSURE

Wraps can be defined for any method of the class, whatever its access specifiers may be.

Wraps cannot be associated to an inherited method, defined in a base class. If you want to do so, define a WRAP in a contract associated with the base class.

If the method is virtual, and the WRAP does not belong to a SINGLE CLASS, the wrap definition also applies to any redefinition of the method,

unless a specific wrap has been defined for the redefinition in a daughter class.

Example

C++ source code:

```
class A {
int f ();
char *g (int); // overloaded method
void g(void *); // overloaded method
};
```

C++ Contract Check Script code:

```
CLASS A {
WRAP f /OK */
REQUIRE ( /*... */ )
ENSURE ( /*... */)

WRAP g /ambiguous -> error */
/*... */

WRAP g(int) / OK */
/*... */

WRAP g(void *p) / OK */
/*... */
}
```

INVARIANT

Syntax

```
INVARIANT <native expression>;
```

Location

CLASS

Description

The INVARIANT statement describes a condition that should be always true in an object life, that is, whenever one of its method can be called. It appears in a CLASS block.

<native expression> is a C++ Boolean expression (or an expression that can be converted to a Boolean).

The following symbols can be used in <native expression>:

- Any of the class members.
- Any of the global variables declared in every file where a method of the class (or a method of descendant if it is not a single class contract) is defined.

The following symbols cannot be accessed in <native expression>:

- Local variables of any methods.
- Macros: Global variables that are not defined in at least one file where a method of the class (or one of its descendants, if it is not a single class contract) is defined.

Evaluation:

<native expression> is evaluated at the end of the execution of the class constructors (except the implicitly defined copy constructor), at the beginning of the class destructors, and both at the beginning and the end of other non-static non implicitly defined methods.

Warning: You can call methods in *<expr>*, but you must ensure that these calls do not modify the object's state (that is, they do not write to any field). You can ensure this by calling *const* methods only. If you want the compiler to check this, see the **ATO_AC_STRICT_CHECKING** Target Package option.

Example

C++ source code example:

```
class Stack {
    int count;
    Stack () : count(0) {}
    void push (void *);
    void *pop ();
};
```

C++ Contract Check Script code example:

```
CLASS Stack {
    INVARIANT (count >= 0);
}
```

STATE

Syntax

```
STATE <state name> { (<native expression>) }
```

Location

CLASS

Description

The STATE statement describes a state for the current class, which is defined by the conjunction of one or several Boolean expression.

Note **STATE** by itself does not generate any source code instrumentation. STATE should be used along with the TRANSITION statement.

<state name> is a C++ Test Script Language identifier.

<native expression> is a C++ Boolean expression (or an expression that can be converted to a Boolean).

The following symbols can be used in *<native expression>*:

- Any of the class members.
- Any of the global variables declared in every file where a method of the class (or a method of descendant if it is not a single class contract) is defined.

The following symbols cannot be accessed in *<native expression>*:

- Local variables of any methods.
- Macros.

- Global variables that are not defined in at least one file where a method of the class (or one of its descendants if it is not a single class contract) is defined.

Evaluation

<*native expression*> may be evaluated at the end of the execution of class constructors (except for implicitly defined copy constructors), at the beginning of class destructors, and both at the beginning and the end of other non-static non-implicitly defined methods.

Warning: You can call methods in <*native expression*>, but you must ensure that these calls do not modify the object's state by writing to any fields. You can ensure this by using *const* methods only. If you want the compiler to check this, see the ATO_AC_STRICT_CHECKING Target Package *option*.

Example

C++ source code example:

```
class Stack {
    int count;
    Stack () : count(0) {}
    void push (void *);
    void *pop ();
};
```

C++ Contract Check Script code example:

```
CLASS Stack {
    STATE Empty { (count == 0) }
    STATE NotEmpty { (count > 0) }
}
```

TRANSITION ... TO

Syntax

```
TRANSITION <state name> TO <state name>;
```

Location

CLASS

Description

The TRANSITION statement describes a transition between two states an object can execute during its life.

<state name> is a valid state name defined with the STATE keyword.

Transitions are checked between two state evaluations. States are evaluated at the end of the execution of class constructors (except for implicitly-defined copy constructor), at the beginning of class destructors, and both at the beginning and the end of other non-static non-implicitly defined methods.

All states are evaluated after an object has been created (when leaving a constructor). Consequently, the initial state must be described in a non-ambiguous way: one - and one only - state must occur when leaving a constructor.

Once a state has been determined, only authorized states (according to the defined transitions) are checked. Ambiguity must not occur when choosing the next state.

States are always reflexive. This means that a transition from a state to itself is implicitly defined. There must be no ambiguity between one state and

any other state that can be reached through a single transition.

Example

C++ source code example:

```
class Stack {
    int count;
    int capacity;
    Stack () : count(0) {}
    void push (void *);
    void *pop ();
};
```

C++ Contract Check Script code example:

```
CLASS Stack {
    STATE Empty { (count == 0) }
    STATE NotEmpty { (count > 0) (count < capacity) }
    STATE Full { (count == capacity) }
    TRANSITION Empty TO NotEmpty;
    TRANSITION NotEmpty TO Full;
    TRANSITION Full TO NotEmpty;
    TRANSITION NotEmpty TO Empty;
}
```

Ada Test Script Language

Purpose

The Test RealTime Component Testing feature for Ada uses its own simple language for test scripting, called Ada Test Script Language.

This section describes each keyword of the Ada Test Script language, including:

- Syntax
- Functionality and rules governing its usage
- Examples of use

Notation Conventions

Throughout this guide, command notation and argument parameters use the following standard convention:

Notation	Example	Meaning
BOLD	BEGIN	Language keyword
<i><italic></i>	<i><filename></i>	Symbolic variables
[]	[<option>]	Optional items
{ }	{<filenames>}	Series of values
{ { }	{ {<filenames> } }	Optional series of variables
	on off	OR operator

Note: Ada Test Script Language keywords can be entered in upper or lower case. To distinguish Ada Test Script Language from native code, this guide conventionally uses upper case typography for Ada Test Script Language keywords.

Ada Test Script Language Identifiers

A Ada Test Script Language identifier is a text string used as a label, such as the name of a **TEST** or a **STUB** in a **.ptu** test script.

Identifiers are made of an unlimited sequence of the following characters:

- a-z
- A-Z
- 0-9
- _ (underscore)

Spaces are not valid identifier characters.

Note that identifiers starting with a numeric character are allowed. The following statement, for example, is syntactically correct:

```
TEST 1
...
END TEST
```

Unlike keywords, Ada Test Script Language identifiers are case sensitive. This means that **LABEL**, **label**, and **Label** are three different identifiers.

Ada Test Script Language Keywords

The Ada Test Script Language keywords are not case sensitive. This means that *STUB*, *stub*, and *Stub* are equivalent. For conventional purposes however, this document uses upper-case notation for the Ada Test Script Language keywords in order to differentiate from native source code.

Block Keywords

- ELEMENT ... END ELEMENT
- INITIALIZATION ... END INITIALIZATION
- SERVICE ... END SERVICE
- SIMUL ... ELSE_SIMUL ... END SIMUL
- TERMINATION ... END TERMINATION
- TEST ... END TEST
- DEFINE STUB ... END DEFINE

Other Keywords

- BEGIN
- COMMENT

- EV
- FAMILY
- HEADER
- IF...ELSE...END IF
- INIT
- INCLUDE
- NEXT_TEST
- STUB
- VAR, ARRAY and STR
 - <variable> Parameter
 - <initialization> Parameter
 - <expected> Parameter

BEGIN

Ada Test Script Language

Purpose

The **BEGIN** instruction marks the beginning of the Ada code generation. The **BEGIN GENERIC** option is specifically for testing Ada generic packages.

Syntax

```
BEGIN [ <parent_unit> [ , <procedure> ] ]  
BEGIN GENERIC (<generic_package>, <instance>) [ , <procedure> ]
```

where:

- *<parent_unit>* is the full name of the unit under test.
- *<procedure>* is the name of the generated separate procedure, by default **ATTOL_TEST**.
- *<generic_package>* is the name of a generic unit under test.
- *<instance>* is the name of the instantiated unit from the generic.

Description

The **BEGIN** instruction is mandatory and must be located after a **HEADER** statement, and before any other Ada Test Script instruction.

By default, the Ada Test Script Compiler creates an independent compilation unit. To test private elements of a package you must first generate a procedure.

The reference body to the separate procedure must be written in the parent

unit package.

If a **BEGIN** keyword is not found in the test script, a warning message is generated and a **BEGIN** instruction is implicitly created before the first occurrence of a **SERVICE** instruction.

To test a generic package, you need to generate the test driver separately and call it as a procedure of the instance. Use the **BEGIN GENERIC** syntax to automatically generate a separate procedure *<procedure>* of *<generic_package>*. This allows you to access the procedure *<instance>.<procedure_name>*, which is generated by the Ada Test Script Compiler.

Note This technique also allows testing of private types within the generic package.

COMMENT

Ada Test Script Language

Purpose

The **COMMENT** instruction adds a textual comment to the test report.

Syntax

```
COMMENT [<text>]
```

where:

- *<text>* is an optional comment text string to be displayed.

Description

The **COMMENT** instruction is optional and can be used anywhere in the test script.

The position of the **COMMENT** instruction in the test script defines the position in which the comments appear in the test report.

- Within an **ELEMENT** block: the comment appears just before the variable state descriptions.
- After a **TEST** instruction: the comment appears in the test header, before the variable descriptions.
- After a **SERVICE** instruction: the comment appears in the service header, before the test descriptions.
- Outside a **SERVICE** block: the comment appears in the service header following the declaration, before the test descriptions.

Example

```
TEST 1
FAMILY nominal
COMMENT histogram computation for a black image
ELEMENT
```

DEFINE STUB ... END DEFINE

Ada Test Script Language

Purpose

The **DEFINE STUB** and **END DEFINE** instructions delimit a simulation block consisting of stub definition declarations written in Ada.

Syntax

```
DEFINE STUB <stub_name> [ <stub_dim> ]  
END DEFINE
```

where:

- <stub_name> is the mandatory name of a simulation block.
- <stub_dim> is an optional maximum number of stub call descriptions for a test scenario. By default, its value is 10.

Description

Defining stubs in a test script is optional.

DEFINE STUB / **END DEFINE** blocks must be located after the **BEGIN** instruction and outside any **SERVICE** block.

Using the stub definitions, the Ada Test Script Compiler generates simulation variables and functions for which the interface is identical to that of the stubbed variables and functions.

The purpose of these simulation variables and functions is to store and test input parameters, assign values to output parameters, and if necessary, return appropriate values.

All functions and procedures of the *<stub_name>* package are simulated, and stub definitions are Ada declarations (beginning with '#') of functions, procedures, or assignment instructions.

Stub parameters describe both the type of item used by the calling function and the mode of passing. The mode of passing the parameter is specified by adding the following before the parameter name:

- **in** for input parameters
- **out** for output parameters
- **in out** for input/output parameters
- **no** for parameters that you do not want to test

The parameter mode is optional. If no parameter mode is specified, the **in** mode is assumed by default.

A return parameter is always deemed to be an output parameter.

Global variables defined in **DEFINE STUB** blocks replace the real global variables.

Example

```
BEGIN
DEFINE STUB file_io
END DEFINE
```

ELEMENT ... END ELEMENT

Ada Test Script Language

Purpose

The **ELEMENT** and **END ELEMENT** instructions delimit a test phase or **ELEMENT** block.

Syntax

```
ELEMENT
END ELEMENT
```

Description

The **ELEMENT** instruction is mandatory and can only be located within a **TEST** block. If absent, a warning message is generated and the **ELEMENT** block is implicitly declared before the first occurrence of a **VAR**, **ARRAY**, **STR**, or **STUB** instruction.

The block must end with the instruction **END ELEMENT**. If absent, a warning message is generated and it is implicitly declared before the next **ELEMENT** instruction, or the **END TEST** instruction.

The **ELEMENT** block contains a call to the service under test as well as instructions describing the initializations and checks on test variables.

Positioning of **VAR**, **ARRAY**, **STR** or **STUB** related to the actual test procedure is irrelevant as the Test Compiler separates these instructions into two parts:

- The test initialization (described by **INIT**) is generated with the **ELEMENT** instruction

- The test of the expected value (described by **EV**) is generated with the **END ELEMENT** instruction

Example

```
TEST 1
  FAMILY nominal
  ELEMENT
    VAR x1,          init = 0,          ev = init
    VAR x2,          init = SIZE_IMAGE-1, ev = init
    VAR y1,          init = 0,          ev = init
    VAR y2,          init = SIZE_IMAGE-1, ev = init
    ARRAY image,     init = 0,          ev = init
    VAR histo(0),    init = 0,          ev = SIZE_IMAGE*SIZE_IMAGE
    ARRAY histo(1..SIZE_HISTO-1), init = 0, ev = 0
    #compute_histo(x1,y1,x2,y2,histo);
  END ELEMENT
END TEST
```

EXCEPTION

Ada Test Script Language

Purpose

The **EXCEPTION** instruction describes the behavior of the test script if any exceptions are raised during the execution.

Syntax

```
EXCEPTION <exception_name>
```

Description

This instruction can only appear in an **ELEMENT** block.

<exception_name> is the name of the exception under test.

This instruction must be unique in the block where it appears. If it is absent, the test shall not raise any exception, otherwise, an error is generated.

Only exceptions raised by the procedure under test can be tested. Exceptions raised during the initialization of the variables or during the test of the variables cannot be tested. They are nevertheless detected and written in the test report.

Note Do not use the **EXCEPTION** statement simultaneously with any native exception handling code, as this will create internal conflicts.

Example

In this example, the exception class is **overflow**.

```
ELEMENT
```

```
-- The test shall raise the overflow exception
EXCEPTION overflow
....
-- Using the 'exception' variable
VAR exception->chl, ....
END ELEMENT
```

FAMILY

Ada Test Script Language

Purpose

The **FAMILY** instruction groups tests by families or classes.

Syntax

```
FAMILY <family_name> { , <family_name> }
```

Argument

<family_name> is a mandatory identifier indicating the name of the test family. Typically, you could specify nominal, structural, or robustness families.

Description

The **FAMILY** instruction appears within **TEST** blocks, where it defines the families to which the test belongs.

When you run the test sequence, you can request that only tests of a given *family* are executed.

A test can belong to several families. In this case, the **FAMILY** instruction contains a <family_name> list, separated by commas.

The **FAMILY** instruction must be located before the first **ELEMENT** block of the **TEST** block and must be unique in the **TEST** block.

The **FAMILY** instruction is optional. If it is omitted, a warning message is generated and the test belongs to every family.

Example

```
TEST 1
  FAMILY nominal
  COMMENT histogram computation on a black image
  ELEMENT
```

HEADER

Ada Test Script Language

Purpose

The **HEADER** instruction specifies the name and version of the module under test as well as the version number of the test script.

Syntax

```
HEADER <module_name>, <module_version>, <test_plan_version>
```

<module_name>, *<module_version>* and *<test_plan_version>* are character strings with no restrictions, except for versions beginning with a dollar sign (**\$**). These instructions must be followed by an identifier.

Description

This information contained in the **HEADER** keyword is reproduced in the test report header to identify the test sequence.

The module and test script versions can be read from the environment variables if they are identifiers beginning with a dollar sign (**\$**).

The **HEADER** instruction is mandatory, but its arguments are optional. It must be the first instruction in the test program. If it is absent, a warning message is generated.

Example

```
HEADER histo, 01a, 01a
BEGIN
```


IF ... ELSE ... END IF

Ada Test Script Language

Purpose

The **IF**, **ELSE** and **END IF** statements allow conditional generation of the test driver.

Syntax

```
IF <condition> { , <condition> }  
ELSE  
END IF
```

where:

- *<condition>* is an identifier sent by the **-define** option to the Ada Test Script Compiler.

Description

These statements enclose portions of script that are included depending on the presence of one of the conditions in the list provided to the Ada Test Compiler by the **-define** option.

The *<condition>* list forms a series of conditions that is equivalent to using an expression of logical **ORs**.

The **IF** instruction starts the conditional generation block.

The **END IF** instruction terminates this block.

The **ELSE** instruction separates the condition block into 2 parts, one being

included when the other is not.

Associated Rules

This block of instructions can appear anywhere in the test program.

<*condition*> is any identifier. You must have at least one condition in an **IF** instruction.

This block can contain any code written in Ada Test Script Language or native Ada.

IF and **END IF** instructions must appear simultaneously.

The **ELSE** instruction is optional.

The generation rules are as follows:

- If at least one of the conditions specified in the **IF** instruction's list of conditions appears in the list associated with the -define option, the first part of the block is included.
- If none of the conditions specified in the **IF** instruction appears in the list associated with the -define option, then the second part of the block is included (if **ELSE** is present).

Example

```
IF test_on_target
  VAR register, init == , ev = 0
ELSE
  VAR register, init = 0 , ev = 0
END IF
```

INCLUDE

Ada Test Script Language

Purpose

The **INCLUDE** statement specifies an external file for the Ada Test Compiler to process.

Syntax

```
INCLUDE CODE <file.ada>
INCLUDE PTU <file.ptu>
```

where:

- <file.ada> is the file name of an external Ada source file
- <file.ptu> is the file name of an Ada test script

Description

When an **INCLUDE** instruction is encountered, the Ada Test Compiler leaves the current file, and starts pre-processing the specified file. When this is done, the Ada Test Compiler returns to the current file at the point where it left.

Including a file with the additional keyword **CODE** lets you include a source file without having to start every line with a hash character (#).

Including a file with the additional keyword **PTU** lets you include an Ada test script within another Ada test script.

The name of the included file can be specified with an absolute path or a path relative to the current directory.

If the file is not found in the current directory, all directories specified by the **-incl** option are searched when the preprocessor is started.

If it is still not found or if access is denied, an error is generated.

Example

```
INCLUDE CODE file1.ada  
INCLUDE CODE ../file2.ada  
INCLUDE PTU /usr/tests/file3.ptu
```

INITIALIZATION ... END INITIALIZATION

Ada Test Script Language

Purpose

Specifies native Ada code to initialize the test driver

Syntax

```
INITIALIZATION  
END INITIALIZATION
```

Description

The **INITIALIZATION** and **END INITIALIZATION** statements let you provide native Ada code that is integrated as the first *main* statements of the test driver.

In some environments, such as when using a different target machine, this is a convenient way to initialize the target.

An **INITIALIZATION** block must appear after the **BEGIN** instruction or between two **SERVICE** blocks.

This block can only contain native Ada code. Each line of native code must be preceded with '#' or '@'.

There is no limit to the number of **INITIALIZATION** blocks. Upon test driver generation, they are concatenated in the order in which they appeared in the test script.

NEXT_TEST

Ada Test Script Language

Purpose

The **NEXT_TEST** instruction starts a **TEST** block that is linked to the previous test block.

Syntax

```
NEXT_TEST [ LOOP <nb> ]
```

where:

- <nb> is an integer expression strictly greater than 1.

Description

The **NEXT_TEST** instruction starts a **TEST** block linked to the previous one. It contains one more **ELEMENT** block. It does not contain the **FAMILY** instruction.

For this new test, a number of iterations can be specified by the keyword **LOOP**.

The **NEXT_TEST** instructions can only appear in a **TEST ... END TEST** block.

The main difference between a **NEXT_TEST** block and an **ELEMENT** block is when you use an **INIT IN** statement within a test block:

- If the **INIT IN** is in a **TEST** block, there will be a loop over the entire **TEST** block, without consideration of the **ELEMENT** blocks that it might contain.

- If the **INIT IN** is inside a **NEXT_TEST** block however, the loop will not affect the **ELEMENT** blocks within other **TEST** blocks

Example

```
SERVICE COMPUTE_HISTO
# x1, x2, y1, y2 : integer ;
# histo : T_HISTO ;
TEST 1
  FAMILY nominal
  ELEMENT
    . . .
  END ELEMENT
NEXT_TEST LOOP 2
ELEMENT
```

SERVICE ... END SERVICE

Ada Test Script Language

Purpose

A **SERVICE** block contains a common description for all tests related to a given service of the module under test.

Syntax

```
SERVICE <service_name>  
END SERVICE
```

where:

- *<service_name>* specified the tested service in the test report

Description

The **SERVICE** instruction starts a **SERVICE** block. This block contains the description of all the tests relating to a given service of the module to be tested.

The *<service_name>* is usually the name of the service under test, although this is not mandatory.

The **END SERVICE** instruction indicates the end of the service block.

Associated Rules

The **SERVICE** instruction must appear after the **BEGIN** instruction.

The *<service_name>* parameter can be any identifier. It is obligatory.

Example

```
BEGIN
SERVICE COMPUTE_HISTO
  # x1, x2, y1, y2 : integer ;
  # histo : T_HISTO ;
TEST 1
FAMILY nominal
```

SERVICE_TYPE

Ada Test Script Language

Purpose

The **SERVICE_TYPE** statement indicates the type of service tested.

Syntax

```
SERVICE_TYPE <type> {, <type>}
```

where:

- <type> is a user-defined service type identifier

Description

The **SERVICE_TYPE** instruction allows you to specify an identifier indicating the type of service tested. This identifier is included in the test report.

You can use this functionality to specify whether a service is internal or external.

If **SERVICE_TYPE** is placed within a **SERVICE ... END SERVICE** block, it indicates the type of the current **SERVICE** block.

If the **SERVICE_TYPE** statement is placed outside a **SERVICE** block, then it indicates the default service type for all **SERVICE** blocks that do not contain a **SERVICE_TYPE** statement.

Example

```
SERVICE_TYPE internal, external  
SERVICE count
```

```
SERVICE_TYPE internal
...
END SERVICE
```

SIMUL ... ELSE_SIMUL ... END SIMUL

Ada Test Script Language

Purpose

The **SIMUL**, **ELSE_SIMUL**, and **END SIMUL** instructions allow conditional generation of test driver.

Syntax

```
SIMUL
ELSE_SIMUL
END SIMUL
```

Description

Code enclosed within a **SIMUL** block is conditionally generated depending on the status of the **Simulation** configuration setting in the Test RealTime GUI, or the **-nosimulation** command line option of the Ada Test Script Compiler.

The **SIMUL** instruction starts the conditional generation block.

The **END SIMUL** instruction marks the end of the conditional block.

The **ELSE_SIMUL** instruction separates this block into two parts, one being included when the other is not, and vice versa.

This block of instructions can appear anywhere in the test program and can contain both Ada Test Script Language or native Ada code.

The **SIMUL** and **END SIMUL** instructions must appear as a pair. One

cannot be used without the other.

The **ELSE_SIMUL** instruction is optional.

When using the Test RealTime user interface, select or clear the **Simulation** option in the **Component Testing for Ada** tab of the **Configuration Settings** dialog box.

The code generation rules are as follows:

- If **Simulation** is enabled => the first part of the **SIMUL** block is included.
- If **Simulation** is disabled => the second part of the block (**ELSE_SIMUL**) is included if it exists. If there is no **ELSE_SIMUL** statement, then the **SIMUL** block is ignored.

Example

```
SIMUL
  #x := 0;
ELSE_SIMUL
  #x := 1;
END SIMUL
...
SIMUL
  VAR x , INIT = 0 , EV = 1
  VAR p , INIT = NIL , EV = NONIL
ELSE_SIMUL
  VAR x , INIT = 0 , EV = 0
  VAR p , INIT = NIL , EV = NIL
END SIMUL
```

STUB

Ada Test Script Language

Purpose

The STUB instruction for Ada describes all calls to a simulated function in a test script.

Syntax

```
STUB <stub_name> [<slice>] ( [<param_val> {, <param_val> } ] )  
[<return_val>] { , [<slice>] ( [ <param_val> { , <param_val> } ] ) [ <return_val> ] }
```

Description

The following is described for every parameter of this function and for every expected call:

- For **in** parameters, the values passed to the function. These values are stored and tested during execution.
- For **out** parameters and, where appropriate, the return value, the values returned by the function. These values are stored in order to be returned during execution.
- For **in out** and **in access** parameters, both the previous two values are required.
- For **no** parameters, no expression is required.

<stub_name> is the name of the simulated procedure or function. It is obligatory. You must previously have described this procedure or function in a **DEFINE STUB** block.

<param_val> is an expression describing the test values for in parameters

and the returned values for out parameters. If named, parameters can be in any order. For in out parameters, *<param_val>* is expressed in the following way:

```
( [IN =>]<in_param_val> , [OUT =>]<out_param_val> )
```

If you use the optional **IN =>** and **OUT =>** specifiers, you can invert the order of the parameters.

<return_val> is an expression describing the value returned by the function if its type is not void. Otherwise, no value is provided.

You must give values for every in, out and in out parameter; otherwise, a warning message is generated. The no parameters are ignored.

<param_val> and *<return_val>* are Ada expressions that can contain:

- Numeric (integer or floating-point), character, or character string literal values. Strings can be delimited by single or double inverted commas.
- Constants, in the Ada sense of the word, which can be numeric, characters, or character strings
- Variables belonging to the test program or the module to be tested
- Ada functions
- The keyword **NIL** to designate a null pointer
- Pseudo-variables **I**, **I1**, **I2** ..., **J**, **J1**, **J2** ..., where **I_n** is the current index of the *n*th dimension of the parameter and **J_m** the current number of the subtest generated by the test scenario's *m*th **INIT IN**, **INIT FROM** or **LOOP**; the **I** and **I1** variables are therefore equivalent as are **J** and **J1**; the subtest numbers begin at **1** and are incremented by **1** at each iteration
- An Ada expression with one or more of the above elements combined using any of the Ada operators and casting, with all required levels of parentheses, and conforming to Ada rules of syntax and semantics

- For arrays and structures, aggregates between parentheses ('()') or brackets ('[]').

<param_val> can contain for an **in** value:

- The <-> expression to specify that the parameter should be ignored
- The <value> <-> <value> expression to specify a range of values for the parameter

<param_val> can contain for an **out** value or **return** value:

- The == expression to specify that the parameter should not be set

If are using one of the above expressions, you can specify the type of parameter by using the ==: <type> syntax for the **out** and **return** value or <:>: <type> for the **in** value.

<return_val> can also refer to an Ada exception name introduced by the following syntax:

[:<return_type>] **RAISE** <exception_name>

where :<return_type> is used to specify the function returned type in case of overloading.

You must describe at least one call in the **STUB** instruction. Several descriptions can occur separated by commas.

The <slice> expression can be used to specify the maximum number of calls to be recorded.

STUB instructions can appear in **ELEMENT** blocks.

Example

```
STUB open_file ("file1")3
STUB create_file ("file2")4
STUB read_file 1..2 =>(3,"line 1",1)1,(3,"line 2",2<->3)1,
  & 4..7 =>(3,"",0)0
STUB write_file (4,"line 1")1,(4,"line 2")1
STUB close_file (3)1,(4)1, (<->) RAISE DEVICE_ERROR
```

TERMINATION ... END TERMINATION

Ada Test Script Language

Syntax

```
TERMINATION
END TERMINATION
```

Description

The **TERMINATION** and **END TERMINATION** instructions delimit a block of native code that is integrated into the generation process as the last *main* statements to be executed.

In some environments, such as when using a different target machine, this is a convenient way to exit the target.

Associated Rules

A **TERMINATION ... END TERMINATION** block must appear after the **BEGIN** instruction and outside any **SERVICE** block.

This block can only contain native Ada code. Each line of native code must be preceded with '#' or '@'.

There is no limit to the number of **TERMINATION** blocks. Upon test driver generation, they are concatenated in the order in which they appeared in the test script.

TEST ... END TEST

Ada Test Script Language

Syntax

```
TEST <test_name> [ LOOP <nb>]  
END TEST
```

Description

The **TEST** instruction starts a **TEST** block. This block describes the test case for a service. It contains one more **ELEMENT** blocks specifying the test.

In the test report, the <test_name> parameter flags the test within the **SERVICE** block. Tests are usually given numbers in ascending order.

A number of iterations can be specified for each test with the optional **LOOP** keyword.

The **TEST LOOP** statement can generate graph metric results in a **.rtx** file. To do this, you must set the environment variable **ATURTX** to *True*. The produced **.rtx** graph can be viewed in the Test RealTime Graphic Viewer.

The **END TEST** instruction marks the end of the **TEST** block.

Associated Rules

The **TEST** and **END TEST** instructions can only appear in a **SERVICE** block.

<test_name> is obligatory. If it is absent, the Test Compiler generates an error message.

<nb> is an integer expression strictly greater than 1.

Example

```
SERVICE COMPUTE_HISTO
# int x1, x2, y1, y2 : integer ;
# histo : T_HISTO ;
TEST 1
FAMILY nominal
ELEMENT
```

VAR, ARRAY, and STR

Ada Test Script Language

Purpose

The **VAR**, **ARRAY**, and **STR** instructions declare the test of a simple variable, a variable array or a variable structure.

Syntax

```
VAR <variable>, <initialization>, <expected>
ARRAY <variable>, <initialization>, <expected>
STR <variable>, <initialization>, <expected>
VAR <expression>, <expected>
ARRAY <expression>, <expected>
STR <expression>, <expected>
```

where:

- *<variable>* is a variable
- *<expression>* is a valid Ada expression that is to be tested
- *<initialization>* is an initialization parameter
- *<expected value>* is an expected parameter

Description

Use the **VAR**, **ARRAY**, and **STR** instructions to declare a variable test. During test execution, if the value of the variable is out of the bounds specified in the *<expected>* expression, the test is *Failed*.

VAR, **ARRAY** or **STR** are synonymous and do not change the way in

which the result displayed in the test report.

- **VAR**: This statement is for simple variables.
- **ARRAY**: This statement is for variable arrays.
- **STR**: This statement is for variable structures.

If you use a **VAR** statement to test an array or structure, the report lists each element of the array or structure.

The **VAR**, **ARRAY**, and **STR** instructions must be located in an **ELEMENT** or an **ENVIRONMENT** block.

VAR, ARRAY and STR <variable> Parameter

Description

In conjunction with the **VAR**, **ARRAY** and **STR** keywords, the <variable> parameter for Ada is a conventional notation name for an Ada variable under test.

Associated Rules

<variable> can be a simple variable (integer, floating-point number, character, pointer, character string, ...), an element of an array or record, part of an array, an entire array, or a complete record.

If the variable is an array for which no test boundaries have been specified, all the array elements are tested. Similarly, if the variable is a record of which one of the fields is an array, all elements of this field are tested.

Brackets or parentheses can be used to index array variables.

The variable must have been declared in Ada before it is used in the **.ptu** test script.

Example

```
VAR x, ...
VAR y(4), ...
VAR z.field, ...
VAR p.value, ...
ARRAY y(0..100), ...
ARRAY y, ...
STR z, ...
STR p.all, ...
```

VAR, ARRAY and STR <expected> Parameter

Ada Test Script Language

Purpose

In conjunction with the **VAR**, **ARRAY** and **STR** keywords, the *<expected value>* parameters for Ada specify the expected value of a variable.

Syntax

```
EV = <exp>
EV = <exp> , DELTA = <delta>
MIN = <exp>, MAX = <exp>
EV IN { <exp>, <exp>, ... }
EV ( <variable> ) IN { <exp>, <exp>, ... }
EV ==
```

where:

- *<exp>* can be any of the expressions of the Initialization Parameters, plus the following expressions:
- *<delta>* is the acceptable tolerance of the expected value and can be expressed:
- *<variable>* is an Ada variable

Description

The *<expected value>* expressions are used to specify a test criteria by comparison with the value of a variable. The test is considered Passed when the actual value matches the *<expected value>* expression.

An acceptable tolerance *<delta>* can be expressed:

- As an absolute value, by a numerical expression in the form described above
- As a percentage of the expected value. Tolerance is then written in the form *<exp>%*.

Expected values can be expressed in the following ways:

- **EV** = *<exp>* specifies the expected value of the variable when it is known in advance. The value of variable is considered correct if it is equal to *<exp>*.
- **EV** = *<exp>*, **DELTA** = *<tolerance>* allows a tolerance for the expected value. The value of variable is considered correct if it lies between *<exp>* - *<tolerance>* and *<exp>* + *<tolerance>*.
- **MIN** = *<exp>* and **MAX** = *<exp>* specify an interval delimited by an upper and lower limit. The value of the variable is considered correct if it lies between the two expressions. Characters and character strings are treated in dictionary order.
- **EV IN** { *<exp>*, *<exp>*, ... } specifies the values expected successively, in accordance with the initial values, for a variable that is declared in **INIT IN**. It is therefore essential that the two lists have an identical number of values.
- **EV (<variable>) IN** is identical to **EV IN**, but the expected values are a function of another variable that has previously been declared in **INIT IN**. As for **EV IN**, the two lists must have an identical number of values.
- **EV ==** allows the value of *<variable>* not to be checked at the end of the test. Instead, this value is read and displayed. The value of *<variable>* is always considered correct.

Expressions

The initialization expressions *<exp>* can be among any of the following values:

1. Numeric (integer or floating-point), character, or character string literal values. Strings can be delimited by single or double quotes
 2. Native constants, which can be numeric, characters, or character strings
 3. Constants, previously defined with the **CONST** instruction
 4. Variables belonging to the test program or the module to be tested
 5. Ada functions
 6. The keyword **NIL** to designate a null pointer
 7. The keyword **NONIL**, which tests if a pointer is non-null
 8. Pseudo-variables **I**, **I1**, **I2** ..., **J**, **J1**, **J2** ..., where **I_n** is the current index of the *n*th dimension of the parameter and **J_m** the current number of the subtest generated by the test scenario's *m*th **INIT IN**, **INIT FROM** or **LOOP**; the **I** and **I1** variables are therefore equivalent as are **J** and **J1**; the subtest numbers begin at **1** and are incremented by **1** at each iteration
1. An Ada expression with one or more of the above elements combined using any operators and casting, with all required levels of parentheses, the + operator being allowed to concatenate character string variables
 2. For arrays and structures, any of the above-mentioned expressions between brackets ('[]') for Ada, including when appropriate:

For an array element, part of an array or a structure field, its index, interval or name followed by '=' and by the value of the array element, common to all elements of the array portion or structure field

For structures you can test some fields only, by using the following syntax:

[<fieldname> => <value>, <fieldname> => <value>]

3. The keyword **OTHERS** followed by '=' and the default value of any array elements or structure fields not yet mentioned
4. The pseudo-variable **INIT**, which copies the initialization expression

Additional Rules

EV with **DELTA** is only allowed for numeric variables.

MIN = <exp> and **MAX** = <exp> are only allowed for alphanumeric variables that use lexicographical order for characters and character strings.

MIN = <exp> and **MAX** = <exp> are not allowed for pointers.

Only **EV** = and **EV** == are allowed for structured variables.

In some cases, in order to avoid generated code compilation warnings, the word **CAST** must be inserted before the **NIL** or **NONIL** keywords.

Example

```
VAR x, ..., EV = pi/4-1
VAR y[4], ..., EV IN { 0, 1, 2, 3 }
VAR y[5], ..., EV(y[4]) IN { 10, 11, 12, 13 }
VAR z.field, ..., MIN = 0, MAX = 100
VAR p->value, ..., EV ==
ARRAY y[0..100], ..., EV = cos(I)
```

```
ARRAY y, ..., EV = {50=>10,others=>0}  
STR z, ..., EV = {0, "", NIL}  
STR *p, ..., EV = {value=>4.9, valid=>1}
```

VAR, ARRAY and STR <initialization> Parameter

Ada Test Script Language

Purpose

In conjunction with the **VAR**, **ARRAY** and **STR** keywords, the <initialization> parameters for Ada specify the initial value of the variable.

Syntax

```
INIT = <exp>
INIT IN { <exp>, <exp>, ... }
INIT ( <variable> ) WITH { <exp>, <exp>, ... }
INIT FROM <exp> TO <exp> [STEP <exp> | NB TIMES <nb> | NB RANDOM
<nb> [+ BOUNDS]]
INIT ==
```

where:

- <exp> is an expression as described below.
- <nb> is an integer constant that is either literal or derived from an expression containing native constants or constants defined with a **CONST** instruction
- <variable> is an Ada variable

Description

The <initialization> expressions are used to assign an initial value to a variable. The initial value is displayed in the Component Testing report for Ada.

Initializations can be expressed in the following ways:

- **INIT** = *<exp>* initializes a variable before the test with the value *<expression>*.
- **INIT IN** { *<exp>* , *<exp>* , ...} declares a list of initial values. This is a condensed form of writing that enables several tests to be contained within a single instruction.
- **INIT** (*<variable>*) **WITH** { *<exp>* , *<exp>* , ...} declares a list of initial values that is assigned in correlation with those of the variable initialized by an **INIT IN** instruction. There must be the same number of initial values.

The **INIT IN** and **INIT** (*<variable>*) **WITH** expressions cannot be used with for arrays that were initialized in extended mode or for structures.

1. **INIT FROM** *<lower>* **TO** *<upper>* allows the initial value of a numeric variable (integer or floating-point) to vary between lower and upper boundary limits:

STEP: the value varies by successive steps

NB_TIMES *<nb>*: the value varies by a number *<nb>* of values that are equidistant between the two boundaries, where *<nb>* \geq 2

NB_RANDOM *<nb>*: the value varies by generating random values between the 2 boundaries, including, when appropriate, the boundaries, where *<nb>* \geq 1

The **INIT FROM** expression can only be used for numeric variables.

- **INIT** == allows the variable to be left un-initialized. You can thus control the values of variables that are dynamically created by the service under test. The initial value is displayed in the test report as a question mark (?).

An initialization expression can still be used (**INIT** == *<expression>*) to

include of expected value expression when using the **INIT** pseudo-variable is used. See Expected_Value Expressions.

Expressions

The initialization expressions *<exp>* can be among any of the following values:

1. Numeric (integer or floating-point), character, or character string literal values. Strings can be delimited by single or double quotes
2. Native constants, which can be numeric, characters, or character strings
3. Constants, previously defined with the **CONST** instruction
4. Variables belonging to the test program or the module to be tested
5. Ada functions
6. The keyword **NIL** to designate a null pointer
7. Pseudo-variables **I**, **I1**, **I2** ..., **J**, **J1**, **J2** ..., where **I_n** is the current index of the *n*th dimension of the parameter and **J_m** the current number of the subtest generated by the test scenario's *m*th **INIT IN**, **INIT FROM** or **LOOP**; the **I** and **I1** variables are therefore equivalent as are **J** and **J1**; the subtest numbers begin at **1** and are incremented by **1** at each iteration
8. An Ada expression with one or more of the above elements combined using any operators and casting, with all required levels of parentheses, the + operator being allowed to concatenate character string variables
9. For arrays and structures, any of the above-mentioned expressions

between brackets ('[]') for Ada, including when appropriate:

For an array element, part of an array or a structure field, its index, interval or name followed by '=>' and by the value of the array element, common to all elements of the array portion or structure field

For structures you can test some fields only, by using the following syntax:

[<fieldname>=> <value>, <fieldname>=> <value>]

10. The keyword **OTHERS** followed by '=>' and the default value of any array elements or structure fields not yet mentioned
11. For **INIT IN** and **INIT WITH** only, a list of values delimited by brackets ('[]') for Ada composed of any of the previously defined expressions

Additional Rules

Any integers contained in an expression must be written either in accordance with native lexical rules, or under the form:

- <hex_integer>**H** for hexadecimal values. In this case, the integer must be preceded by **0** if it begins with a letter
- <binary_integer>**B** for binary values

Example

```
VAR x, INIT = pi/4-1, ...
VAR y[4], INIT IN { 0, 1, 2, 3 }, ...
VAR y[5], INIT(y[4]) WITH { 10, 11, 12, 13 }, ...
VAR z.field, INIT FROM 0 TO 100 NB_RANDOM 3, ...
VAR p->value, INIT ==, ...
ARRAY y[0..100], INIT = sin(I), ...
ARRAY y, INIT = {50=>10,others=>0}, ...
STR z, INIT = {0, "", NIL}, ...
```



```
STR *p, INIT = {value=>4.9, valid=>1}, ...
```

Java Test Primitives

All the test primitives described here are available for both J2SE and J2ME versions of the JUnit testing framework.

This section describes each method used by Component Testing for Java, including:

- Syntax
- Functionality and rules governing its usage
- Examples of use

Notation Conventions

Throughout this guide, command notation and argument parameters use the following standard convention:

Notation	Example	Meaning
BOLD	BEGIN	Language keyword
<i><italic></i>	<i><string></i>	Symbolic variables
[]	[<i><option></i>]	Optional items
{ }	{ <i><string></i> }	Series of values
[{ }]	[{ <i><string></i> }]	Optional series of variables
	on off	OR operator

JUnit Assert Primitives

Component testing for Java uses the standard *assert* test primitives provided by JUnit. Please refer to documentation provided with JUnit documentation for further information.

Method	Purpose
assertEquals()	Checks that two values are equal.
assertNotNull()	Checks that an object is not null.
assertNull()	Checks that an object is null.
assertSame()	Checks that two object are actually the same.
assertTrue()	Checks that a condition is true.
fail()	Marks a test as <i>Failed</i> .

Extended Primitives

Component Testing for Java extends the set of *assert* test primitives provided by JUnit with a special set of *verify* primitives. These extended test methods are part of the TestCase class. User test classes must derive from TestCase to use these primitives.

Method	Purpose
verifyEquals()	Checks that two values are equal.
verifyNotNull()	Checks that an object is not null.
verifyNull()	Checks that an object is null.
verifySame()	Checks that two object are actually the same.
verifyTrue()	Checks that a condition is true.
verify()	Checks that an exception is thrown.
verifyApproxEquals()	Checks that two variables have the same value within a given margin.
verifyGreaterThan()	Checks that a tested value is strictly greater than a reference value.
verifyGreaterThanEquals() s()	Checks that a tested value is greater or equal to reference value.
verifyLowerThan()	Checks that a tested value is strictly lower than a reference value.
verifyLowerThanEquals() ()	Checks that a tested value is lower or equal to reference value.
verifyLogMessage()	Logs a message in the test report.
verifyLogfail()	Marks a test as <i>Failed</i> with a message.

Timer Primitives

Component Testing for Java provides multiple timer control primitives, allowing

you to perform basic performance testing.

Method	Purpose
<code>createTimer()</code>	Creates a timer.
<code>timerStart()</code>	Starts a timer.
<code>timerReportElapsedTime()</code>	Logs the elapsed time since a timer was started.
<code>e()</code>	
<code>verifyElapsedTime()</code>	Checks that the elapsed time for a timer is below a given value.

assertEquals()

Purpose

Checks that two values are equal.

Syntax

```
assertEquals( [<string> ,] <val1>, <val2> [, <precision>] )
```

where:

- *<string>* is an optional message
- *<val1>* and *<val2>* are two Java type values
- *<precision>* is an optional precision argument for Float or Double types

Description

The corresponding test result is *Passed* if the *<val1>* and *<val2>* are equal within a given *<precision>* margin if specified and *Failed* if the condition is *False*.

Compared values may be of any Java type: *Boolean*, *float*, *double*, *short*, *byte*, *char*, *int*, *long* or *object*. *<val1>* and *<val2>* must be of the same type.

An optional *<string>* message can be logged and displayed in the test report.

An optional *<precision>* argument specifies an acceptable margin for Float or Double values. By default, *<val1>* and *<val2>* must be strictly equal.

If an exception is thrown in the **assertEquals()** method, the test is stopped.

Examples

Boolean:

```
assertEquals("assert equals true", true, true);
assertEquals("assert equals false", true, false);
assertEquals(true, true);
assertEquals(true, false);
```

Float:

```
float val1;
float val2;
val1 = 1;
val2 = 1;
assertEquals("assert equals true", val1, val2, 0.1);
val2 = 3;
assertEquals("assert equals false", val1, val2, 0.1);
```

```
float val1;
float val2;
val1 = 1;
val2 = 1;
assertEquals(val1, val2, 0.1);
val2 = 3;
assertEquals(val1, val2, 0.1);
```

Double:

```
double val1;
double val2;
val1 = 1.05;
val2 = 1.05;
assertEquals("assert equals true", val1, val2, 0.01);
val2 = 1.06;
assertEquals("assert equals false", val1, val2, 0.0001);
```

```
double val1;
double val2;
val1 = 1.05;
val2 = 1.05;
assertEquals(val1, val2, 0.01);
val2 = 1.06;
assertEquals(val1, val2, 0.0001);
```

Short:

```
short val1 = 1;
short val2 = 1;
assertEquals("assert equals true short", val1,val2);
```

Byte:

```
byte val1 = 1;
byte val2 = 1;
assertEquals(val1,val2);
```

Char

```
char val1 = 'a';
char val2 = 'a';
assertEquals("assert equals true char", val1,val2);
```

Int

```
int val1 = 1;
int val2 = 1;
assertEquals("assert equals true int", val1,val2);
```

Long

```
long val1 = 1;
long val2 = 1;
assertEquals("assert equals true long", val1,val2);
```

Object

```
long val1 = 1;
long val2 = 1;
assertEquals("assert equals true Object", new Long(val1), new
Long(val2));
```

assertNotNull()

Purpose

Checks that an object is not null.

Syntax

```
assertNotNull( [ <string>, ] <object> )
```

where:

- *<string>* is an optional message
- *<object>* is a Java object

Description

The corresponding test result is *Passed* if *<object>* is not null, and *Failed* if the condition is *False*.

An optional *<string>* message can be logged and displayed in the test report.

If an exception is thrown in the **assertNotNull()** method, the test is stopped.

Examples

```
Object one_obj = new Long(10);  
assertNotNull("assert not null passed", one_obj);
```

```
Object one_obj = new Long(10);  
assertNotNull(one_obj);
```

assertNull()

Purpose

Checks that an object is not null.

Syntax

```
assertNull( [<string>,] <object> )
```

where:

- *<string>* is an optional message
- *<object>* is a Java object

Description

The corresponding test result is *Passed* if *<object>* is null, and *Failed* if the condition is *False*.

An optional *<string>* message can be logged and displayed in the test report.

If an exception is thrown in the **assertNull()** method, the test is stopped.

Examples

```
Object one_obj = null;  
assertNull("assert null passed", one_obj);
```

```
Object one_obj = null;  
assertNull(one_obj);
```


assertSame()

Purpose

Checks that two object are actually the same.

Syntax

```
assertSame ( [ <string> ] , <object1> , <object2> )
```

where:

- <string> is an optional message
- <object1> and <object2> are two Java objects

Description

The corresponding test result is *Passed* if <object1> and <object2> refer to the same object, and *Failed* if the condition is *False*.

An optional <string> message can be logged and displayed in the test report.

If an exception is thrown in the **assertSame()** method, the test is stopped.

Examples

```
Object one_obj = new Long(10);  
Object other=one_obj;  
assertSame("assert Same passed", one_obj,other);
```

```
Object one_obj = new Long(10);  
Object other=one_obj;  
assertSame(one_obj, other);
```

assertTrue()

Purpose

Checks that a condition is true.

Syntax

```
assertTrue( [<string>,] <Boolean> )
```

where:

- *<string>* is an optional message
- *<Boolean>* is a Boolean condition

Description

The corresponding test result is *Passed* if the **assertTrue()** condition is *True* and *Failed* if the condition is *False*.

An optional *<string>* message can be logged and displayed in the test report.

If an exception is thrown in the **assertTrue()** method, the test is stopped.

Examples

```
assertTrue("Should be true",true);  
assertTrue("Should be failed",false);  
assertTrue(true);  
assertTrue(false);
```

fail()

Purpose

Marks a test as *Failed*.

Syntax

```
fail( [<string>] )
```

where:

- *<string>* is an optional message

Description

The corresponding test result is *Failed* if the **fail()** method is encountered.

An optional *<string>* message can be logged and displayed in the test report.

If an exception is thrown in the **fail()** method, the test is stopped.

Examples

```
verifyLogMessage("test fail");  
fail();
```

```
verifyLogMessage("test fail");  
fail("test fail");
```

verifyEquals()

Purpose

Checks that two values are equal.

Syntax

```
verifyEquals( [<string>,] <Boolean1>, <Boolean2> )  
verifyEquals( [<string>,] <val1>, <val2> [, <precision>] )  
verifyEquals( [<string>,] <vector1>, <vector2> [, <precision>] )
```

where:

- *<string>* is an optional message
- *<Boolean1>* and *<Boolean2>* are two Boolean conditions
- *<val1>* and *<val2>* are two Java type values
- *<vector1>* and *<vector2>* are two vectors
- *<precision>* is an optional precision argument for Float or Double types

Description

The corresponding test result is *Passed* if the **verifyEquals()** condition is *True* and *Failed* if the condition is *False*..

Compared values may be of any Java type: *Boolean*, *float*, *double*, *short*, *byte*, *char*, *int*, *long*, *object* or *vectors*. *<val1>* and *<val2>* must be of the same type.

An optional *<string>* message can be logged and displayed in the test report.

An optional *<precision>* argument specifies an acceptable margin for *Float* or *Double* values. By default, *<val1>* and *<val2>* must be strictly equal.

If an exception is thrown in a **verifyEquals()** method, an error is logged and the test continues.

Examples

Char:

```
char valc1 = 'a';
char valc2 = 'a';
verifyEquals("verify equals true char", valc1, valc2);
valc2 = 'b';
verifyEquals("verify equals false char", valc1, valc2);
```

Short:

```
short val1 = 1;
short val2 = 1;
verifyEquals("verify equals true short", val1, val2);
val2 = 2;
verifyEquals("verify equals false short", val1, val2);
```

Byte:

```
byte b1 = 1;
byte b2 = 1;
verifyEquals("verify byte equals true", b1, b2);
b2 = 2;
verifyEquals("verify byte equals false", b1, b2);
```

Int:

```
int i1 = 1;
int i2 = 1;
verifyEquals("verify equals int true", i1, i2);
i1 = 2;
verifyEquals("verify equals int false", i1, i2);
```

Long:

```
long l1 = 1;
long l2 = 1;
verifyEquals("verify equals long true", l1, l2);
l1 = 2;
```

```
verifyEquals("verify equals long false", l1, l2);
```

Object:

```
long l1 =1;
long l2 =1;
verifyEquals("verify equals object true", new Long(l1), new
Long(l2));
l1 = 3;
verifyEquals("verify equals object false", new Long(l1), new
Long(l2));
```

Double:

```
double vald1 = 1;
double vald2 = 1;
verifyEquals("verify equals true double", vald1, vald2);
vald2 = 2;
verifyEquals("verify equals false double", vald1, vald2);
```

Float:

```
float valf1 = 1;
float valf2 = 1;
verifyEquals("verify equals true float", valf1, valf2);
valf2 = 2;
verifyEquals("verify equals false float", valf1, valf2);
```

Vector:

```
Vector vec_int = new Vector();
Vector same_vec = new Vector();
Vector vec_int2_length = new Vector();
Vector vec_char = new Vector();
for (int i = 0;i<3;i++)
{
    vec_int.addElement(new Integer(i));
    same_vec.addElement(new Integer(i));
    vec_int2_length.addElement(new Integer(i));
    vec_char.addElement( new Character((char)i));
}
vec_int2_length.addElement(new Integer(500));
Vector another = new Vector();
another.addElement( new Character('a'));
another.addElement( new Character('b'));
verifyLogMessage("Check Vector true and false");
verifyEquals("verify equal vector should be true", vec_int,
same_vec);
verifyEquals("verify equal vector should be false for length",
vec_int, vec_int2_length);
```

```
verifyEquals("verify equal vector should be false, not the same",  
vec_int, vec_char);
```

verifyNotNull()

Purpose

Checks that an object is not null.

Syntax

```
verifyNotNull( [<string>,] <object> )
```

where:

- *<string>* is an optional message
- *<object>* is a Java object

Description

The corresponding test result is *Passed* if *<object>* is not null, and *Failed* if the condition is *False*.

An optional *<string>* message can be logged and displayed in the test report.

If an exception is thrown in a **verifyNotNull()** method, an error is logged and the test continues.

Example

```
Object one_obj = null;  
verifyNotNull("verify not null failed", one_obj);
```


verifyNull()

Purpose

Checks that an object is not null.

Syntax

```
verifyNull( [<string>,] <object> )
```

where:

- *<string>* is an optional message
- *<object>* is a Java object

Description

The corresponding test result is *Passed* if *<object>* is null, and *Failed* if the condition is *False*.

An optional *<string>* message can be logged and displayed in the test report.

If an exception is thrown in a **verifyNull()** method, an error is logged and the test continues.

Example

```
Object one_obj = null;  
verifyNull("verify null passed", one_obj);
```

verifySame()

Purpose

Checks that two object are actually the same.

Syntax

```
verifySame( [<string>], <object1>, <object2> )
```

where:

- *<string>* is an optional message
- *<object1>* and *<object2>* are two Java objects

Description

The corresponding test result is *Passed* if *<object1>* and *<object2>* refer to the same object, and *Failed* if the condition is *False*.

An optional *<string>* message can be logged and displayed in the test report.

If an exception is thrown in a **verifySame()** method, an error is logged and the test continues.

Examples

```
Object one_obj = new Integer(10);  
Object another = one_obj;  
verifySame("verify same passed", another, one_obj);
```

```
Object one_obj = new Long(10);  
Object other=one_obj;  
assertSame(one_obj, other);
```

verifyTrue()

Purpose

Checks that a condition is true.

Syntax

```
verifyTrue( [<string>, ]<Boolean> )
```

where:

- *<string>* is an optional message
- *<Boolean>* is a Boolean condition

Description

The corresponding test result is *Passed* if the **verifyTrue()** condition is *True* and *Failed* if the condition is *False*.

An optional *<string>* message can be logged and displayed in the test report.

If an exception is thrown in a **verifyTrue()** method, an error is logged and the test continues.

Examples

```
verifyTrue("verify 1 true", true);  
verifyTrue("verify 2 false", false);  
verifyTrue(true);  
verifyTrue(false);
```

verify()

Purpose

Checks that an exception is thrown.

Syntax

```
verify( <exception> )
```

where:

- <exception> is a throwable exception

Description

Verifies that an exception compatible with the specified <exception> is thrown during the test.

The corresponding test result is *Passed* if the exception is thrown or *Failed* if not.

If an exception is thrown in a **verify()** method, an error is logged and the test continues.

Examples

```
public void WillThrowRTE()
{
    verifyLogMessage("RTE in next call");
    throw new RuntimeException("Exception Message");
}
public void testException1()
{
    verifyLogMessage("Check true for RTE");
    Throwable tosee= new RuntimeException("Exception Runtime ");
    verify(tosee);
    WillThrowRTE();
}
public void testException2()
```

```

    {
        verifyLogMessage("Check true for RTE");
        Throwable tosee= new Exception("Exception");
        verify(tosee);
        WillThrowRTE();
    }
public void testException3()
{
    verifyLogMessage("Check true for RTE");
    Throwable tosee= new Throwable();
    verify(tosee);
    WillThrowRTE();
}
public void testException4()
{
    verifyLogMessage("Check false for RTE");
    Throwable tosee= new ArithmeticException("Throwable");
    verify(tosee);
    WillThrowRTE();
}
public void testException5()
{
    verifyLogMessage("Check false for RTE");
    Throwable tosee= new ClassCastException("Throwable");
    verify(tosee);
    WillThrowRTE();
}
public void testException6()
{
    verifyLogMessage("Check false for RTE");
    Throwable tosee= new ClassCastException("Throwable");
    verify(tosee);
    //Do not call anything.
}

```

verifyApproxEquals()

Purpose

Checks that two variables have the same value within a given margin.

Syntax

```
verifyApproxEquals( [<string>, ] <variable1>, <variable2>,  
<precision> )
```

where:

- *<string>* is an optional message
- *<variable1>* and *<variable2>* are two Java type variables
- *<precision>* is a precision argument

Description

The corresponding test result is *Passed* if the **assertApproxEquals()** condition is *True* and *Failed* if the condition is *False*.

Compared variables may be of any Java type: *Boolean*, *float*, *double*, *short*, *byte*, *char*, *int*, *long* or *object*. *<variable1>* and *<variable2>* must be of the same type.

An optional *<string>* message can be logged and displayed in the test report.

The *<precision>* argument specifies an acceptable margin.

If an exception is thrown in a **verifyApproxEquals()** method, an error is logged and the test continues.

Example

Char:

```
char valc1 = 'a';  
char valc2 = 'e';  
verifyApproxEquals("verify approx char passed", valc1, valc2, 10);
```

Byte:

```
byte valb1 = 5;  
byte valb2 = 10;  
verifyApproxEquals("verify approx byte passed", valb1, valb2, 10);
```

Short:

```
short vall1 = 1;  
short val2 = 5;  
verifyApproxEquals("verify approx short passed", vall1, val2, 10);
```

Int:

```
int vali1 = 1;  
int vali2 = 20;  
verifyApproxEquals("verify approx int passed", vali1, vali2, 20);
```

Long:

```
long vall1 = 1;  
long vall2 = 20;  
verifyApproxEquals("verify approx long passed", vall1, vall2, 20);
```

Float:

```
float valf1 = 1;  
float valf2 = 20;  
verifyApproxEquals("verify approx float passed", valf1, valf2, 20);
```

Double:

```
double vald1 = 1;  
double vald2 = 20;  
verifyApproxEquals("verify approx double passed", vald1, vald2,  
20);
```

verifyGreaterThan()

Purpose

Checks that a tested value is strictly greater than a reference value.

Syntax

```
verifyGreaterThan( [<string>,] <reference_value>, <tested_value> )
```

where:

- *<string>* is an optional message
- *<reference_value>* and *<tested_value>* are two Java type values

Description

The corresponding test result is *Passed* if the *<tested_value>* is strictly greater than *<reference_value>*, and *Failed* if the condition is *False*.

Compared values may be of any numeric Java type: *Boolean*, *float*, *double*, *short*, *byte*, *char*, *int* or *long*. *<reference_value>* and *<tested_value>* must be of the same type.

An optional *<string>* message can be logged and displayed in the test report.

If an exception is thrown in a **verifyGreaterThan()** method, an error is logged and the test continues.

Examples

Char:

```
char valc1 = 10;  
char valc2 = 20;  
verifyGreaterThan("verify greater char true", valc1, valc2);
```


Byte:

```
byte valb1 = 10;  
byte valb2 = 20;  
verifyGreaterThan("verify greater byte true", valb1,valb2);
```

Short:

```
short val1 = 10;  
short val2 = 20;  
verifyGreaterThan("verify greater short true", val1,val2);
```

Int:

```
int vali1 = 10;  
int vali2 = 20;  
verifyGreaterThan("verify greater int true", vali1,vali2);
```

Long:

```
long vall1 = 10;  
long vall2 = 20;  
verifyGreaterThan("verify greater long true", vall1,vall2);
```

Float:

```
float valf1 = 10;  
float valf2 = 20;  
verifyGreaterThan("verify greater float true", valf1,valf2);
```

Double:

```
double vald1 = 10;  
double vald2 = 20;  
verifyGreaterThan("verify greater double true", vald1,vald2);
```

verifyGreaterThanEquals()

Purpose

Checks that a tested value is greater or equal to reference value.

Syntax

```
verifyGreaterThanEquals( [<string>,) <reference value>,  
<tested_value> )
```

where:

- *<string>* is an optional message
- *<reference_value>* and *<tested_value>* are two Java type values

Description

The corresponding test result is *Passed* if the *<tested_value>* is greater than *<reference_value>* or equals *<reference_value>*, and *Failed* if the condition is *False*.

Compared values may be of any numeric Java type: *Boolean*, *float*, *double*, *short*, *byte*, *char*, *int* or *long*. *<reference_value>* and *<tested_value>* must be of the same type.

An optional *<string>* message can be logged and displayed in the test report.

If an exception is thrown in a **verifyGreaterThanEquals()** method, an error is logged and the test continues.

Example

Char:

```
char valc1 = 10;  
char valc2 = 20;
```

```
verifyGreaterThanOrEquals("verify greater char true", valc1, valc2);
```

Byte:

```
byte valb1 = 10;  
byte valb2 = 20;  
verifyGreaterThanOrEquals("verify greater byte true", valb1, valb2);
```

Short:

```
short val1 = 10;  
short val2 = 20;  
verifyGreaterThanOrEquals("verify greater short true", val1, val2);
```

Int:

```
int vali1 = 10;  
int vali2 = 20;  
verifyGreaterThanOrEquals("verify greater int true", vali1, vali2);
```

Long:

```
long vall1 = 10;  
long vall2 = 20;  
verifyGreaterThanOrEquals("verify greater long true", vall1, vall2);
```

Float:

```
float valf1 = 10;  
float valf2 = 20;  
verifyGreaterThanOrEquals("verify greater float true", valf1, valf2);
```

Double:

```
double vald1 = 10;  
double vald2 = 20;  
verifyGreaterThanOrEquals("verify greater double true",  
vald1, vald2);
```

verifyLowerThan()

Purpose

Checks that a tested value is strictly lower than a reference value.

Syntax

```
verifyLowerThan( [<string> ,] <reference_value>, <tested_value> )
```

where:

- *<string>* is an optional message
- *<reference_value>* and *<tested_value>* are two Java type values

Description

The corresponding test result is *Passed* if the *<tested_value>* is strictly lower than *<reference_value>*, and *Failed* if the condition is *False*.

Compared values may be of any numeric Java type: *Boolean*, *float*, *double*, *short*, *byte*, *char*, *int* or *long*. *<reference_value>* and *<tested_value>* must be of the same type.

An optional *<string>* message can be logged and displayed in the test report.

If an exception is thrown in a **verifyLowerThan()** method, an error is logged and the test continues.

Examples

Short:

```
short val1 = 21;  
short val2 = 20;  
verifyLowerThan("verify lower true short", val1, val2);
```

Byte:

```
byte valb1 = 21;  
byte valb2 = 20;  
verifyLowerThan("verify lower true byte", valb1, valb2);
```

Char:

```
char valc1 = 21;  
char valc2 = 20;  
verifyLowerThan("verify lower true char", valc1, valc2);
```

Int:

```
int vali1 = 21;  
int vali2 = 20;  
verifyLowerThan("verify lower true int", vali1, vali2);
```

Long:

```
long vall1 = 21;  
long vall2 = 20;  
verifyLowerThan("verify lower long true", vall1, vall2);
```

Float:

```
float valf1 = 21;  
float valf2 = 20;  
verifyLowerThan("verify lower float true", valf1, valf2);
```

Double:

```
double vald1 = 21;  
double vald2 = 20;  
verifyLowerThan("verify lower double true", vald1, vald2);
```

verifyLowerThanEquals()

Purpose

Checks that a tested value is lower or equal to reference value.

Syntax

```
verifyLowerThanEquals( [<string>,) <reference value>,  
<tested_value> )
```

where:

- *<string>* is an optional message
- *<reference_value>* and *<tested_value>* are two Java type values

Description

The corresponding test result is *Passed* if the *<tested_value>* is lower than *<reference_value>* or equals *<reference_value>*, and *Failed* if the condition is *False*.

Compared values may be of any numeric Java type: *Boolean*, *float*, *double*, *short*, *byte*, *char*, *int* or *long*. *<reference_value>* and *<tested_value>* must be of the same type.

An optional *<string>* message can be logged and displayed in the test report.

If an exception is thrown in a **verifyLowerThanEquals()** method, an error is logged and the test continues.

Example

Char:

```
char valc1 = 10;  
char valc2 = 20;
```

```
verifyLowerThanEquals("verify lower char true", valc1,valc2);
```

Byte:

```
byte valb1 = 10;  
byte valb2 = 20;  
verifyLowerThanEquals("verify lower byte true", valb1,valb2);
```

Short:

```
short val1 = 10;  
short val2 = 20;  
verifyLowerThanEquals("verify lower short true", val1,val2);
```

Int:

```
int vali1 = 10;  
int vali2 = 20;  
verifyLowerThanEquals("verify lower int true", vali1,vali2);
```

Long:

```
long vall1 = 10;  
long vall2 = 20;  
verifyLowerThanEquals("verify lower long true", vall1,vall2);
```

Float:

```
float valf1 = 10;  
float valf2 = 20;  
verifyLowerThanEquals("verify lower float true", valf1,valf2);
```

Double:

```
double vald1 = 10;  
double vald2 = 20;  
verifyLowerThanEquals("verify lower double true", vald1,vald2);
```

verifyLogMessage()

Purpose

Logs a message in the test report.

Syntax

```
verifyLogMessage( <string> )
```

where:

- *<string>* is a message

Description

The *<string>* message is logged and displayed in the test report.

If an exception is thrown in a **verifyLogMessage()** method, an error is logged and the test continues.

Example

```
verifyLogMessage("Hello World");$
```


verifyLogfail()

Purpose

Marks a test as *Failed* with a message.

Syntax

```
verifyLogfail( <string> )
```

where:

- <string> is a message

Description

The corresponding test result is *Failed* if the **verifyLogfail()** method is encountered.

The <string> message is logged and displayed in the test report.

If an exception is thrown in a **verifyLogfail()** method, an error is logged and the test continues.

Example

```
verifyLogfail("Log message with fail");$
```

createTimer()

Purpose

Creates a timer.

Syntax

```
createTimer( <string> )
```

where:

- *<string>* is a message

Description

Several timers can be created. When a timer is created, **createTimer()** returns an identification number for the timer (integer) for use with other timer-related functions.

Timers must be started with the **timerStart()** method.

The *<string>* message is logged and displayed in the test report.

If an exception is thrown in a **createTimer()** method, an error is logged and the test continues.

Example

```
int timer1, timer2, timer3;  
int any = 10;  
timer1 = createTimer("first timer created");  
timer2 = createTimer("second timer created");  
timer3 = createTimer("third timer created");
```

timerStart()

Purpose

Starts a timer.

Syntax

```
timerStart( <timer_id>, <string> )
```

where:

- *<timer_id>* is in an integer timer identification number
- *<string>* is a string message

Description

Starts the timer identified as *<timer>* and logs the message provided in *<string>*.

Timers must be created with **createTimer()** before being started. Several timers can run simultaneously.

If an exception is thrown in a **timerStart()** method, an error is logged and the test continues.

Example

```
int timer1, timer2, timer3;
int any = 10;
timer1 = createTimer("first timer created");
timer2 = createTimer("second timer created");
timer3 = createTimer("third timer created");
//then start the timers.
timerStart(timer1, "Start 1");
timerStart(timer2, "Start 2");
timerStart(timer3, "Start 3");
```

timerReportElapsedTime()

Purpose

Logs the elapsed time since a timer was started.

Syntax

```
timerReportElapsedTime( <timer_id>, <string> )
```

where:

- <timer_id> is an integer timer identification number
- <string> is a message to be logged with the elapsed time value

Description

The elapsed time of the timer identified as <timer_id> is logged and displayed in the test report with a <string> message.

The time unit is specified by the current Target Deployment Port. By default, the elapsed time is returned in milliseconds.

Timers must be started with **timerReportElapsedTime()** before an elapsed time can be calculated.

If an exception is thrown in a **timerReportElapsedTime()** method, an error is logged and the test continues.

Example

```
int timer1, timer2, timer3;  
int any = 10;  
timer1 = createTimer("first timer created");  
timer2 = createTimer("second timer created");
```

```
timer3 = createTimer("third timer created");
//then start the timers.
timerStart(timer1,"Start 1");
timerStart(timer2,"Start 2");
timerStart(timer3,"Start 3");
//Report time.
long val1, val2, val3;
val2 = 100;
val3 = 1000;
val1 = 10000;
verifyLogMessage("Timer report");
timerReportElapsedTime(timer1,"report 1 ");
timerReportElapsedTime(timer2,"report 2 ");
timerReportElapsedTime(timer2,"report 2 ");
```

verifyElapsedTime()

Purpose

Checks that the elapsed time for a timer is below a given value.

Syntax

```
verifyElapsedTime( <timer_id>, <value>, <string> )
```

where:

- *<timer_id>* is an *int* timer identification number
- *<value>* is a *long* expected time
- *<string>* is a message to be logged with the elapsed time value

Description

The corresponding test result is *Passed* if the elapsed time of the timer identified as *<timer_id>* is lower or equal to the expected time *<value>*, and *Failed* if the condition is *False*.

The result is logged and displayed in the test report with a *<string>* message.

The time unit is specified by the current Target Deployment Port. By default, the elapsed time is returned in milliseconds.

Timers must be started with **timerStart()** before an elapsed time can be calculated.

If an exception is thrown in a **verifyElapsedTime()** method, an error is logged and the test continues.

Example

```
int timer1, timer2, timer3;
int any = 10;
timer1 = createTimer("first timer created");
timer2 = createTimer("second timer created");
timer3 = createTimer("third timer created");
//then start the timers.
timerStart(timer1,"Start 1");
timerStart(timer2,"Start 2");
timerStart(timer3,"Start 3");
//Report time.
long val1, val2, val3;
val2 = 100;
val3 = 1000;
val1 = 10000;
verifyLogMessage("Timer report");
timerReportElapsedTime(timer1,"report 1 ");
timerReportElapsedTime(timer2,"report 2 ");
timerReportElapsedTime(timer2,"report 2 ");
//then some verifys.
verifyLogMessage("Timer verifys");
verifyElapsedTime(timer1,val1,"ellapsed 1 with 10000");
verifyElapsedTime(timer1,val2,"ellapsed 1 with 100");
verifyElapsedTime(timer1,val3,"ellapsed 1 with 1000");
```


System Testing Script Languages

4

System Testing Language for C

Purpose

The Test RealTime System Testing feature uses two different script languages and C macros that can be added to your C code:

- System Testing Test Script Language (**.pts** scripts)
- System Testing Supervisor Script Language (**.spv** scripts)
- System Testing Probe Macros

Note When using the Test RealTime graphical user interface, the **.spv** supervisor scripts are generated automatically. Experienced users can edit these files manually. See **Editing Supervisor Scripts** in the **Test RealTime User Guide**.

This section describes each System Testing Language scripting instruction, including:

- Syntax
- Functionality and rules governing its usage
- Examples of use

Notation Conventions

Throughout this guide, command notation and argument parameters use the following standard convention:

Notation	Example	Meaning
BOLD	ADD_ID	Language keyword
<i><italic></i>	<i><filename></i>	Symbolic variables
[]	[<i><option></i>]	Optional items
{ }	{ <i><filenames></i> }	Series of values
[{ }]	[{ <i><filenames></i> }]	Optional series of variables
	on off	OR operator

Note: PTS and SPV keywords are case sensitive. Keywords must be typed in upper case as specified in this guide.

System Testing Language Identifiers

A System Testing Language identifier for C is a text string used as a label, such as the name of a message type in an **.spv** supervisor script or **.pts** test script.

Identifiers are made of an unlimited sequence of the following characters:

- a-z
- A-Z
- 0-9
- _ (underscore)

Spaces are not valid identifier characters.

Unlike keywords, System Testing Language identifiers are case sensitive. This means that **LABEL**, **label**, and **Label** are three different identifiers.

Test Script Keywords

Test Script Keywords (.pts Test Script)

System Testing Script keywords are not case sensitive. This means that **PRINT**, **print**, and **Print** are equivalent. For conventional purposes, however, this document uses upper case notation for System Testing Test Script keywords in order to differentiate from native source code.

- ADD_ID
- CALL
- CALLBACK ... END CALLBACK
- CASE ... IS ... WHEN ... WHEN OTHERS ... END CASE
- CATCH
- CHANNEL
- CLEAR_ID
- COMMENT
- COMMTYPE
- DECLARE_INSTANCE
- DEF_MESSAGE
- END
- ERROR
- EXCEPTION ... END EXCEPTION
- EXIT
- FAMILY
- FLUSH_TRACE
- FORMAT

- HEADER
- IF...THEN...ELSE ... END IF
- INCLUDE
- INITIALIZATION ... END INITIALIZATION
- INSTANCE ... END INSTANCE
- MATCHED()
- MATCHING()
- MESSAGE
- MESSAGE_DATE
- NIL
- NO_MESSAGE
- NONIL
- NOTMATCHED()
- NOTMATCHING()
- PAUSE
- PRINT
- PROC ... END PROC
- PROCSEND ... END PROCSEND
- RENDEZVOUS
- RESET
- SCENARIO ... LOOP ... END SCENARIO
- SEND
- SHARE
- TERMINATION ... END TERMINATION

- TIME
- TIMER
- TRACE_OFF
- TRACE_ON
- VAR
- VIRTUAL CALLBACK
- VIRTUAL PROCSEND
- WAITTIL
- WHILE ... END WHILE
- WTIME

ADD_ID

Syntax

```
ADD_ID(<channel_identifier>, <connection_identifier>)
```

Description

The **ADD_ID** instruction dynamically adds the value of a connection identifier to a communication channel identifier.

A communication channel is a logical medium that integrates (multiplexes) the same type of connection between the virtual tester and remote applications under test.

When opening a connection with your communication API, you must dynamically link the connection identifier with a channel identifier.

You must declare a channel identifier with the **CHANNEL** instruction.

C connection identifiers must be compatible with C communication channels.

Examples

```
...
COMMTYPE ux_inet IS integer_t
CHANNEL ux_inet: ch
...
SCENARIO First
...
#integer_t id;
CALL socket(AF_UNIX, SOCK_STREAM, 0) @@ id
ADD_ID(ch, id)
....
```

CALL

Syntax

```
CALL <identifier> ( <arguments> ) [ @ [<expected_expr>] @  
[<return_var>] ]
```

Description

The **CALL** instruction lets you call a specific interface routine. This routine may be a function or a procedure.

You can check a function's return values for interface routine calls.

The @ character is a separator.

<expected_expr> gives the expected return value of the function.

<return_var> gives the variable in which the return value of the function is stored.

If <return_var> is specified, the return value is stored in <return_var>.

The **CALL** instruction can be used in a **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION**, or **EXCEPTION** blocks.

Example

```
#int return_val;  
#int V_in;  
#int V1_out, V2_out;  
SCENARIO TEST_1  
FAMILY nominal  
...  
CALL API_function(V_in,  
REF(@0@V1_out), &1@0@V2_out)@1@return_val  
...
```

CALLBACK ... END CALLBACK

Purpose

The **CALLBACK** instruction dynamically recalls message reception and links a connection identifier value to a communication channel identifier.

Syntax

```
CALLBACK <message_type>: <msg> ON <commtype>: <id> [ <n> ]  
END CALLBACK
```

<message_type> is a message type previously declared in a **MESSAGE** statement.

<msg> is the output parameter of *<message_type>* that must be initialized in the callback if a message is received.

<commtype> is the type of communication used for reading messages previously declared in a **COMMTYPE** statement.

<id> is the input connection parameter on which a message must be read.

Description

Callbacks are declared in the first part of the test script, before the first scenario.

<commtype> must be declared with the **COMMTYPE** instruction.

<message_type> must be declared with the **MESSAGE** instruction.

You can declare only one callback per combination of message and communication type.

Message reception in the **CALLBACK** statement must never be blocked. If no message is received, you must exit the block using the **NO_MESSAGE** instruction.

Use of both a **NO_MESSAGE** and **MESSAGE_DATE** statement is mandatory within the callback or a procedure called from a callback.

If the C *<message_type>* contains *unions*, you can define for each union the display and comparison field. The system implicitly defines a structured variable, named as **ATL_** followed by the name of the *<message_type>*. You can specify which field to use by specifying *select* attribute for the union.

Freeze Mode

Freeze mode is a blocking mode in which the **CALLBACK** waits for a message to be received. To use *freeze mode*, you must use only one **CALLBACK** block throughout the entire test script, messages can be read in *freeze mode*. In this mode, the **ATL_TIMEOUT** macro specifies the maximum wait delay for a message. The value of **ATL_TIMEOUT** is calculated from a **WTIME** expression used in the **WAITTIL** statement. The **ATL_TIMEOUT** macro is an integer and uses the time unit defined in the Target Deployment Port. By default, the time unit is a hundredth of second.

Example

```
typedef enum { e_name, e_id, e_balance } client_kind_t;
typedef struct {
  client_kind_t kind;
  union {
    char name[50];
    int id;
    float balance;
  } my_union;
} client_info_t;
COMMTYPE socket IS socket_id_t
CHANNEL socket: ch
MESSAGE client_info_t: msg
CALLBACK client_info_t: info ON socket: id
      CALL read(id, &info, sizeof(client_info_t))@@ret
```

```
IF (ret == 0) THEN
    NO_MESSAGE
END IF
MESSAGE_DATE
VAR ATL_client_info_t.my_union.select, INIT=info.kind
END CALLBACK
```

CASE ... IS ... WHEN ... WHEN OTHERS ... END CASE

Syntax

```
CASE <expression> IS
WHEN <constant1> => <instructions>
WHEN <constant2> => <instructions>
WHEN <constant3> => <instructions>
WHEN OTHERS => <instructions>
END CASE
```

Description

The **CASE** instruction allows you to choose one of several sets of instructions according to the value of an expression.

The **CASE** instruction may appear in a **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION** or **EXCEPTION** block.

The list of options for the expression begins after **IS** and ends in **END CASE**.

WHEN identifies the different constant expressions that cause a specific process to be carried out. This process is defined by the instructions following the => symbol.

OTHERS processes all the values of expression that have not been explicitly processed in the **CASE**. This instruction set is optional.

<expression> must take an integer value.

Examples

```
##define ACK 0
##define NACK 1
#int choice;
SCENARIO TEST_1
FAMILY nominal
CALL ApiGetChoice(choice)
CASE (choice) IS
WHEN ACK => CALL ApiAcknowledge()
WHEN NACK => CALL ApiReset()
...
WHEN OTHERS => CALL Api_DefaultMsg()
END CASE
...
```

CHANNEL

Syntax

```
CHANNEL <communication_type>: <channel> {[, <channel> ]}
```

Description

The **CHANNEL** instruction allows you to declare a set of communication channels.

You must declare the *<communication_type>* with the **COMMTYPE** instruction.

Each *<channel>* variable identifies a new type of communication channel. A communication channel is a logical medium that integrates (multiplexes) the same type of connection among virtual testers and remote applications under test.

Use the **CHANNEL** instruction at the beginning of the test script, before the first scenario.

Examples

```
#typedef int inet_id_t;
COMMTYPE ux_inet IS inet_id_t WITH MULTIPLEXING
CHANNEL ux_inet: ch_1, ch_2, ch_3
CHANNEL ux_inet: ch_out
```

CLEAR_ID

Syntax

```
CLEAR_ID ( <channel_identifier> )
```

Description

The **CLEAR_ID** instruction clears a communication channel.

The communication channel has no more links with remote applications under test.

You must declare a communication channel with the **CHANNEL** instruction.

Example

```
...
COMMTYPE ux_inet IS integer_t
CHANNEL ux_inet: ch
...
SCENARIO First
...
#integer_t id;
CALL socket(AF_UNIX, SOCK_STREAM, 0) @@ id
ADD_ID(id,ch)
...
CLEAR_ID(ch)
....
```

COMMENT

Syntax

COMMENT

Description

The **COMMENT** instruction allows you to add comments to the results file by inserting text.

Its use in test scenarios is optional.

The position of the **COMMENT** instruction in the test program defines the position in which the comment appears in the test report.

The **COMMENT** instruction may appear in a **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION** or **EXCEPTION** block.

In the command line interface, you can deactivate the processing of comments by adding the **-NOCOMMENT** option to the C Test Script Compiler command line.

Example

```
SCENARIO TEST_1
FAMILY nominal
COMMENT calling connection confirmation
CALL api_trsprt_connectionCF()
...
```

COMMTYPE

Syntax

```
COMMTYPE <identifier> IS <connection_id_type> [WITH MULTIPLEXING]
```

Description

The **COMMTYPE** instruction defines a type of communication. The C connection identifies the communication type.

The C <connection_id_type> must be a *typedef*, as defined in the interface file, an included file, or in the test script.

You can define the communication type as being able to multiplex connections for the read operation, using the multiplexing option.

You must use the **COMMTYPE** instruction at the beginning of the test script, before the first scenario.

Example

```
...
typedef int inet_id_t;
COMMTYPE ux_inet IS inet_id_t WITH MULTIPLEXING
typedef struct { int key; int id; } msgqueue_id_t;
COMMTYPE ux_msgqueue IS msgqueue_id_t
....
```


DECLARE_INSTANCE

Syntax

```
DECLARE_INSTANCE <instance> { [, <instance> ] }
```

Description

The **DECLARE_INSTANCE** instruction allows you to define the set of the instances described in the test script.

A **DECLARE_INSTANCE** instruction takes effect after you have declared it.

<instance> may be any identifier. The **DECLARE_INSTANCE** must have at least one instance name passed by parameter.

Example

```
HEADER "DEMO SOCKET", "1.0", "2.4"  
DECLARE_INSTANCE client, server  
SCENARIO Principal  
...  
END SCENARIO
```

DEF_MESSAGE

Syntax

```
DEF_MESSAGE <message>, EV= <cmp_expression>
```

Description

The **DEF_MESSAGE** instruction allows you to define a reference *<message>* variable. In order to do this, you must define the reference values with *<cmp_expression>*.

The message variable is the reference event variable initialized by the **DEF_MESSAGE** instruction. It has to be declared by the **MESSAGE** instruction.

Associated Rules

The **DEF_MESSAGE** instruction can appear in a **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION**, or **EXCEPTION** block.

You may partially define a reference message. The undefined *<cmp_expression>* fields are not used to compare incoming messages.

Interface File

```
typedef struct {
    int type;
    struct {
        char app_name[8];
        unsigned char class_name;
    } data;
    char userdata[100];
} message_t;
```

Example

```
MESSAGE message_t: msg
SCENARIO first
```

```
DEF_MESSAGE msg, EV= { code=>ConnectCF,  
&                      data=>{ app_name=>"ATCMKD" }}
```

END

Syntax

```
END <block>
```

Description

The **END** instruction delimits an instruction block.

You use it to end the following:

- A callback: **END CALLBACK**
- A procedure: **END PROC**
- A message sending procedure: **END PROCSEND**
- An initialization block: **END INITIALIZATION**
- A termination block: **END TERMINATION**
- An exception block: **END EXCEPTION**
- A scenario: **END SCENARIO**
- An instance block: **END INSTANCE**
- A **CASE** instruction: **END CASE**
- An **IF** instruction: **END IF**
- A **WHILE** instruction: **END WHILE**

Example

```
INSTANCE tester1, tester2:  
PROC clean1  
...  
END PROC  
...  
END INSTANCE  
INITIALIZATION  
...
```

```
END INITIALIZATION  
SCENARIO TEST1  
...  
END SCENARIO
```

ERROR

Syntax

ERROR

Description

When an unexpected output value for a function or a **WAITTIL** causes a problem, the current scenario halts as a result. You may terminate the scenario deliberately with the **ERROR** instruction.

After an **ERROR** instruction, the **EXCEPTION** block is executed on the next scenario at the same level, if there is one.

Example

```
#int sock;
...
SCENARIO Main
SCENARIO Test1
...
IF (sock==-1) THEN
ERROR
END IF
...
END SCENARIO
SCENARIO Test2
...
CALL ...
...
END SCENARIO
END SCENARIO
```

In the above example, you can stop the *Test1* scenario with the **ERROR** instruction. The virtual tester then proceeds to *Test2* scenario.

EXCEPTION ... END EXCEPTION

Syntax

```
EXCEPTION [ <proc>( [ <arg> { [, <arg>] } ] ] ]  
END EXCEPTION
```

Description

The **EXCEPTION** instruction or block deletes a specific environment by executing the set of instructions or the procedure *<proc>*. **END EXCEPTION** marks the end of the **EXCEPTION** block.

Associated Rules

An **EXCEPTION** block or instruction applies to the set of scenarios at its level.

It does not apply to subscenarios of these scenarios.

The **EXCEPTION** instruction or block is optional.

A maximum of one **EXCEPTION** block may occur in a scenario.

The **EXCEPTION** instruction is only executed if a scenario terminates with an error.

It does not matter where the **EXCEPTION** instruction is placed among scenarios in a given level.

Example

```
#int sock;  
EXCEPTION  
CALL close (sock)
```

```
...  
END EXCEPTION  
...  
SCENARIO Main  
...  
END SCENARIO
```


EXIT

Syntax

```
EXIT
```

Description

This instruction lets you exit from the virtual tester. It causes all scenarios to terminate.

After an **EXIT**, the virtual tester terminates. For an **EXIT** instruction, the end of execution code of the virtual tester process is -1.

The scenario in which the **EXIT** instruction was executed is deemed incorrect.

Example

```
#int sock;
...
SCENARIO Main
SCENARIO Test1
...
IF (sock==-1) THEN
COMMENT stop tester
EXIT
END IF
...
END SCENARIO
SCENARIO Test2
...
CALL ...
...
END SCENARIO
END SCENARIO
```

FAMILY

Syntax

```
FAMILY <family> {[, <family> ]}
```

Description

The **FAMILY** instruction allows you to group tests by families or classes.

This instruction appears just once at the beginning of a **SCENARIO** block, where it defines the family or families to which the scenario belongs.

When starting tests, you can request to execute only tests of a given family.

The <family> parameter indicates the name of the test family. You can define the following families: nominal, structural, robustness.

A test can belong to several families: in this case, the **FAMILY** instruction contains a <family> list, separated by commas.

<family> can be any identifier. You must have at least one family name.

The **FAMILY** instruction is optional. If omitted, the test belongs to every family.

Example

```
SCENARIO Test_1
FAMILY nominal
COMMENT ...
...
END SCENARIO
```

FLUSH_TRACE

Syntax

```
FLUSH_TRACE
```

Description

The **FLUSH_TRACE** instruction dumps the execution traces stored in the circular buffer to the **.rio** file.

This instruction is taken into account only when the **-TRACE=CIRCULAR** test compiler option is set.

The **FLUSH_TRACE** instruction can be used in a **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION**, or **EXCEPTION** block. You may not use **FLUSH_TRACE** in a **CALLBACK** or **PROCSEND** block.

Example

```
SCENARIO one
  (...)
  FLUSH_TRACE
  (...)
END SCENARIO
```

FORMAT

Syntax

```
FORMAT <variable> = <format>
FORMAT <type> = <format>
FORMAT <field> = <format>
```

Description

This **FORMAT** instruction modifies the way a variable, type, or field of a structure is tested and printed. All formats of the same type is modified.

The new format is defined in C.

A format can also specify a print mode in binary or hexadecimal, using the options **#B** and **#H**.

The **FORMAT** instruction is optional. You may use it at the beginning of the test script or in a block of instructions, depending on the required scope. However, **FORMAT** statements that apply to data contained in a **CALLBACK** or **PROCSSEND** block must be located before:

- any **CALLBACK** or **PROCSSEND** block
- any **PROC** statements that contain **DEF_MESSAGE** or **SEND** instructions

Example

```
SCENARIO first
#char buffer[100];
#typedef struct {
#   int ax_register;
#   int bx_register;
#   int cx_register;
#} 8088_register_t;
FORMAT buffer = unsigned char[50]
```

```
FORMAT 8088_register_t.ax_register = #B
FORMAT 8088_register_t.bx_register = #H
END SCENARIO
```

HEADER

Syntax

```
HEADER <test_name>, <version>, <test_plan_version>
```

Description

This instruction allows you to define a standard header at the beginning of the test script. The information contained in this header enables you to identify a list of scenarios.

The headers can be strings or environment variables.

<*test_name*> is the name for the test script.

<*version*> is the version of the system tested.

<*test_plan_version*> is the test script version.

This instruction must appear before the first instruction block and strings must be enclosed in double-quotes (" ").

Example

```
HEADER "DEMO SOCKET", $VERSION, "2.4"  
INITIALIZATION  
...  
END INITIALIZATION  
SCENARIO Main  
...  
END SCENARIO
```

IF...THEN...ELSE

Syntax

```
IF <condition> THEN
ELSE
END IF
```

Description

This is a control statement. The simplest form of an **IF** instruction begins with the keyword **IF**, is followed by a Boolean expression, and then the keyword **THEN**. A set of instructions follows. These instructions are executed if the expression is true. The last **END IF** marks the end of the set of instructions.

Other actions can be executed depending on the value of the condition. Add an **ELSE** block, followed by the set of instructions to be executed if the condition is false.

IF may be placed anywhere in the test program.

THEN must be placed at the end of a line.

ELSE must be on its own line.

END IF must be on its own line.

Example

```
HEADER "DEMO SOCKET RPC", "1.0a", "2.5"
#int sock;
INITIALIZATION
...
IF (sock==-1) THEN
ERROR
```

```
ELSE
CALL listen(sock,5)
...
END IF
...
END INITIALIZATION
SCENARIO Main
.....
```


INCLUDE

Syntax

```
INCLUDE <string>
```

Description

The **INCLUDE** instruction lets you include scenarios in the current test script.

Its use in test scenarios is optional.

The **INCLUDE** instruction may appear in any scenario as long as the scenario does not contain any primary instructions. <string> is the name of the file to be included. The system searches for files in the current directory and then searches the list of paths passed on to the Test Script Compiler.

Example

```
SCENARIO Test_1
FAMILY nominal
INCLUDE "../common/initialization"
INCLUDE "scenario_1_and_2"
SCENARIO scenario_3
COMMENT call connection
CALL api_trsprt_connexionCF()
CALL ...
END SCENARIO
END SCENARIO
```

INITIALIZATION ... END INITIALIZATION

Syntax

```
INITIALIZATION [ <proc> ( [ <arg> { , <arg> } ] ) ]  
END INITIALIZATION
```

Description

The **INITIALIZATION** instruction initializes a specific environment by executing a set of instructions or the procedure *<proc>*. **END INITIALIZATION** marks the end of the **INITIALIZATION** block.

An **INITIALIZATION** block or instruction applies to the set of scenarios at its level. It does not apply to sub-scenarios.

The **INITIALIZATION** instruction or block is optional.

A maximum of one **INITIALIZATION** block or instruction may occur at a given scenario level.

This instruction is executed before every scenario at the same level.

The **INITIALIZATION** instruction may appear anywhere among scenarios at a given level.

Example

```
...  
INITIALIZATION  
CALL  socket (AF_INET, SOCK_DGRAM, 0)@ds  
...  
FD_ADD(ds,SOCKAPI)  
END INITIALIZATION  
...
```

INSTANCE ... END INSTANCE

Syntax

```
INSTANCE <instance>{ [, <instance>] }:  
END INSTANCE
```

Description

An **INSTANCE ... END INSTANCE** block allows you to specify associated declarations or the instructions.

When the **INSTANCE ... END INSTANCE** block is located before the top-level scenarios, it gives global declarations to the test script for all the specified instances.

At the block or nested scenario level, it gives instructions or local declarations to the wrapping block or scenario.

You may not nest instance blocks.

You cannot mix declarations and instructions in the same instance block.

Instance blocks containing instructions follow instance blocks containing declarations.

Examples

```
HEADER "DEMO SOCKET", $VERSION, "2.4"  
DECLARE_INSTANCE client, server  
INSTANCE server:  
    #static int var_c_time ;  
END INSTANCE  
INITIALIZATION  
    INSTANCE server:  
        var_c_time = 0;  
    END INSTANCE  
END INITIALIZATION
```

```
SCENARIO Principal
...
INSTANCE client:
    #int connectStatus ;
END INSTANCE
...
INSTANCE server:
    var_c_time = TIME(globalTime);
END INSTANCE
END SCENARIO
```

INTERRECV

Syntax

```
INTERRECV (<integer_pointer>)  
INTERRECV (<string_pointer>, <buffer size>)
```

<integer_pointer> indicates the memory location of a 32-bit integer message.

<string_pointer> points to a static or allocated memory zone containing the incoming message.

<buffer size> is the size of the memory zone starting at <string_pointer>.

Description

The **INTERRECV** statement allows the virtual tester to receive a simple message sent by an **INTERSEND** statement from another virtual tester.

Received messages are stored in static or allocated memory zone indicated by <integer_pointer> or <string_pointer>.

The message can be either an integer or a string. However if the message type expected by the **INTERRECV** mismatches the actual message type sent by **INTERSEND**, System Testing for C attempts to convert the message.

Example

```
INSTANCE JUPITER:  
    INTERSEND( "How many messages did you receive from  
SUT?" , "SATURN_0" )  
    INTERRECV( &transmitted_int)  
END INSTANCE  
  
INSTANCE SATURN:  
    INTERRECV( buffer, 1024 )
```

```
INTERSEND( 2 , "JUPITER_0" )  
END INSTANCE
```

INTERSEND

Syntax

```
INTERSEND (<integer>, <identifier>)
```

```
INTERSEND (<string>, <identifier>)
```

<identifier> is the unique identifier of a virtual tester to which the message is to be sent.

<integer> is a 32-bit integer value.

<string> is a string-type value.

Description

The **INTERSEND** statement allows the virtual tester to send a simple message to another virtual tester. The other virtual tester receives the incoming message with the **INTERRECV** statement.

The message can be either an integer or a string.

<identifier> is <instance_name>_<occid> or <test_script.rio>_<occid>

The default value for <occid> is 0.

Example

```
INSTANCE JUPITER:
    INTERSEND( "How many messages did you receive from
SUT?" , "SATURN_0" )
    INTERRECV( &transmitted_int)
END INSTANCE
```

```
INSTANCE SATURN:
    INTERRECV( buffer, 1024 )
    INTERSEND( 2 , "JUPITER_0" )
```

END INSTANCE

MATCHED

Syntax

```
MATCHED ( <ref_msg> { [, <channel> ] } )
```

Description

<ref_msg> is a reference message variable declared with the **MESSAGE** instruction and initialized with the **DEF_MESSAGE** instruction.

<channel> is a communication channel declared with the **CHANNEL** instruction and initialized by the **ADD_ID** instruction.

MATCHED is a function that returns a Boolean value. It returns true if one of the messages received during a **WAITTIL** matches the reference message <ref_msg>. If you specify a channel, it returns true only if the matching message was received on this channel.

It returns true if at least one received message has the same values as those defined for the reference message.

MATCHED is only meaningful when used in a **WAITTIL** instruction or in control statements following a **WAITTIL**, such as **IF**, **WHILE**, or **CASE**.

The **MATCHED** return value changes when you reuse it in a **WAITTIL** statement.

Examples

```
...
CHANNEL ux_socket: ch
SCENARIO Main
DEF_MESSAGE msg_1, EV={100,10}
DEF_MESSAGE msg_2, EV={200,20}
...
WAITTIL (MATCHED(msg_1) && MATCHED(msg_2,ch), WTIME==10)
```

```
...  
IF (MATCHED(msg_1,ch)) THEN  
...
```

MATCHING

Syntax

```
MATCHING( <ref_msg> { [, <channel> ] } )
```

Description

MATCHING is a function that returns a Boolean value. It returns true if the last message received during a **WAITTIL** matches the reference message *<ref_msg>*. If you specify a channel, it returns true only if the matching message was received on this channel.

<ref_msg> is a reference message variable declared with the **MESSAGE** instruction and initialized with the **DEF_MESSAGE** instruction.

<channel> is a communication channel declared with the **CHANNEL** instruction and initialized by the **ADD_ID** instruction.

It returns true if the last received message has the same values as those defined for the reference message.

Associated Rules

MATCHING is only meaningful when used in a **WAITTIL** instruction and in control statements following a **WAITTIL**, such as **IF**, **WHILE**, or **CASE**.

The **MATCHING** return value changes when you reuse it in a **WAITTIL**.

Examples

```
...
CHANNEL ux_socket: ch
SCENARIO Main
DEF_MESSAGE msg_1, EV={100,10}
DEF_MESSAGE msg_2, EV={200,20}
...
```

```
WAITTIL(MATCHING(msg_1) || MATCHING(msg_2,ch),WTIME==10)
...
IF (MATCHING(msg_1,ch)) THEN
...
```

MESSAGE

Syntax

```
MESSAGE <message_type>: <ref_msg> {[, <ref_msg>]}
```

Description

The MESSAGE instruction allows you to declare a list of reference messages <ref_msg> of the <message_type> type.

<message_type> is in C and must be defined by a **typedef** in the interface file, an included file, or the test script.

You must use the **MESSAGE** instruction at the beginning of the test script, before the first scenario.

The reference messages are global variables. After a **WAITTIL** instruction, the reference messages used contains the value of the last received message.

Interface file

```
typedef struct {
    int code;
    int flight_number;
    struct {
        char flight_name[8];
        unsigned char class_name;
    } data;
} aircraft_data_t;
```

Examples

```
MESSAGE aircraft_data_t: air_msg
SCENARIO first
DEF MESSAGE air_msg, EV= {code => FlightReport }
WAITTIL(MATCHING(air_msg), WTIME == 100)
...
IF (air_msg.flight_number == 321) THEN
...
```

MESSAGE_DATE

Syntax

MESSAGE_DATE

Description

The **MESSAGE_DATE** instruction marks the date the user receives the message.

For instance, this date may be the moment a message is present in a reception queue or when a message has been read and decoded. This instruction must appear once in a callback or in a procedure called in a callback.

The **MESSAGE_DATE** instruction must be used in a callback.

Examples

```
COMMTYPE socket IS socket_id_t
CHANNEL socket: ch
MESSAGE client_info_t: msg
CALLBACK client_info_t: info ON socket: id
CALL read(id, &info, sizeof(client_info_t))@ret
IF (ret == 0) THEN
NO MESSAGE
END IF
MESSAGE_DATE
END CALLBACK
```

NIL

Syntax

NIL

Description

NIL is a macro that represents the value of a null pointer and can be used in any C expression.

Example

```
...  
SCENARIO Main  
CALL free_object (@NIL@object)  
...  
END SCENARIO
```

NO_MESSAGE

NO_MESSAGE

Syntax

NO_MESSAGE

Description

The **NO_MESSAGE** instruction is used to exit the callback if no message has been received.

This instruction has to appear once in a callback or in a procedure called in a callback.

The **MESSAGE_DATE** instruction must be used in a callback.

Example

```
COMMTYPE socket IS socket_id_t
CHANNEL socket: ch
MESSAGE client_info_t: msg
CALLBACK client_info_t: info ON socket: id
CALL read(id, &info, sizeof(client_info_t))@@ret
IF (ret == 0) THEN
NO_MESSAGE
END IF
MESSAGE_DATE
END CALLBACK
```


NONIL

Syntax

NONIL

Description

NONIL is a macro that represents the value of a non-null pointer and can be used in any C expression.

NONIL is useful in a **CALL** or a **VAR** instruction. In these two cases, it verifies that the pointer does not have a null value.

Example

```
...
SCENARIO Main
CALL alloc_object( ) @ NONIL @ object
VAR object, VA = NONIL
...
END SCENARIO
```

NOTMATCHED

Syntax

```
NOTMATCHED ( <ref_msg> [, <channel> ] )
```

Description

NOTMATCHED is a function that returns a Boolean value. It returns true if one of the messages received during a **WAITTIL** does not match the reference message *<ref_msg>*. If you specify a channel, it returns true only if the non-matching message was received on this channel.

<ref_msg> is a reference message variable declared with the **MESSAGE** instruction and initialized with the **DEF_MESSAGE** instruction.

<channel> is a communication channel declared with the **CHANNEL** instruction and initialized by the **ADD_ID** instruction.

It returns true if at least one received message has a value different from those defined for the reference message.

NOTMATCHED is only meaningful when used in a **WAITTIL** instruction or in control statements following a **WAITTIL**, such as **IF**, **WHILE**, or **CASE**.

The **NOTMATCHED** return value changes when reused in a **WAITTIL**.

Example

```
...
CHANNEL ux_socket: ch
SCENARIO Main
DEF_MESSAGE msg_1, EV={100,10}
DEF_MESSAGE msg_2, EV={200,20}
...
WAITTIL (WTIME==10, NOTMATCHED(msg_1))
```

```
...  
IF (NOTMATCHED(msg_1,ch)) THEN  
...
```

NOTMATCHING

Syntax

```
NOTMATCHING ( <ref_msg> [, <channel> ] )
```

Description

NOTMATCHING is a function that returns a Boolean value. It returns true if the last message received during a **WAITTIL** does not match the reference message *<ref_msg>*. If you specify a channel, it returns true only if the non-matching message was received on this channel.

<ref_msg> is a reference message variable declared with the **MESSAGE** instruction and initialized with the **DEF_MESSAGE** instruction.

<channel> is a communication channel declared with the **CHANNEL** instruction and initialized by the **ADD_ID** instruction.

It returns true if the value of the last received message differs from the values specified for the reference message.

NOTMATCHING is only meaningful when used in a **WAITTIL** instruction or in control statements following a **WAITTIL**, such as **IF**, **WHILE**, or **CASE**.

The **NOTMATCHING** return value changes when reused in a **WAITTIL**.

Example

```
...
CHANNEL ux_socket: ch
SCENARIO Main
DEF_MESSAGE msg_1, EV={100,10}
DEF_MESSAGE msg_2, EV={200,20}
...
WAITTIL (WTIME==10, NOTMATCHING (msg_2, ch) )
```

```
...  
IF (NOTMATCHING(msg_2, ch)) THEN  
...
```

PAUSE

Syntax

```
PAUSE <numeric_expression>
```

Description

The **PAUSE** instruction lets you temporarily suspend the test script execution. The *<numeric_expression>* parameter defines the duration of the suspend execution.

The **PAUSE** instruction does not appear in generated reports.

Example

```
#int hp = 3;
    #int ds = 5;
    PROC init (int sock_type)
    ...
    PAUSE 20
    ...
    END PROC
    SCENARIO Main
    ...
    CALL init( AF_UNIX )
    PAUSE (hp+ds)*23
    ...
    END SCENARIO
```

PRINT

Syntax

```
PRINT <identifier>, <numeric_expression>
```

Description

The **PRINT** instruction lets you print the value of *<numeric_expression>* in the generated reports. The identifier names the value.

<numeric_expression> must be a numerical C expression.

The same identifier can be used in different **PRINT** instructions.

Example

```
#int hp = 3;
#int ds = 5;
TIMER time
PROC init (int sock_type)
...
PRINT SocketValue, sockType
...
END PROC
SCENARIO Main
...
CALL init( AF_UNIX )
PRINT HpDs, (hp+ds)*10
PRINT elapsedTime, TIME (time)
...
END SCENARIO
```

PROC ... END PROC

Syntax

```
PROC <arg> {[, <arg> ]}  
END PROC
```

Description

The **PROC** instruction lets you define a local procedure inside a scenario. A procedure can take parameters defined as data types.

Any previously defined global variables declared in the test script are visible in the **PROC** block. Variables declared locally to a procedure block are only visible within that procedure.

Procedure parameters take basic data: *int*, *char*, and *float* as well as any data types defined by the a ***typedef*** statement.

Procedures must be located at the beginning of the test script file, before the highest-level scenarios.

Procedures can be called from any scenario.

Procedures do not return any parameters.

Example

```
#int hp,ds;  
PROC init (int sock_type)  
...  
CALL gethostbyname (serv_name)@@hp  
CALL socket (sock_type, SOCK_DGRAM, 0)@@ds  
...  
END PROC  
SCENARIO Main  
...  
CALL init( AF_UNIX )
```


...
END SCENARIO

PROCSEND

Syntax

```
PROCSEND <message_type>: <msg> ON <commtype>: <id>  
END PROCSEND
```

Description

The **PROCSEND** instruction allows you to define a message-sending procedure. The **SEND** statement uses this instruction.

<message_type> is declared with the **MESSAGE** instruction.

<msg> is the input parameter of <message_type> that describes the message to be sent.

<commtype> is the communication method for sending messages.

<id> is the input parameter of the connection on which a message has to be sent.

You must declare the message-sending procedure in the first part of the test script, before the first scenario.

Declare <commtype> with the instruction **COMMTYPE**.

Declare <message_type> with the instruction **MESSAGE**.

You only need to declare one message-sending procedure a message and communication type pair.

If the structured C <message_type> contains unions, you should declare the field of the union that you want to use. For this purpose, a structured

variable is implicitly defined. Its name adds **ATL_** before the name of the *<message_type>*. An attribute selected for each union lets you define the field.

Example

```
typedef enum { e_name, e_id, e_balance } client_kind_t ;
typedef struct {
    client k_kind_t kind ;
    union {
        char name[50];
        int id ;
        float balance ;
    } my_union
} client_info_t;
COMMTYPE socket IS socket_id_t
CHANNEL socket: ch
MESSAGE client_info_t: msg
#socket_id_t id;
PROCSSEND message_t: msg ON appl_comm: id
...
CALL socket (sock_type, SOCK_DGRAM, 0) @ 0
...
END PROCSSEND
SCENARIO Principal
...
    ADD_ID(ch,id)
...
SEND (msg,ch)
...
END SCENARIO
```

RENDEZVOUS

Syntax

```
RENDEZVOUS <identifier>
```

Description

The **RENDEZVOUS** instruction allows you to synchronize several virtual testers. A rendezvous name is the <identifier> following the keyword.

When the scenario is executed, the **RENDEZVOUS** instruction stops the execution until all virtual testers have reached the rendezvous point, thereby validating the rendezvous.

When the rendezvous is valid, the scenario resumes the execution.

A **RENDEZVOUS** identifier does not appear more than one time in a scenario.

Example

```
SCENARIO Connection  
RENDEZVOUS begin  
...
```

RESET

Syntax

```
RESET <identifier>
```

Description

The **RESET** instruction lets you reset the *<identifier>* timer.

Declare the timer identifier with the **TIMER** instruction.

You may use a timer identifier only once in the same block. The timer immediately restarts after being reset.

Example

```
TIMER time  
SCENARIO Connexion  
...  
RESET time  
...  
END SCENARIO
```

SCENARIO ... LOOP ... END SCENARIO

Syntax

```
SCENARIO <scenario> [ LOOP <iteration_factor> ]  
END SCENARIO
```

Description

This instruction allows you to define a scenario block. This is the highest level of instruction.

<*scenario*> is the name of the scenario.

The optional **LOOP** keyword lets you state the identifier's scenario <*iteration_factor*>.

Associated Rules

Scenarios at the same level must have different names.

A scenario that contains other scenarios can only include **FAMILY** and **SCENARIO** statements.

<*scenario*> must begin with an upper or lower case letter and may contain letters, numbers, underscores, and dollar signs.

<*iteration_factor*> must be a positive integer.

Example

The J_n variable (n is the nesting level of the scenario that starts at 0) gives the current scenario iteration number.

```
SCENARIO principal LOOP 10
```

```
FAMILY nominal, robustness
...
SCENARIO number_one
...
SCENARIO number_one_two LOOP 10
CALL ...
PRINT iteration_number_one_two, J2
END SCENARIO
...
END SCENARIO
SCENARIO number_two LOOP 5
...
CALL ...
PRINT iteratio_number_two, J1
PRINT global_iteratio, J0
...
END SCENARIO
END SCENARIO
```

SEND

Syntax

```
SEND ( <message>, <channel> )
```

Description

The **SEND** instruction allows you to send a *<message>* on a specific *<channel>*. It calls the message-sending procedure associated with the message and communication types.

The **SEND** instruction may be located in a **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION**, or **EXCEPTION** block.

Example

```
CHANNEL appl_comm: appl_ch
#message_t msg;
SCENARIO TEST_1
FAMILY nominal
...
SEND( msg, appl_ch )
```


SHARE

Syntax

```
SHARE <identifier>
```

Description

The **SHARE** instruction allows you to specify global static variables declared in a test script.

This allows all instances of the same test script, to share these variables in multi-thread environments.

Associated Rules

The **SHARE** instruction must be at the beginning of a test script, before the first block.

The identifier is the name of the global static variable declared at the beginning of the test script.

Example

```
#static int id_Connection;  
#static int Synchro;  
#static int buffer;  
SHARE Synchro  
SCENARIO Test1  
FAMILY nominal  
...
```

TERMINATION ... END TERMINATION

Syntax

```
TERMINATION [ <proc>( [<type identifier>]{ , type identifier } ) ]  
END TERMINATION
```

Description

The **TERMINATION** instruction deletes a specific environment by executing a set of instructions or the procedure *<proc>*. **END TERMINATION** marks the end of the **TERMINATION** block.

A **TERMINATION** block or instruction applies to the set of scenarios on its level. It does not apply to sub-scenarios.

The **TERMINATION** instruction or block is optional. A maximum of one **TERMINATION** block or instruction may occur at a given scenario level. The **TERMINATION** instruction is only executed when a scenario terminates without errors.

You may place a **TERMINATION** instruction anywhere among scenarios at the same level.

Example

```
#int sock;  
TERMINATION  
    ...  
    CALL close (sock)  
    ...  
END TERMINATION  
...  
SCENARIO Main  
    ...  
END SCENARIO
```

TIME

Syntax

```
TIME ( <identifier> )
```

Description

The **TIME** instruction gives the value of the identifier timer.

The timer <identifier> must be declared by a **TIMER** instruction.

The **TIME** instruction can only appear in a C expression (analyzed or not).

Example

```
#static int id_connexion;
#static int Synchro;
#static int buffer;
TIMER globalTime
SCENARIO TEST_1
FAMILY nominal
#unsigned long C_var_Time = TIME (globalTime);
...
PRINT time, TIME (globalTime)
END SCENARIO
```

TIMER

Syntax

```
TIMER <identifier>
```

Description

The **TIMER** instruction lets you define a timer (which automatically starts after being defined).

A timer <identifier> can be declared once in the same block. The scope of an identifier is its definition block. For example, an identifier declared in an exception block can only be used in this block. However, you may use an identifier declared in the global block in all the other blocks.

Example

```
#static int id_connexion;
#static int Synchro;
#static int buffer;
TIMER globalTime
PROC dummy
TIMER procTime
END PROC
SCENARIO TEST_1
FAMILY nominal
#unsigned long C_var_Time = TIME (globalTime);
...
PRINT time, TIME (globalTime)
END SCENARIO
```

TRACE_OFF

Syntax

```
TRACE_OFF
```

Description

The **TRACE_OFF** instruction turns off storage of execution traces in the circular buffer.

This instruction is taken into account only when the **-TRACE=CIRCULAR** option is set.

Associated Rules

The **TRACE_OFF** instruction can be used in **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION**, or **EXCEPTION** blocks, but not in **CALLBACK** or **PROCSEND** blocks.

Example

```
SCENARIO one
  ...
  TRACE_OFF
  ...
END SCENARIO
```

TRACE_ON

Syntax

```
TRACE_ON
```

Description

The **TRACE_ON** instruction stores execution traces in the circular buffer.

This instruction is taken into account only when the **-TRACE=CIRCULAR** option is set.

Associated Rules

The **TRACE_ON** instruction can be used in **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION**, or **EXCEPTION** blocks, but not in **CALLBACK** or **PROCSSEND** blocks.

Example

```
SCENARIO one
  ...
  TRACE_ON
  ...
END SCENARIO
```

VAR

Syntax

```
VAR <variable>, INIT= <cmp_expression> | EV= <cmp_expression>
```

Description

This instruction allows you to initialize or check a variable. The first statement performs the initialization. The second statement compares the contents of the variable with the expression.

<variable> is a message or a variable that has previously been declared in native language. It may be any basic or structure type expression.

<cmp_expression> is in C and takes the following form:

```
cmp_expression ::= C_CPP_lang_exp
                 {cmp_init {,cmp_initialization}}
                 [attol_init {,attol_init}]
cmp_init ::= Constant=>C_CPP_lang_exp |
            Constant1 .. Constant2=>C_CPP_lang_exp |
            C_CPP_lang_exp
field_name =>C_language_expression
```

When controlling a numeric value (VAR ... EV=), you can check a range of values with one of following syntaxes:

```
VAR <variable>, EV= [ <expr_min> .. ]
VAR <variable>, EV= [ .. <expr_max> ]
VAR <variable>, EV= [ <expr_min> .. <expr_max> ]
```

These indicates that the value should be greater than or equal to the expression, less than or equal to the expression, or between the two expressions.

The **VAR** instruction may appear in a **PROC**, **SCENARIO**, **INITIALIZATION**, **TERMINATION** or **EXCEPTION** block.

The keyword **OTHERS** in a *<cmp_expression>* that represents ranges in an array or fields in a structure that have not been previously specified.

The identifiers **I1**, **I2**, ... **I20** are reserved to access different dimensions of an array. For a three-dimensional matrix, **I1** represents the index for the first dimension, **I2** the index for the second dimension, and **I3** the index for the third dimension.

Example

```
SCENARIO Main
#int matrix[3][3];
#struct {
#   char name[30];
#   char color[20];
#   double size;
# } object;
# long x;
CALL compute(matrix)
VAR matrix, EV=[ [1, 1, 1], [2, 2, 2], [1, 1, 1] ]
-- OR
VAR matrix, EV=[ 2 => [2, 2, 2], OTHERS=>[1,1,1] ]
-- multiplication table:
VAR matrix, INIT= I1 I2
VAR object, INIT=[ name => "car", color => "rouge",
&                size => 2.50 ]
VAR object, INIT=[ size => 0.10, OTHERS => "orange" ]
VAR x, EV=[11..28]
END SCENARIO
```


VIRTUAL CALLBACK

C++ only.

The **VIRTUAL** keyword modifies the **CALLBACK** statement, allowing it to handle messages using C++ inheritance.

Syntax

```
VIRTUAL CALLBACK <message_type>: <msg> ON <commtype>: <id> [<n>]  
END CALLBACK
```

Description

The **CALLBACK** instruction dynamically recalls message reception and adds a connection identifier value to a communication channel identifier.

<message_type> is a message type, previously declared with a C++ *typedef* statement. Syntax using <message_type>* is not allowed.

<msg> is the output parameter of <message_type> that must be a *polymorphic* C++ class, which means that it must contain at least one virtual method.

<commtype> is the type of communication used for reading messages.

<id> is the input connection parameter on which a message must be read.

Because a single **VIRTUAL CALLBACK** can read several message types, the implicit choice of a **CALLBACK** may be ambiguous. The following rules apply:

- If a **CALLBACK** exists for a given <message type>, System Testing chooses it.
- If not, and if the message type is actually a virtual class, then System

Testing chooses the **VIRTUAL CALLBACK** with the closest type in terms of path in the inheritance diagram of *<message_type>*.

- If more than one **VIRTUAL CALLBACK** can be chosen by following the above rules, the **CALLBACK** is ambiguous and System Testing produces an error.

Example

```
class high_level_message
{
public:
    char from[12];
    char applname[12];
    virtual int get_type(){return 0;}
};

class ack : public high_level_message
{
public:
    int get_type(){return ACK;}
};

class negack : public high_level_message
{
public:
    int get_type(){return NEG_ACK;}
};

class data : public high_level_message
{
public:
    char userdata[MAX_USERDATA_LENGTH];
    int length;
    int get_type(){return DATA;}
};

typedef high_level_message * pt_high_level_message;
VIRTUAL CALLBACK pt_high_level_message: msg ON appl_comm:
id
    CALL get_message ( &id, &msg, 0 ) @@ errcode
        MESSAGE_DATE
        IF ( errcode == err_empty ) THEN
            NO MESSAGE
        END IF
        IF ( errcode != err_ok ) THEN
            ERROR
        END IF
    END CALLBACK
```

This VIRTUAL CALLBACK allows you to read **high_level_message**, **ack**, **negack** and **data** message types, as shown on the following lines:

```
MESSAGE data : a_data
MESSAGE ack : my_ack
MESSAGE negack : my_neg_ack
MESSAGE high_level_message : hm
DEF MESSAGE my_ack, EV={}
WAITTIL (MATCHING(my_ack), WTIME==300)
  DEF MESSAGE a_data, EV={}
  WAITTIL (MATCHING(a_data), WTIME==300)
```

VIRTUAL PROCSSEND

For C++ only.

The **VIRTUAL** keyword modifies the **PROCSSEND** statement, allowing it to handle messages using C++ inheritance.

Syntax

```
VIRTUAL PROCSSEND <message_type>: <msg> ON <commtype>: <id>  
END CALLBACK
```

Description

The **PROCSSEND** instruction allows you to define a message-sending procedure using C++ classes.

<message_type> is a message type, previously declared with a C++ *typedef* statement. Syntax using <message_type>* is not allowed.

<msg> is the output parameter of <message_type> that must be a *polymorphic* C++ class, which means that it must contain at least one virtual method.

<commtype> is the type of communication used for reading messages.

<id> is the input connection parameter on which a message must be read.

Associated Rules

Because a single **VIRTUAL PROCSSEND** can read several message types, the implicit choice of a **PROCSSEND** may be ambiguous. The following rules apply:

- If a **PROCSSEND** exists for a given <message type>, System Testing

chooses it.

- If not, and if the message type is actually a virtual class, then System Testing chooses the **VIRTUAL PROCSEND** with the closest type in terms of path in the inheritance diagram of *<message_type>*.
- If more than one **VIRTUAL PROCSEND** can be chosen by following the above rules, the **PROCSEND** is ambiguous and System Testing produces an error.

Example

```
VIRTUAL PROCSEND pt_high_level_message : msg ON appl_comm
: id_stack
CALL send_message (msg) @ err_ok
END PROCSEND
```

This **VIRTUAL PROCSEND** example allows you to send **high_level_message**, **ack**, **negack** et **data** message types, as shown on the following lines:

```
MESSAGE data : a_data
MESSAGE ack : my_ack
MESSAGE negack : my_neg_ack
MESSAGE high_level_message : hm
VAR a_data, INIT={applname=>"SATURN",userdata=>"Hello
Saturn!"}
SEND( a_data , appl_ch )
VAR my_ack, INIT={applname=>"SATURN"}
SEND(my_ack , appl_ch )
VAR my_neg_ack, INIT={applname=>"SATURN"}
SEND(my_neg_ack , appl_ch )
```

WAITTIL

Syntax

```
WAITTIL (<passed_expr>, <failed_expr>)
```

Description

This instruction waits for several events and/or a timer.

<passed_expr> is a parameter that contains a Boolean expression. If this expression is true, the waiting process is disabled and the test sequence continues.

<failed_expr> is a parameter that contains a Boolean expression. If this expression is true, the waiting process is disabled and it ends with an error.

When <failed_expr> is true, the execution of the scenario containing the **WAITTIL** is interrupted. The next scenario at the same level is then executed.

To use this instruction, you need to take the following actions:

- Declare a type of communication with the **COMMTYPE** instruction.
- Declare a communication channel with the **CHANNEL** instruction.
- Declare the reference messages with the **MESSAGE** instruction.
- Write a callback for a non-blocking read of communication and message type.
- Define the expected values for each reference message with the **DEF_MESSAGE** instruction.
- Associate the identifier of a communication connection with the **ADD_ID** instruction.

- Use the four comparison operators, **MATCHING**, **MATCHED**, **NOTMATCHING**, **NOTMATCHED**, and the timer **WTIME**. Also use the **&&** (logical *and*) and **||** (logical *or*) operators.

Interface file

```
#include <sys/types.h>
#include <sys/socket.h>
typedef struct{
char validity[60];
char cnumber[60];
}account_t;
typedef struct {
char code;
char name[60];
char number[60];
account_t account;
}client_info_t;
```

Example

```
HEADER "Client","1.0","1.0"
#typedef int socket_id_t
COMMTYPE socket IS socket_id_t
CHANNEL socket: ch
MESSAGE client_info_t: msg
CALLBACK client_info_t: info ON socket: id
CALL read(id, &info, sizeof(client_info_t))@ret
IF (ret == 0) THEN
NO_MESSAGE
END IF
MESSAGE_DATE
END CALLBACK
SCENARIO simple_test
...
ADD_ID(ConnectId,ch)
DEF_MESSAGE msg, EV={name=>"John Smith"}
WAITTIL ( MATCHING(msg), NOTMATCHING(msg) )
END SCENARIO
```

WHILE ... END WHILE

Syntax

```
WHILE ( condition )  
END WHILE
```

Description

The instruction **WHILE** is a control structure. All the instructions between **WHILE** and **END WHILE** is executed if the condition is true.

Example

```
#int i = 0;  
SCENARIO Main  
CALL api_func...  
WHILE (i<100)  
CALL api_val(i)  
VAR i, INIT=i+1  
END WHILE  
...  
END SCENARIO
```


WTIME

Syntax

WTIME

Description

WTIME is a macro that acts as a timer in a **WAITTIL** instruction.

The value of **WTIME** is reset to zero before every **WAITTIL**.

You can assign parameters to the timer's unit of time in the Target Deployment Port.

Example

```
...
SCENARIO Acknowledge
...
WAITTIL (MATCHING (OK), WTIME == 1000 )
END SCENARIO
```

Macro Keywords

ATL_OCCID

Description

ATL_OCCID is a macro that returns the value of the occurrence identification number (**OCCID**) that uniquely identifies a virtual tester.

You can change the occurrence identification number of a virtual tester by adding the **-OCCID=<number>** parameter to the command line of the generated virtual tester.

By default, the value of **ATL_OCCID** within a test script is **0**.

Example

```
HEADER "Client", "1.0", "3.0"  
SCENARIO Main  
    . . .  
    PRINT occnumber, ATL_OCCID  
    . . .  
END SCENARIO
```

ATL_TIMEOUT

Description

The value of **ATL_TIMEOUT** is calculated from a **WTIME** expression used in the **WAITTIL** statement. The **ATL_TIMEOUT** macro is an integer and uses the time unit defined in the Target Deployment Port. By default, the time unit is a hundredth of second.

Supervisor Script Keywords

Supervisor Script Keywords (.spv Supervisor Script)

Supervisor Script keywords are not case sensitive. This means that **SHELL**, **shell**, and **Shell** are equivalent. For conventional purposes, however, this document uses upper case notation for Supervisor Script keywords in order to differentiate from native source code.

- Expressions
- Variables
- Environment variables
- CHDIR
- COPY
- DELETE
- DO
- ENDOF
- ERROR
- EXECUTE
- EXIT

- HOST
- IF ... THEN ... ELSE ... END IF
- INCLUDE
- MEMBERS
- MKDIR
- PAUSE
- PRINT
- PRINTLN
- RENDEZVOUS
- RMDIR
- SET
- SHELL
- STATUS
- STOP
- TIMEOUT
- TRACE ... FROM
- UNSET
- WHILE

CHDIR

Purpose

The CHDIR instruction changes the current working directory of the System Testing Supervisor machine or of a target machine.

Syntax

```
CHDIR [ <hostname>: ] <directory>
```

where:

- <hostname> is an optional logical name of a target machine (see HOST)
- <directory> is the relative or absolute path of a directory

Description

When supervisor execution starts, the working directory of the System Testing Supervisor machine is the current directory of the shell that runs the System Testing Supervisor.

When the script starts, the working directory of the target machine is the directory where the Agent has been started.

The <directory> path may contain local environment variables from the System Testing Supervisor machine, or remote environment variables defined on the target machine. For more information, refer to the section on Environment variables.

If the operation fails, you will receive an error message (see ERROR).

The <directory> path may contain long quoted pathnames, such as **"C:\Program Files\Rational\Test RealTime\"**.

Example

```
HOST target IS workstation.domain.com
CHDIR localdir
CHDIR $ATS_DIR
CHDIR target:$HOME
CHDIR target:/tmp/project
SET DIR=C:\tmp
CHDIR $DIR
```

COPY

Purpose

The COPY instruction transfers a binary or ASCII file from the System Testing Supervisor machine to a target machine, or the opposite.

Syntax

```
COPY [<hostname>:]<destination> [<hostname>:]<source> [/ASCII ]
```

where:

- *<source>* is the absolute or relative filename of the file to be copied.
- *<destination>* is the absolute or relative path to which *<source>* is to be copied.
- *<hostname>* is the optional name of the source or destination machine.

Description

When the *<hostname>* is not specified, the filename refers to a local file on the System Testing Supervisor machine. When a *<hostname>* is specified, the filename refers to a file on the corresponding remote host.

COPY instructions can only transfer files from the System Testing Supervisor machine to a remote machine, or from a remote machine to the System Testing Supervisor machine. Transfers from one remote machine to another must be performed using two COPY instructions.

By default, transfers are in binary mode. If you specify the keyword /ASCII, the transfer is performed in character mode, which insures that text files are correctly copied between different types of machines. In binary mode, the target file's access permissions are updated so that the file is executable.

A filename may contain environment variables that are local to the System Testing Supervisor machine or that are defined on the remote machine. For more information, refer to the section on Environment variables.

If the file to be copied does not exist or is read-protected, you will receive an error message (see ERROR).

Path and filenames may contain long quoted pathnames, such as "**C:\Program Files\Rational\Test RealTime**".

Example

```
HOST target_1 IS antares
...
COPY localfile target_1:$HOME/file.bin
COPY target_1:remotefile localfile /ASCII
...
```


DELETE

Purpose

The DELETE instruction deletes a local or remote file.

Syntax

```
DELETE <filename>
```

where:

- <filename> is a local or remote file to be deleted.

Description

<filename> may be specified with an absolute or relative path, or as <hostname>: <filename>, where <hostname> is a remote host running a System Testing Agent daemon.

The filename may contain environment variables that are local to the System Testing Supervisor machine or that are defined on the remote machine. For more information, refer to the section on Environment variables

If the file to be deleted does not exist or is write-protected, you will receive an error message. (See ERROR.)

Path and filenames may contain long quoted pathnames, such as "C:\Program Files\Rational\Test RealTime\".

Example

```
HOST target_2 IS 123.4.56.7(10098)
DELETE target_2:$DIR/../remote_file
DELETE local_file
```

DO

Purpose

The **DO** instruction executes a program on a remote machine and waits for the end of its execution.

Syntax

```
DO [ <process> IS ] <hostname>: <program> [ <parameters> ]
```

where:

- <process> optionally assigns a process name to the program
- <hostname> is the name of the remote machine as defined by a **HOST** instruction
- <program> is the name of the program to execute
- <parameters> is a set of optional parameters that can be sent to <program>

Description

DO is a blocking instruction that waits for the program to end.

The field <hostname> is mandatory and must specify a remote machine.

You can give a logical name to a program by including the clause <process> **IS**. You can then form expressions with the **ENDOF** and **STATUS** operators.

A process name may only appear once in a supervision script, otherwise you will receive an error when the scenario does not execute. If <process> **IS** is not present, the **ENDOF** and **STATUS** operators cannot be used.

While the program runs, all logs sent to the standard and error outputs are redirected to the supervisor, except if you have set **TRACE OFF**.

If the program does not start or does not have execution permission, an error message is produced. (See **ERROR**.)

Note If a logical process name is used in a **DO** instruction within a **WHILE** loop, the name refers not to a single process, but a group of processes. (See the **ENDOF** and **STATUS** operators.)

Example

```
HOST remote IS 192.3.2.1
DO process_1 IS remote: ls /tmp -l
i = 1
WHILE i < 10
DO group IS remote:program
i = i + 1
END WHILE
-- the variable group refers to a group of 9
-- executions of the process called program
```

ENDOF

Purpose

ENDOF is a Boolean function that tests whether *<process>* has ended or not. **ENDOF** is true if the execution of *<process>* has ended.

Syntax

```
ENDOF ( <process> )
```

<process> is a logical process name, defined with an **EXECUTE** statement.

Description

You can use the **ENDOF** function in expressions analyzed by the supervisor.

ENDOF is a non-blocking operator.

If an unknown process identifier is specified, an error is generated during analysis of the supervision script before it is executed.

Note If an **EXECUTE** instruction is placed inside a **WHILE** loop, the process identifier denotes a group of processes. In this case, an **ENDOF** expression with this process identifier is true when all the processes associated with the identifier have ended.

Example

```
...  
i = 1  
WHILE i < 10  
    EXECUTE proc_group IS machine:program  
    i = i + 1  
END WHILE  
...
```

```
IF ENDOF ( proc_group ) THEN
    PRINT "end of execution of all processes"
END IF
```

ERROR

Purpose

The **ERROR** instruction indicates to the supervisor whether or not execution of a scenario should be interrupted if an error occurs.

Syntax

```
ERROR [ ON | OFF ]
```

Description

Use **ERROR ON** to interrupt execution of the supervision script if an error is detected.

Use **ERROR OFF** to ignore errors and continue execution of the supervision script.

In both cases, you will still receive an error message through the standard output.

The use of **ERROR** in supervision scripts is optional. **ERROR ON** is the default setting.

You may use **ERROR ON** and **ERROR OFF** several times in the same supervisor script.

Example

```
...
COPY localfile_1 target:file_1
ERROR OFF
DELETE localfile_1
ERROR ON
...
ERROR OFF
```

```
EXECUTE target:file_1  
ERROR ON
```

EXECUTE

Purpose

The EXECUTE instruction executes the program *<program_name>* on the *<hostname>* defined by a previous **HOST** instruction.

Syntax

```
EXECUTE [ <process> IS ] <hostname>: <program> [ <parameters> ]
```

where:

- *<process>* optionally assigns a process name to the program
- *<hostname>* is the name of the remote machine as defined by a **HOST** instruction
- *<program>* is the name of the program to execute
- *<parameters>* is a set of optional parameters that can be sent to *<program>*

Description

EXECUTE is a non-blocking instruction that asynchronously starts the *<program>* on *<hostname>*, and then returns.

The field *<hostname>* is mandatory and must specify a remote machine.

You can assign a logical name to the *<program>* by adding the optional *<process>* **IS** statement. You can use this logical name to form expressions with the **ENDOF** and **STATUS** operators.

Any logical process name must be unique to a supervision script, otherwise it will generate an error when the scenario execution fails.

If no logical process name is assigned to the program execution, the **ENDOF** and **STATUS** operators will generate an error during the analysis of the supervisor script.

While *<program>* is running, all logs normally sent to the standard and error outputs are redirected to the supervisor, except if you have used a **TRACE OFF** statement.

If the *<program>* file is missing or does not have execution permission, an error is generated.

Note If a logical process name is used in an **EXECUTE** instruction within a **WHILE** loop, the name refers not to a single process, but a group of processes. (See the **ENDOF** and **STATUS** operators).

Example

```
HOST remote IS 192.3.2.1
EXECUTE process_1 IS remote: ls /tmp -l
EXECUTE remote: myFoo
i = 1
WHILE i < 10
    EXECUTE group IS remote:program
    i = i + 1
END WHILE
-- the variable group refers to a group of 9
-- executions of the process called program
```

EXIT

Purpose

The **EXIT** instruction stops execution of the supervision script.

Syntax

```
EXIT [ "<message>" ]
```

<message> is an optional character string delimited by double-quotes (").

Description

Stopping the supervisor causes all processes started by agents to stop as well.

The optional *<message>* is printed as an information message.

Note If you need to include a double-quote in the message, use \".

Example

```
HOST remote IS 192.6.2.1
...
IF ( i = 3 ) THEN
    EXECUTE remote: ls /tmp -l
ELSE
    EXIT "Exit on incorrect value of \"i\""
END IF
```

HOST

Purpose

The **HOST** instruction assigns a logical machine name to a target machine.

Syntax

```
HOST <logical_name> IS <address> [ (<port_number>) ]
```

<logical_name> is the identifier of the target machine.

<address> is the network address of the target machine

<port_number> is the network port to which the target machine's Agent is assigned.

Description

Executing a **HOST** instruction opens a connection with an agent on the target machine.

Logical machine names are used in **CHDIR**, **COPY**, **DO**, **DELETE**, **EXECUTE**, **MKDIR**, **RMDIR**, **SET**, **TRACE** and **UNSET** instructions to refer to target machines.

The host <address> may be:

- a hostname (for example: **workstation.domain.com**),
- an alias (for example: **workstation**),
- or an IP address (for example: **155.22.9.3**).

The TCP/IP port number is optional. It helps specify the port used by the target machine's agent that listens for connection demands. By default, the

port used by the supervisor is the one specified by the **ATS_PORT** environment variable, or 10000.

A logical machine name must be unique within the supervision script. If the System Testing Supervisor machine cannot connect to the agent, the supervisor produces an error message and terminates, regardless of any **ERROR** statement.

Example

```
HOST machine_1 IS localhost
HOST machine_2 IS 193.256.6.2(10098)
HOST machine_3 IS $HOSTNAME
COPY local_file machine_2:remote_file
DO machine_1:program
```

IF ... THEN ... ELSE ... END IF

Purpose

The **IF ... END IF** statement allows you to define a conditional behavior based on the result of an expression.

Syntax

```
IF <expression> THEN
ELSE
END IF
```

<expression> is a Boolean expression. See Expressions.

Description

IF defines the Boolean expression.

Instructions following the **THEN** keyword are executed if the expression is true.

Instructions following **ELSE** are executed if the expression is false.

END IF marks the end of the of the IF statement.

Example

```
HOST machine IS 193.6.2.1
DO prepro IS machine:preprocessing.exe
IF ( STATUS ( prepro ) == 0 ) THEN
    PRINTLN "preprocessing OK"
ELSE
    PRINTLN "preprocessing FAILED"
EXIT
END IF
```

INCLUDE

Purpose

The **INCLUDE** instruction allows you to nest supervision scripts.

Syntax

```
INCLUDE "<filename>"
```

<filename> is the absolute or relative file name of an included supervision script, delimited by double quotes (").

Description

There is no limit to the levels of nested **INCLUDE** commands.

If an infinite loop of included files is detected during analysis, you will receive an error message and the execution will fail.

INCLUDE instructions may appear anywhere in a supervision script, including inside a structured **IF** or **WHILE** instruction.

There is no default file extension. If the filename has an extension, you must state it in the **INCLUDE** instruction.

Example

```
HOST machine_1 IS 193.6.2.1
INCLUDE "included_file.spv"
...
DO test_1 IS machine_1:test_1
```

MEMBERS

Purpose

The **MEMBERS** instruction lets you declare the number of members awaited at a given rendezvous.

Syntax

```
MEMBERS <rendezvous> <number>
```

where:

- *<rendezvous>* is the rendezvous identifier
- *<number>* is a positive integer representing the number of members to wait for

Description

MEMBERS lets you synchronize virtual testers with the **RENDEZVOUS** instructions or with other applications with the rendezvous Target Deployment Port.

A *<rendezvous>* identifier must be unique within the supervision script. If not, an error message is produced and the scenario execution fails.

Example

```
...  
MEMBERS beginning 3  
...  
EXECUTE machine_1:test1  
EXECUTE machine2:test2  
...  
RENDEZVOUS beginning
```

MKDIR

Purpose

The **MKDIR** instruction creates a new directory on the System Testing Supervisor machine or on a target machine.

Syntax

```
MKDIR [ <hostname>: ] <directory>
```

where:

- *<hostname>* is an optional logical name of a target machine (see HOST)
- *<directory>* is the relative or absolute path of a directory

Description

The directory path name may contains local environment variables of the System Testing Supervisor machine, or remote environment variables defined on the target machine.

If the operation fails, the script returns an error message.

Example

```
HOST target IS workstation.domain.com(10098)
MKDIR ../localdir
MKDIR target:$HOME/tmp
```


PAUSE

Purpose

You may use the **PAUSE** instruction to delay script execution.

Syntax

```
PAUSE <duration>
```

<duration> is an integer expression specifying the length of the delay in seconds.

Description

<duration> is in seconds. It may be an integer constant or an integer expression.

Executing the **PAUSE** instruction will not delay any processes already running on target machines.

Example

```
DELAY = 25  
...  
PAUSE 3  
...  
PAUSE DELAY
```

PRINT

Purpose

The **PRINT** instruction prints *<argument>* to the supervision script execution log file without a carriage return or line feed.

Syntax

```
PRINT <argument>
```

where:

- *<argument>* is a string or a variable that points to a string

Description

The **PRINT** instruction does not cause a carriage return or line feed after printing the value of *<argument>*.

<argument> can be a string constant, delimited by quote double-quotes, or a variable integer value used in the scenario.

If *<argument>* uses an unknown variable, the scenario execution exits with an error message.

Example

```
var_i = 25  
PRINT "value of var_i "  
PRINT var_i
```

PRINTLN

Purpose

The **PRINTLN** instruction prints *<argument>* to the supervision script execution log file with a carriage return or line feed.

Syntax

```
PRINTLN [ <argument> ]
```

<argument> is an optional string or identifier that is to be printed.

Description

The value of *<argument>* can be a string constant, delimited by double-quotes, or a variable integer value used in the scenario.

If you provide no argument, the instruction causes a carriage return or line feed.

If *<argument>* uses an unknown variable, the scenario execution exits with an error message.

Example

```
var_i = 25
PRINTLN "value of var_i "
PRINTLN var_i
```

RENDEZVOUS

Purpose

The **RENDEZVOUS** instruction synchronizes virtual testers and other processes.

Syntax

```
RENDEZVOUS <rendezvous>
```

<rendezvous> is a rendezvous identifier, previously declared by a **MEMBERS** statement.

Description

When the scenario reaches a **RENDEZVOUS** statement, the script is halted until all declared members arrive at the rendezvous. When the rendezvous is met by all members, the supervisor orders all processes to resume.

RENDEZVOUS identifiers must be unique in the script, including from logical process names or variable names, otherwise you will receive an error when execution fails.

If the rendezvous does not occur before the end of the timeout delay, you will receive an error. The default delay is five minutes. You can modify the delay with the **TIMEOUT** instruction.

Example

```
...  
MEMBERS test1_test2 3  
EXECUTE machine_1:test1  
EXECUTE machine_2:test2  
RENDEZVOUS test1_test2
```

RMDIR

Purpose

The **RMDIR** instruction deletes a directory from the System Testing Supervisor machine or from a target machine.

Syntax

```
RMDIR [ <hostname>: ] <directory>
```

where:

- *<hostname>* is an optional logical name of a target machine (see HOST)
- *<directory>* is the relative or absolute path of a directory

Description

The directory path name may contain local environment variables of the System Testing Supervisor machine or remote environment variables defined on the target machine. For more information, refer to the section on Environment variables.

If the operation fails, the script returns an error message.

The *<directory>* path may contain long quoted pathnames, such as "**C:\Program Files\Rational\Test RealTime**".

Example

```
HOST target IS antares.tlse.fr(10098)
RMDIR ../localdir
RMDIR target:$HOME/tmp
```

SET

Purpose

The **SET** instruction sets an environment variable on either the System Testing Supervisor machine or the target machine.

Syntax

```
SET [ <hostname>: ] <env_var> << <expression>  
SET [ <hostname>: ] <env_var> = <string>
```

<hostname> is the logical name of the target machine,

<env_var> is the name of the environment variable,

<expression> is a numerical expression,

<string> is a text string.

Description

<hostname> must be previously declared with a **HOST** instruction. If you do not specify a hostname, the **SET** instruction sets a local environment variable on the supervisor machine.

The environment variable is set when the SET instruction executes. It keeps its value until the end of the execution, or until it resets.

The string from the equal sign (=) to the end of the line belongs to the expression.

To evaluate an expression and assign it to the variable, use the << symbol. The expression may contain variables.

Example

```
HOST target IS workstation(10098)
...
SET LOCAL_TMP_DIR=/tmp
SET target:REMOTE_TMP_DIR << $TMPDIR
SET target:NUMVALUE <<i+2
```

SHELL

Syntax

```
SHELL command
```

Purpose

The **SHELL** instruction executes a command by the System Testing Supervisor machine.

Description

SHELL commands block execution of the supervision script until the command is complete.

The command's execution log is not recorded in the supervision script execution log.

Example

```
...  
SHELL ls /tmp -l ...
```


STATUS

Purpose

STATUS is an integer operator that retrieves the code returned by a remote process when it terminates.

Syntax

```
STATUS ( process )
```

where:

- *<process>* is a logical process identifier

Description

The execution of a **STATUS** expression does not block execution of the scenario.

Applying **STATUS** to an ongoing process always returns a zero value. We recommend you use the **STATUS** operator in conjunction with **ENDOF**.

Note If you place an **EXECUTE** or **DO** instruction inside a **WHILE** loop, the process identifier denotes a group of processes. In this case, a **STATUS** expression returns a binary result or code from all the processes in the group. For example, if ten processes terminate with a return code of 0 and one process terminates with the return code of 1, the **STATUS** operator returns the value 1.

Example

```
EXECUTE proc_1 IS machine:foo0098
WHILE !ENDOF(proc_1)
PAUSE 1
END WHILE
j = STATUS ( proc_1 )
```

```
IF j != 0 THEN
  PRINT "incorrect termination of program -> "
  PRINTLN j
  EXIT
END IF
```

STOP

Syntax

```
STOP <process>
```

where:

- <process> is the identifier of a process

Purpose

The **STOP** instruction stops a process began with the **EXECUTE** instruction.

Example

```
HOST target IS antares  
EXECUTE server IS machine:server  
...  
STOP server
```

TIMEOUT

Syntax

```
TIMEOUT <integer>
```

Purpose

The **TIMEOUT** instruction lets you define the time to wait for a rendezvous.

The value is measured in seconds.

Description

You may use only one **TIMEOUT** instruction in a test script.

The default value is 300 seconds (5 minutes).

Example

```
HOST machine_1 IS 193.5.4.3
HOST machine_2 IS remote.domain.fr
TIMEOUT 40
RENDEZVOUS phase_1
```

TRACE ... FROM

Syntax

```
TRACE ON | OFF [ FROM <host_name> ]
```

Purpose

The **TRACE** instruction enables or disables execution traces from the machine specified by **host_name**, where this name was defined by a **HOST** instruction.

The traces are consolidated into the supervisor log file.

The keyword **ON** enables traces.

The keyword **OFF** disables traces.

Description

If the clause **FROM host_name** is not present, all traces from all machines are enabled or disabled.

If the clause **FROM host_name** is present, traces from machine **host_name** are enabled or disabled.

If you specify an unknown host name, you will receive an error when scenario execution fails.

By default, traces follow the **HOST** instruction.

Example

```
HOST machine_1 IS 193.5.4.3
HOST machine_2 IS remote
TRACE OFF FROM machine_1
```

UNSET

Syntax

```
UNSET [ <hostname>: ] <env_var>
```

where:

- *<hostname>* is the logical name of the target machine (See HOST.)
- *<env_var>* is the name of the environment variable

Purpose

The **UNSET** instruction deletes an environment variable from the System Testing Supervisor machine or from the target machine.

Description

Hostname is the logical name on a target machine as defined in the HOST instruction. If you do not specify a hostname, the **UNSET** instruction deletes a local variable.

When you execute the **UNSET** instruction, the environment variable deletes until the end of the execution, or until you reset it.

Example

```
HOST target IS workstation(10098)
...
SET LOCAL TMP_DIR=/tmp
SET target:REMOTE_TMP_DIR=$TMPDIR
...
UNSET LOCAL TMP_DIR
UNSET target:REMOTE_TMP_DIR
...
```

WHILE

Syntax

```
WHILE expression
instructions
END WHILE
```

Purpose

The **WHILE** instruction creates an execution loop.

Example

```
HOST machine IS 193.6.2.1
EXECUTE proc_1 IS machine:program
i = 1
WHILE !ENDOF ( proc_1 )
    PAUSE 1
    i = i + 1
END WHILE
j = STATUS ( proc_1 )
PRINT "execution time: "
PRINTLN i
PRINT "return code: "
PRINTLN j
```

Variables

A supervision script may contain integer variables only.

The system implicitly declares variables the first time they appear. The variable must first appear in an assignment instruction.

A variable must have a different name from any logical hostname defined in a **HOST** instruction, from any logical process name defined in an **EXECUTE** instruction, and from any **RENDEZVOUS** name. Otherwise, you will receive an error when scenario execution fails.

Variable names must begin with an upper or lowercase letter or with an underscore (`_`), followed, if necessary, by a series of letters, digits, or underscore characters.

Variable names are case sensitive. For example, the variable **Aa5** is different from the variable **aA5**.

Example

```
HOST machine IS 193.6.2.1
EXECUTE proc_1 IS machine:program
i = 1
-- declaration of i
WHILE !ENDOF ( proc_1 )
    PAUSE 1
    i = i + 1
END WHILE
j = STATUS ( proc_1 )
-- declaration of j
PRINT "execution time "
PRINTLN i
PRINT "return code "
PRINTLN j
```


Environment variables

Supervisor scripts may read and write environment variables on the System Testing Supervisor machine and on target machines.

Precede an environment variable name with a dollar sign (\$) to substitute the environment variable by its value within a statement.

To force a variable to refer to the environment of the System Testing Supervisor machine, precede the environment variable with the 'at' sign (@) instead of the dollar sign.

Example

```
HOST machine IS $HOSTNAME
-- show the contents of the target home directory
DO machine: ls $HOME
-- show the contents of the local home directory
SHELL ls $HOME
```

Expressions

Supervision scripts may contain integer expressions only.

You may use expressions in variable assignments, **IF** instructions, and **WHILE** instructions.

Expressions may contain the following operators:

Operator	Description
==	Equals
!=	Does not equal
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
+	Plus
-	Unary or binary minus
*	Multiply
/	Divide
%	Modulo
!	Negation
&&	Logical AND
	Logical OR
ENDOF	See ENDOF
STATUS	See STATUS

Expressions may be nested with parentheses. Operators obey the following

ascending order of priority:

- &&, ||
- ==, !=
- >, >=, <, <=
- +, Unary or binary -
- *, /, %
- **!, ENDOF, STATUS**

Example

```
HOST machine IS 193.6.2.1
EXECUTE proc_1 IS machine:program
i = 1
  -- declaration of i
j = ( i + 3 2 ) + ( i <= 2 )
  -- declaration of j
PRINTLN j
```

System Testing Probe Macros

Probe Macros allow you to manually instrument your source code under test to add message tracing capability to your test binary. This feature is tightly linked with System Testing for C.

Upon execution of the instrumented binary, the probes write trace information on the exchange of specified messages to the **.rio** System Testing output file.

Using Probe Macros

Before adding probe macros to your source code, add the following **#include** statement to each source file that is to contains a probe:

```
#include "atlprobe.h"
```

The `atl_start_trace()` and `atl_end_trace()` macros must be called when the application under test starts and terminates.

Other macros must be placed in your source code in locations that are relevant for the messages that you want to trace.

Probe Macros

- `atl_dump_trace()`
- `atl_end_trace()`
- `atl_recv_trace()`
- `atl_select_trace()`
- `atl_send_trace()`
- `atl_start_trace()`
- `atl_format_trace()`

atl_dump_trace()

Purpose

Writes traces from the *custom* location, when **FIFO**, **File** or **User** buffer is selected, to the **.rio** System Testing output files for C.

This is not used in **Default** mode.

Syntax

```
atl_dump_trace()
```

Example

```
int main(int argc, char** argv)
{
    ...
    atl_start_trace(atl_client, "../res/", client, 0);
    atl_start_trace(atl_serv, "../res/", serv, 0);
    ...
    atl_end_trace(atl_client);atl_end_trace(atl_serv);
    atl_dump_trace();
    ...
}
```

atl_end_trace()

Purpose

Closes the trace environment of an instance. This macro must be executed before the application terminates.

Syntax

```
atl_end_trace(<ctx>)
```

where:

- <ctx> is the identifier of a context variable linked to an instance.

Example

```
int main(int argc, char** argv)
{
    ...
    atl_start_trace(atl_client, "client.rio", client, 1000);
    atl_start_trace(atl_serv, "serv.rio", serv, 2000);
    ...
    atl_end_trace(atl_client);
    atl_end_trace(atl_serv);
    ...
}
```

atl_format_trace()

Purpose

This macro allows you to include a format file for the trace output.

Syntax

```
atl_format_trace(<file>)
```

where:

- <file> is the name of a format file, containing System Testing FORMAT instructions for C.

Example

```
int main(int argc, char** argv)
{
    ...
    atl_start_trace(atl_client, "client.rio", client, 1000);
    atl_start_trace(atl_serv, "serv.rio", serv, 2000);
    ...
    atl_format_trace("atl_format.hts");
    ...
    atl_end_trace(atl_client);
    atl_end_trace(atl_serv);
    ...
}
```

atl_recv_trace()

Purpose

Traces the reception of message.

Syntax

```
atl_recv_trace(<ctx>, <dist>, <msg>, <type>, <msgname>)
```

where:

- <ctx> is the identifier of a context variable linked to an instance.
- <dist> is the identifier of the emitter of a message.
- <msg> is the message address.
- <type> is the message type as defined in the **msg_type.h** file.
- <msgname> is the name of the message traced in the report.

Example

```
atl_recv_trace(atl_client, fl, serv, t_cost, cost);  
atl_send_trace
```


atl_select_trace()

Purpose

Specifies for a given union type, the field to use for a message instance.

Syntax

```
atl_select_trace(<ctx>, <idx>, <val>)
```

where:

- <ctx> is the identifier of a context variable linked to an instance.
- <idx> is a union type semantic path.
- <val> is the rank of the field used in the union type.

Example

```
atl_recv_trace(atl_client, f1, serv, t_cost, cost);  
atl_send_trace
```

atl_send_trace()

Purpose

Traces a message sent.

Syntax

```
atl_send_trace(<ctx>, <dist>, <msg>, <type>, <msgname>)
```

where:

- <ctx> is the identifier of the context variable linked to an instance.
- <dist> is the identifier of the receiver of a message.
- <msg> is the message identifier.
- <type> is the message type as defined in the **msg_type.h** file.
- <msgname> is the name of the message traced in the report.

Example

```
atl_send_trace(atl_client, fl_serv, t_cost, cost);
```

atl_start_trace()

Purpose

Initializes the environment of an instance trace. This macro must be executed when the application under test starts.

Syntax

```
atl_start_trace(<ctx>, <path>, <instance>, <size>)
```

where:

- <ctx> is an **ATL_CTX_PRB** type identifier. This type is defined in the file produced by the Probe Test Script Generator or in the probecst.h file
- <path> is the path to the .rio file to which the traces are to be written
- <instance> is the name of the instance
- <size> specifies the memory size used in bytes in FIFO or USER mode

Example

```
int main(int argc, char** argv)
{
    ...
    atl_start_trace(atl_client, "../res/", client, 0);
    atl_start_trace(atl_serv, "../res/", serv, 0);
    ...
    atl_end_trace(atl_client);
    atl_end_trace(atl_serv);
    ...
}
```

Appendices

5

This section provides extra reference information that may be necessary when using the product.

GUI Macro Variables

Some parts of the graphical user interface (GUI) allow you to specify command lines, such as in the Tools menu or in User Command nodes.

To enhance the usability of this feature, the product includes a macro language, allowing you to pass system and application variables to the command line.

Usage

Macro variables are preceded by **\$\$** (for example: **\$\$WSPNAME**).

Macro functions are preceded by **@@** (for example: **@@PROMPT**).

Environment variables are also accessible, and start with **\$** (for example: **\$DISPLAY**).

When specifying a command line, variables and functions are replaced with their value.

In Windows, when long filenames are involved, it is necessary to add double quotes (" ") around filename variables. For example:

```
"C:\Program Files\Internet Explorer\IEXPLORE.EXE"
```

"\$NODEPATH"

Node variables are context-sensitive: the variable returned relates to the node selected in the File or Test Browser. Multiple selections are supported. If a node variable is invoked when there is no selection, no value is returned by the variables.

Macro variables and functions are case-insensitive. For clarity, they are represented in this document in upper case characters.

Language Reference

- Global variables: not node-related, include Workspace and application parameters.
- Node attribute variables: general attributes of a node.
- Functions: return a value to the command line after an action has been performed.

Functions

Functions process an input value and return a result. Input values are typically a global or node variable.

Environment Variable	Description
@@PROMPT ('<message>')	Opens a prompt dialog box, allowing the user to enter a line of text. The optional <message> parameter allows you to define a prompt message, surrounded by single quotes (').
@@EDITOR (<filename>)	Opens the product Text Editor.
@@OPEN (<filename>)	Opens <filename>. <filename> must be a file type recognized by the product. This is the equivalent of selecting Open from the File menu.

Global Variables

Global variables always return the same value throughout the Workspace.

Environment Variable	Description
\$\$PRJNAME	Returns the name of the current .rtp Project file
\$\$PRJDIR	Returns the directory name of the current .rtp Project file
\$\$PRJPATH	Returns the absolute path of the current .rtp Project file
\$\$VCSDIR	Returns the local repository for files retrieved from Rational ClearCase, as specified in the ClearCase Preferences dialog box
\$\$CPPINCLUDES	Returns the directory of C and C++ include files, as specified in the Directories Preferences dialog box
\$\$PERL	Returns the full command-line to run the PERL interpreter included with the product

\$\$CLIPBOARD	Returns the text content of the clipboard
\$\$VCSITEMS	Returns a list of installed configuration management system (CMS) tools

Node Attribute Variables

These variables represent the attributes of a selected node. If no node is selected, these variables return an empty string.

Environment Variable	Description
\$\$NODENAME	Returns the name of the node. In the case of files, this is the node's short filename
\$\$NODEPATH	Returns the absolute path and filename of the selected node
\$\$CFLAGS	Returns the compilation flags
\$\$DLIBS	Returns the filenames of link definition libraries
\$\$LDFLAGS	Returns the flags used for link definition
\$\$ARGS	Returns all arguments sent to the command line
\$\$OUTDIR	Returns the name of the product features output directory
\$\$REPORTDIR	Returns name of the text report output directory
\$\$TARGETDIR	Returns the absolute path to the current Target Deployment Port
\$\$BINDIR	Returns the binary directory where the product is installed
\$\$OBJECTS	Returns a list of .o or .obj object files generated by the compiler
\$\$TIO	Returns the name of the current .tio trace file generated by Code Coverage
\$\$TSF	Returns the name of the current UML/SD .tsf static file generated by Runtime Tracing
\$\$TDF	Returns the name of the current UML/SD .tdf dynamic file generated by Runtime Tracing

\$\$TDC	Returns the name of the current Code Coverage .tdc correspondence file
\$\$ROD	Returns the name of the current .rod report file
\$\$FDC	Returns the name of the current .fdc correspondence files for Code Coverage

Instrumentation Pragma

The Runtime Tracing feature allows the user to add special directives to the source code under test, known as *pragma* directives. When the source code is instrumented, the Instrumentor replaces *pragma* directives with dedicated code.

Usage

```
#pragma attol <pragma name> <directive>
```

Example:

```
int f ( int a )
{
#pragma attol att_insert  if ( a == 0 ) _ATT_DUMP_STACK
    return a;
}
```

This code will be replaced, after instrumentation, with the following line:

```
/*#pragma attol att_insert*/  if ( a == 0 )
    _ATT_DUMP_STACK
```

Note Pragma directives are implemented only if the routine in which it is used is instrumented.

Instrumentation Pragma Names

```
#pragma attol insert <directive>
```

replaced by the instrumentation to be:


```
/*#pragma attol insert*/ <directive>
```

if any of Code Coverage, Runtime Tracing, Memory Profiling or Performance Profiling is/are selected.

```
#pragma attol atc_insert <directive>
```

replaced by the instrumentation to be:

```
/*#pragma attol atc_insert*/ <directive>
```

if Code Coverage is selected.

```
#pragma attol att_insert <directive>
```

replaced by the instrumentation to be:

```
/*#pragma attol att_insert*/ <directive>
```

if Runtime Tracing is selected.

```
#pragma attol atp_insert <directive>
```

replaced by the instrumentation to be:

```
/*#pragma attol atp_insert*/ <directive>
```

if Memory Profiling is selected.

```
#pragma attol atq_insert <directive>
```

replaced by the instrumentation to be:

```
/*#pragma attol atq_insert*/ <directive>
```

if Performance Profiling is selected.

Code Coverage, Memory Profiling and Performance Profiling Directives

```
_ATCPQ_DUMP(<reset>)
```

where <reset> is 1 if internal tables reset is wanted or 0 if not.

This macro **ATCPQ_DUMP** does nothing if Code Coverage, Memory Profiling, or Performance Profiling are not selected.

Runtime Tracing Directives

When using this mode, the Target Deployment Package only sends messages related to instance creation and destruction, or user notes. All other events are ignored. See Partial message dump for more information about this feature.

```
_ATT_START_DUMP
```

```
_ATT_STOP_DUMP
```

These directives activate and deactivate the partial message dump mode.

```
_ATT_TOGGLE_DUMP
```

This directive toggles the dump mode on and off. **_ATT_TOGGLE_DUMP** can be used instead of **_ATT_START_DUMP** and **_ATT_STOP_DUMP**.

```
_ATT_DUMP_STACK
```

When invoked, this directive dumps the contents of the call stack at that moment.

```
_ATT_FLUSH_ITEMS
```

When in Target Deployment Package buffer mode, this directive flushes the buffer. All buffered trace information is dumped. Flushing the buffer be useful before entering a time-critical phase of the trace.

```
_ATT_USER_NOTE (<text>)
```

This directive associates a text note to the function or method instance. *<text>* is a user-specified alphanumeric string containing the note text of type *char**. The length of *<text>* must be within the maximum note length

specified in the Runtime Tracing Settings dialog box.

Environment Variables

Mandatory Environment Variables

The following environment variables **MUST** be set to run the product:

- **TESTRTDIR** for the graphical user interface
- **ATUDIR** for Component Testing for C and Ada
- **ATS_DIR** for System Testing for C
- **ATLTGT** in the command line interface

Environment Variable List

Environment Variable	Description
TESTRTDIR	A mandatory environment variable that points to the installation directory of the product.
ATTOLSTUDIO_VERBOSE	Setting this variable to 1 forces the product GUI to display verbose messages, including file paths, in the Build Message Window.

Runtime Analysis Features

The Runtime Analysis Features use the following environment variables:

Environment Variable	Description
ATLTGT	A mandatory environment variable that points to the Target Deployment Port directory when you are using the product in the command line interface. When you are using the Instrumentation Launcher or the product GUI, you do not need to set ATLTGT manually, as it is calculated

automatically.

ATL_TMP_DIR	Indicates the location for temporary files. By default, they are placed in /tmp for UNIX or the current directory for Windows.
ATL_EXT_SRC	This variable allows you to instrument additional files with filename extensions other than the defaults (.c and .i). The .c extension is reserved for C source files that require preprocessing, while .i is for already preprocessed files. All other extensions supported by this variable are assumed to be of source files that need to be preprocessed.
ATL_EXT_OBJ	Lets you specify an alternative extension to .o (UNIX) or .obj (DOS) for object files.
ATL_EXT_ASM	Lets you specify more than .s extension for assembler source files when the compiler offers an option to generate an assembler listing without compiling it to the object file.
ATL_EXT_TMP_CMD	Windows only. Lets you specify an alternative extension to the Windows temporary options file. Defaults to ._@@ .
ATL_EXT_SRCCP	The variable lets you add C++ source file extensions (defaults are .C , .cpp , .c++ , .cxx , .cc , and .i) to specify the C++ source files to be instrumented. Extensions .C to .cc in the list are reserved for source files under analysis. The .i extension is reserved for those to be processed, if the ATL_FORCE_CPLUSPLUS variable is set to ON . Any other extension implies that pre-processing is to be performed.
ATL_FORCE_CPLUSPLUS	If set to ON , this variable allows you to force C++ instrumentation whether the file extension is .c , .i , or any added extension.

Component Testing for C and Ada

Component Testing for C and Ada uses the following environment variables:

Environment Variable	Description
-----------------------------	--------------------

ATUDIR	Points to the /lib directory in the product installation directory.
---------------	--

ATTUTGT

Points to the Target Deployment Port directory for Component Testing for C and Ada.

You can change default extensions for Component Testing for C and Ada through the use of environment variables when the Test Script Compiler or Test Report Generator is started.

The following table summarizes these environment variables and the extensions they modify.

Environment Variable	File	Default extension
ATTOLPTU	Test script	.ptu
ATTOLTDC	Table of correspondence file	.tdc
ATTOLLIS	List of errors	.lis
ATTOLRIO	Trace file	.rio
ATTOLRO	Test report	.ro
ATTOLROD	Unformatted test report	.rod
ATTOLDEF	Standard definitions file	.def
ATTOLSMB	Symbol table file	.smb

The rule whereby a "2" is added to the extension of the **.rio** trace file when the **-compare** option is used still applies if the default extension is changed in the **ATTOLRIO** environment variable.

System Testing for C

System Testing for C uses the following environment variable:

Environment Variable	Description
ATS_DIR	Points to the directory containing the System Testing binaries for C.

Test Process Monitor

The Test Process Monitor uses the following environment variables.

Environment Variable	Description
ATTOL_TPM_ROOT	This variable indicates the directory where Test Process Monitor databases are located for a project. ATTOL_TPM_ROOT is a mandatory variable and must be set when a project is created. It should be a shared directory accessible by all users who work on a project.
ATTOL_TPM_USER	This optional variable specifies the name of the user. If this variable is not set, the Test Process Monitor uses the current user, if possible.

C and C++ Instrumentation Launcher

The Instrumentation Launcher uses the following additional variables:

Environment Variable	Description
ATTOLBIN	If set, this variable must contain the path to the Instrumentor binaries. If not, this path is determined automatically from the PATH variable. This variable can be useful if the Target Deployment Port has been moved to a non-standard location.
ATTOLOBJ	If set, this variable points to a valid directory where the products.h file is generated and the Target Deployment Port (TP.o or TDP.obj) is compiled. By default, these files are generated in the current directory.
ATL_OVER_SET	This variable must indicate the path to a copy of the BatchCCDefaults.pl file if you want to change any Target Deployment Port compilation flags contained in that file.
ATL_EXT_LIB	Lets you specify additional alternative extensions for library files. By default .a or .lib are used.
ATL_FORCE_C_TDP	If set to ON , the & file is used instead of the tpcpp.ini file (used for C++ language). If the Target Deployment Port supports only C language, the tp.ini file is always used.

ATL_OVER_SET As an alternative to using the --settings of the Instrumentation Launcher, you can copy and modify the `<InstallDir>/lib/scripts/BatchCCDefaults.pl` file. In this case, set **ATL_OVER_SET** to the directory and filename of the new copy of this file.

Ada Tools

The Ada Link File Generator and Ada Unit Maker use the following additional variables:

Environment Variable	Description
ATTOLCHOP	Selects the default naming convention. The following values can be used: ATTOLCHOP="APEX" : for Rational Apex naming ATTOLCHOP="GNAT" : for Gnat naming All other values end with a fatal error. By default, Gnat naming is used.

ATTOLALK_EXT	Specifies allowed extensions separated by the semicolon (':') character on UNIX systems and (';') on Windows. By default, the allowed extension list is ".ada:.ads:.adb"
ATTOLALK_NOEXT	Specifies forbidden extensions separated by the ':' character on UNIX systems and ';' on Windows. By default, the forbidden extension list is empty.
LD_LIBRARY_PATH	Specifies the location of libraries required by the Ada Link File Generator. By default, these libraries are located in the /lib directory of the installation directory.

Setting Environment Variables

Solaris, Linux or HP-UX Platforms

To set an environment variable with a csh shell:

1. Open a shell window
2. Type the following command:
`setenv <variable> <value>`

To set an environment variable with a sh, ksh, or Bourne shell:

1. Open a shell window
2. Type the following commands:
`<variable>=<value>`
`export <variable>`

Windows Platforms

To set an environment variable under Windows NT, 2000 or XP:

1. From the **Start** menu, select **Parameters, Control Panel**, and double-click **System**.

2. Select the **Advanced** tab and click **Environment variables**.
3. Click the **New...** button to add the new environment variable.
4. Click **OK**.

File Types

This table summarizes all the file types generated and used by GUI.

File Type	Default Extension	Generated By	Used By
Component Testing for C++ Declaration Files	.dcl	C++ Source code Parser*	C++ Test Script Compiler
Code Coverage Correspondence File	.fdc	Instrumented application (Code Coverage)	Code Coverage Report Generator
Metrics File	.met	GUI	GUI Metrics Viewer
Component Testing for C++ Contract Check Script	.otc	C++ Source code Parser*	C++ Test Script Compiler
Component Testing for C++ Test Driver Script	.otd	C++ Source code Parser*	C++ Test Script Compiler
Component Testing for C++ Instrumentation File	.oti	C++ Test Script Compiler	C and C++ Instrumentor
Component Testing for C++ Intermediate File	.ots	C++ Test Script Compiler	C++ Test Report Generator
System Testing for C Test Script	.pts	User	System Testing Script Compiler
Component Testing for C and Ada Test Script	.ptu	C Source code Parser*	C Test Script Compiler

System Testing for C Result File	.rio	Test Driver (System Testing for C)	System Testing Report Generator
Component Testing for C and Ada Result File		Test Driver (Component Testing for C and Ada)	C Test Report Generator Ada Test Report Generator
Project File	.rtp	GUI	GUI
Workspace File	.rtw	GUI	GUI
Graphic Report	.rtx	C and Ada Test Report Generator	GUI Report Viewer
System Testing for C Supervision Script	.spv	User (the command line interface) or Virtual Tester Deployment Wizard	System Testing for C Supervisor

Target Output File	.spt	Target Deployment Port	GUI
Component Testing for C++ Stub Files	.stb	C++ Source Code Parser*	C++ Test Script Compiler
System Testing for C Intermediate File	.tdc	System Testing Script Compiler	System Testing Report Generator
Component Testing for C and Ada Intermediate File		C Test Script Compiler	C Test Report Generator Ada Test Report Generator
UML/SD Dynamic Trace File	.tdf	Instrumented application (Runtime Tracing, Component Testing for C++ and Java)	GUI UML/SD Viewer
Code Coverage Intermediate File	.tio	Instrumented application (Code Coverage)	Code Coverage Report Generator
Memory Profiling Dynamic Trace File	.tpf	Instrumented application (Memory Profiling)	GUI Memory Profiling Viewer
Performance Profiling Dynamic Trace File	.tqf	Instrumented application (Performance Profiling)	GUI Performance Profiling Viewer
Static Trace File	.tsf	C++ Test Script Compiler C and C++ Instrumentor Java Test Report Generator	GUI UML/SD Viewer
Target Deployment Port Customization File	.xdp	TDP Editor	TDP Editor
XML Report File	.xrd	C Test Report Generator Ada Test Report	GUI Report Viewer

Generator
C++ Test Report
Generator
Java Test Report
Generator
System Testing
Report Generator

* Indicates files that are generated test script templates. Use these files to write your own test scripts.