

Rational® Test RealTime

USER GUIDE

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.

FLEXIm and GLOBEtrotter are trademarks or registered trademarks of GLOBEtrotter Software, Inc. Licensee shall not incorporate any GLOBEtrotter software (FLEXIm 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.

User Guide

Contents

Product Overview	13
About Online Documentation	14
Context-Sensitive Online Help	14
Finding Information	15
Printing from the Online Documentation.....	16
Documentation Updates and Feedback	17
Source Code Insertion	17
Estimating Instrumentation Overhead	18
Reducing Instrumentation Overhead.....	21
Information Modes	22
Generating SCI Dumps.....	23
Target Deployment Ports	25
Launching the TDP Editor.....	27
Reconfiguring a TDP for a Compiler or JDK.....	28
Unified Modeling Language	29
UML Sequence Diagrams.....	29
Model Elements and Relationships in Sequence Diagrams	30
Activations	30
Classifier Roles.....	31
Destruction Markers.....	33
Lifelines	34
Messages	36
Objects	38
Stimuli.....	41
Actions.....	43
Exceptions	43
Actors	44
Loops.....	45
Synchronizations	46
Notes	48
Upgrading from a Previous Version	48

Runtime Analysis	51
Using Runtime Analysis Features	51
Code Coverage	52
Coverage Types	53
Ada Coverage.....	54
Ada Block Coverage	54
Ada Call Coverage	58
Ada Condition Coverage	58
Ada Unit Coverage	62
Ada Link Files.....	64
Ada Additional Statements.....	66
C Coverage	67
C Block Coverage	67
C Call Coverage.....	70
C Condition Coverage	72
C Function Coverage	76
C Additional Statements.....	77
C++ Coverage	78
C++ Block Code Coverage	78
C++ Method Code Coverage	81
C++ Template Instrumentation.....	83
C++ Additional Statements	84
Java Coverage	84
Java Block Coverage	84
Java Method Coverage	88
Java Additional Statements.....	90
Code Coverage Viewer.....	90
About the Code Coverage Viewer.....	90
Source Report	93
Rates Report	95
Code Coverage Toolbar	95
Code Coverage Viewer Preferences.....	96
Code Coverage Dump Driver.....	96
Static Metrics.....	97
Static Metric Viewer	97
Static Metrics	98
Root Level File View.....	99
Root Level Object View	101
Halstead Metrics.....	103

V(g) or Cyclomatic Number.....	104
Metrics Viewer Preferences.....	105
Memory Profiling for C and C++	106
Memory Profiling Results for C and C++	106
Memory Profiling Errors	108
Freeing Freed Memory (FFM)	108
Freeing Unallocated Memory (FUM)	109
Late Detect Array Bounds Write (ABWL).....	109
Late Detect Free Memory Write (FMWL).....	110
Memory Allocation Failure (MAF)	111
Core Dump (COR)	111
Memory Profiling Warnings	111
Memory in Use (MIU).....	112
Memory Leak (MLK)	112
Memory Potential Leak (MPK).....	112
File in Use (FIU).....	113
Signal Handled (SIG).....	113
Memory Profiling User Heap in C and C++	114
Using the Memory Profiling Viewer	119
Memory Profiling Viewer Preferences.....	120
Memory Profiling for Java	121
Memory Profiling Results for Java.....	121
JVMPI Technology.....	124
Performance Profiling	126
Performance Profiling Results.....	127
Performance Profiling SCI Dump Driver.....	128
Performance Profiling Viewer Preferences.....	129
Using the Performance Profiling Viewer.....	129
Runtime Tracing.....	130
About Runtime Tracing	130
Understanding Runtime Tracing UML Sequence Diagrams	131
Runtime Tracing with a Test Node	133
Multi-Thread Support	134
Partial Trace Flush.....	135
Trace Item Buffer	136
Splitting Trace Files	137

Automated Testing	141
Using Test Features	141
Component Testing for C and Ada	142
C and Ada Testing Overview	143
Integrated, Simulated and Additional Files	143
Tester Configuration	145
Importing V2001A Component Testing Files	146
Options and Settings	147
Array and Structure Display	147
Initial and Expected Values	148
Test Script Compiler Macro Definitions	149
Pointers	150
Testing Pointers against Pointer Structure Elements	150
Pointer and Array Ambiguities	151
Testing an Array Whose Elements are Unions	151
Initializing Pointer Variables while Preserving the Pointed Value	153
Functions	153
Testing Main Functions	153
Functions Using a Variable Number of Parameters	154
Functions Taking void* Parameters	155
Functions Using const Parameters	155
Functions Containing Type Modifiers	156
Functions Using _inout Mode Arrays	157
Functions Taking char* Parameters	158
C and Ada Test Script	159
Ada	159
Ada Records with Discriminants	159
Separate Compilation	160
Generic Units	160
Unknown Values	161
Test Program Entry Point	161
Testing Generic Packages	163
Declaring Global Variables for Testing	163
Generating a Separate Test Harness	165
Test Script Modification	166
Testing Ada Tasks	166
Environments	168
About Environments	169
Declaring Environments	169
Environment Override	170

Specifying Parameters for Environments	172
Using Environments.....	172
Exceptions	173
Unexpected Exceptions	173
Overview	173
Declaring Parameters.....	173
Test Script Structure	174
Simulations	176
C and Ada Syntax Extensions.....	176
Creating Complex Stubs.....	178
Excluding a Parameter from a Stub.....	178
Sizing Stubs.....	179
Simulation of Generic Units	180
Stub Definition in C.....	181
Stub Simulation Overview	184
Stub Usage in Ada.....	185
Stub Usage in C.....	187
C and Ada Test Reports	188
Comparing Reports.....	188
Understanding Component Testing Reports	189
Understanding Component Testing UML Sequence Diagrams for C and Ada.....	191
Component Testing for C++.....	191
About Component Testing for C++.....	191
C++ Testing Overview	193
C++ Test Nodes.....	193
C++ Contract-Check Script.....	193
C++ Test Driver Script	194
Files and Classes Under Test.....	194
Simulated, Additional and Included Files.....	196
Declaration Files	198
C++ Test Reports	199
Understanding Component Testing for C++ Reports	199
Understanding Component Testing for C++ UML Sequence Diagrams	202
Illegal and Multiple Transitions	202
Contract-Check Sequence Diagrams	203
Test Driver Sequence Diagrams.....	204
Component Testing for Java.....	208
Java Testing Overview.....	209
Java Test Nodes.....	211

Java Test Harness	211
Using the TestCase Class	214
Using the TestResult Class	217
Using the TestSuite Class	218
Simulated and Additional Classes	219
Java Stubs	220
Importing a JUnit Test Campaign	221
J2ME Specifics	223
Java Test Reports	224
Understanding Java Test Reports	224
Understanding Java Component Testing UML Sequence Diagrams	226
System Testing for C	228
System Testing Overview	229
Circular Trace Buffer	229
System Testing Supervisor	230
Agents and Virtual Testers	231
System Testing Agents	231
Installing System Testing Agents	231
System Testing Agent Access Files	235
Configuring Virtual Testers	235
Debugging Virtual Testers	237
Deploying Virtual Testers	237
Editing the Deployment Script	239
Optimizing Execution Traces	240
Setting Up Rendezvous Members	241
System Testing in a Multi-Threaded or RTOS Environment	241
Virtual Tester Thread Starter Program	243
System Testing for C Test Scripts	244
Basic Structure	244
Include Statements	245
Procedures	246
Flow Control	247
Conditions	247
Iterations	248
Multiple Conditions	249
Native C	250
CALL Instruction	250
Using Native Language	250
Instances	251
Instance Declaration	251
Instance Synchronization	252

Instances	255
Environments	255
Error Handling.....	256
Exception Environment (Error Recovery Block)	257
Initialization Environment.....	258
Termination Environment	260
Time Management.....	261
TIME Instruction.....	262
TIMER Instruction	263
RESET Instruction	263
PRINT Instruction	264
PAUSE Instruction.....	264
Event Management.....	265
Basic Declarations	266
Sending Messages	267
Receiving Messages.....	269
Messages and Data Management.....	273
Communication Between Virtual Testers	279
Understanding System Testing for C Reports	280
Understanding System Testing UML Sequence Diagrams	282
Advanced System Testing for C.....	285
Trace Probes	285
Using Probe Macros	286
Generated Test Script.....	287
On-the-Fly Tracing.....	288

Graphical User Interface..... 291

Discovering the GUI.....	292
Start Page.....	293
Output Window	293
Project Explorer	294
Properties Window.....	296
Report Explorer.....	298
Standard Toolbars	298
Using the GUI Components	300
Report Viewer	300
Understanding Test and Runtime Analysis Reports.....	301
Setting a Zoom Level.....	302
Report Viewer Toolbar.....	302

Report Viewer Style Preferences	303
Text Editor	303
Creating a Text File	304
Opening a Text File	304
Finding Text in the Text Editor	305
Replacing Text in the Text Editor	306
Locating a Line and Column in the Text Editor	307
Text Editor Syntax Coloring	307
Text Editor Preferences	308
Tools Menu	309
Tool Configuration	310
Test Process Monitor	311
Changing Curve Properties	312
Custom Curves	313
Event Markers	314
Setting the Time Scale	315
Test Process Monitor Toolbar	315
Adding a Metric	316
UML/SD Viewer	317
Navigating through UML Sequence Diagram	317
Time Stamping	317
Coverage Bar	318
Memory Usage Bar	319
Thread Bar	320
Applying Filters	321
Sequence Diagram Triggers	322
Editing Trigger or Filter Events	324
Finding Text in a UML Sequence Diagram	327
Step-by-Step mode	327
UML/SD Viewer Toolbar	328
UML/SD Viewer Preferences	329
Configurations and Settings	331
Configurations and Settings	331
General Settings	334
Build Settings	335
External Command Settings	338
Probe Control Settings	339
Runtime Analysis Settings	340
General Runtime Analysis Settings	340
Memory Profiling Settings	343
Performance Profiling Settings	346

Code Coverage Settings.....	347
Runtime Tracing Control Settings.....	349
Automated Testing Settings.....	352
Component Testing Settings for C and Ada.....	352
Component Testing for C++ Settings.....	353
Component Testing for Java Settings.....	357
System Testing for C Settings.....	358
Selecting Configurations.....	361
Modifying Configurations.....	361
Working with Projects.....	363
Creating a Group.....	363
Manually Creating a Test or Application Node.....	363
Creating an External Command Node.....	364
Importing a Makefile.....	365
Refreshing the Asset Browser.....	366
Deleting a Node.....	367
Renaming a Node.....	367
Viewing File Properties.....	368
Excluding a Node from a Build.....	368
Adding Files to the Project.....	369
Selecting Build Options.....	369
Building and Running a Node.....	370
Cleaning Up Generated Files.....	371
Creating a Source File Folder.....	371
Importing a Data Table (.csv File).....	372
Opening a Report.....	373
Debug Mode.....	375
Editing Preferences.....	375
Project Preferences.....	375
Connection Preferences.....	376
Activity Wizards.....	376
New Project Wizard.....	377
Runtime Analysis Wizard.....	377
Component Testing Wizard.....	379
System Testing Wizard.....	382
Metrics Diagram.....	384
Advanced Options.....	385

Command Line Interface.....	389
Running a Node from the Command Line	389
Command Line Runtime Analysis for C and C++	390
Command Line Runtime Analysis for Java.....	392
Command Line Component Testing for C, Ada and C++	393
Command Line Component Testing for Java.....	394
Command Line System Testing for C.....	395
Command Line Tasks	397
Setting Environment Variables.....	397
Preparing an Options Header File.....	399
Preparing a Products Header File.....	400
Instrumenting and Compiling the Source Code.....	401
Compiling the TDP Library.....	402
Compiling the Test Harness.....	404
Linking the Application.....	405
Running the Test Harness or Application.....	405
Splitting the Trace Dump File.....	406
Troubleshooting Command Line Usage.....	407
Working with Other Development Tools.....	409
Working with Configuration Management.....	409
Working with Rational ClearCase	409
Working with Rational ClearQuest	411
CMS Preferences	412
ClearQuest Preferences	412
Customizing Configuration Management	413
Working with Rational Rose RealTime	413
Installing Rose RealTime Integration	413
Using the Product with Rose RealTime	414
Collecting Trace Dump Data.....	416
Viewing Results from Rose RealTime.....	417
Advanced Rose RealTime Integration	418
Working with Rational TestManager.....	422

Before Using the TestManager Integration.....	423
Installing TestManager Integration	424
Submitting a ClearQuest Defect from TestManager	425
Viewing Results in TestManager.....	425
Working with Rational TestManager	426
Working with Microsoft Visual Studio	427
Installing Microsoft Visual Studio Integration.....	427
Configuring Microsoft Visual Studio Integration.....	427
Technical Support.....	433
Glossary.....	437

Product Overview

1

Implementing a practical, effective and professional testing process within your organization has become essential because of the increased risk that accompanies software complexity. The time and cost devoted to testing must be measured and managed accurately. Very often, lack of testing causes schedule and budget over-runs with no guarantee of quality.

Critical trends require software organizations to be structured and to automate their test processes. These trends include:

- Ever increasing quality and time to market constraints;
- Growing complexity, size and number of software-based equipment;
- Lack of skilled resources despite need for productivity gains;
- Increasing interconnectedness of critical and complex embedded systems;
- Proliferation of quality & certification standards throughout critical software markets, including the avionics, medical, and telecommunications industries.

Rational Test RealTime provides a full range of answers to these challenges by enabling full automation of system and software test processes.

Test RealTime is a complete test and runtime analysis toolset for embedded, real-time and networked systems created in any cross-development environment. Automated testing, code coverage, memory leak detection, performance profiling, UML tracing - with Test RealTime you fix your code before it breaks.

Test RealTime covers runtime analysis and software testing, all in a fully integrated testing environment.

About Online Documentation

The entire documentation set for Test RealTime 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 4.7 or later on UNIX operating systems or any other Java-enabled web browser.

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-hand side of the Help window.

Context-Sensitive Online Help

As you work with the product, you can obtain information about windows or dialogs by using the context-sensitive Help available in the application. You can access this Help in several ways, including:

- **Dialog boxes:** Dialogs display **Help** buttons. Click the button and a topic opens in the Help browser window that explains how to use the fields and controls at the dialog. You can also press the **F1** key in dialog boxes to get help.
- **Windows:** From any window, press **F1** to get more information about using it.

- **Right-click menus:** Every right-click menu includes a **Help** option. For help on a specific item, right-click and select **Help**. You can also right-click inside windows for help.





Finding Information

There are several ways of navigating through the online help system to find information. These are available through the Navigation Pane at the left of this window.

- **Using the Navigation Pane:** The navigation pane is the frame on the left of this window which contains the Contents, the Index and the Search facilities.

On the Windows platform, click the button in the Windows Help browser to hide or show the Navigation Pane.

On UNIX platforms, use the **Hide Navigation Pane** or **Show Navigation Pane** links at the top of the Help page.

- **Contents:** The **Contents** tab displays books  and pages  that represent the categories of information in the online Help system. When you click a book, it opens the first page of its content. To expand or reduce book's contents, click the  or  buttons to the left of the book. When you click pages , you select topics to view in the right-hand pane of the HTML Help viewer.
- **Index:** The **Index** tab displays a multi-level list of keywords and keyword phrases. These terms are associated with topics in the Help system and they are intended to direct you to specific topics according to your way of working. Keywords are cross-referenced with synonyms to provide multiple ways to locate information. To open a topic in the right-hand pane associated with a keyword, select the keyword and then click **Display**. If the keyword is used with more than one topic, a Topics Found dialog opens so you can select a specific topic to view.
- **Search:** The **Search** tab enables you to search for words in the Help

system and locate topics containing those words. Full-text searching looks through every word in the online Help to find matches. When the search is completed, a list of topics is displayed so you can select a specific topic to view.

Printing from the Online Documentation

In some cases, you might find useful to print out portions of the documentation for reading off-line.

To print a topic in Windows:

1. Select a topic
2. In the toolbar, click the **Print** icon.

To print a topic in UNIX:

1. Select a topic
2. Use the **Print** command of your Web browser

To print an entire chapter in Windows:

1. In the navigation pane, select the Contents tab.
2. Select a book.
3. In the toolbar, click the **Print** icon.
4. Select Print the selected section and all subchapter and click OK.

Note Printing an entire chapter is not currently supported in the UNIX online help.

Documentation Updates and Feedback

Latest Updates

For the most recent documentation updates please visit the Product Support section of the following website:

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

Feedback

We do our best to provide you with the highest possible quality in our user documentation, and your feedback is essential for us to improve the standards 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.

Source Code Insertion

Rational's Source Code Insertion (SCI) technology uses instrumentation techniques that automatically adds special code to the source files under analysis. After compilation, execution of the code produces SCI dump data for the selected runtime analysis or automated testing features.

Rational Test RealTime makes extensive use of SCI technology to transparently produce test and analysis reports on both native and embedded target platforms.

Estimating Instrumentation Overhead

Instrumentation overhead is the increase in the binary size or the execution time of the instrumented application, which is due to Source Code Insertion (SCI) generated by the Runtime Analysis features.

Rational's SCI technology is designed to reduce both types of overhead to a bare minimum. However, this overhead may still impact your application.

The following table provides a quick estimate of the overhead generated by the product.

Code Coverage Overhead

Overhead generated by the Code Coverage feature depends largely on the coverage types selected for analysis.

A 48 byte structure is declared at the beginning of the instrumented file.

Depending on the information mode, each branch is referenced by a 1 byte (pass mode), 1 bit (compact mode) or 4 byte (count mode) array.

The size of this array may be rounded up by the compiler (especially in compact mode because of the 8 bit minimum integral type found in C/C++).

Other Specifics:

- loops, switch, case statements: a 1 byte local variable is declared for each instance
- (not forced) modified/multiple conditions: a n byte local array is declared at the beginning of the enclosing routine, where n is the number of conditions belonging to a decision in the routine

I/O is either performed at the end of the execution or when the end-user decides (please refer to Coverage Snapshots in the documentation).

In conclusion:

Count mode and modified/multiple conditions have the greatest data and execution time overhead. In most cases, it is recommended that coverage types be independently selected and pass mode be used as the default. Source code can also be partially instrumented. Compact mode is helpful when data space is lacking, but there is still an unavoidable increase in code size (shift/bits masks) and execution time.

Memory and Performance Profiling and Runtime Tracing

Any source file containing an instrumented routine receives a declaration for a 16 byte structure.

Within each instrumented routine, a n byte structure is locally declared, where n is:

16 bytes

+4 bytes for Runtime Tracing

+4 bytes for Memory Profiling

+ $3*t$ bytes for Performance Profiling, where t is the size of the type returned by the clock-retrieving function

For example, if t is 4 bytes, each instrumented routine is increased of:

- 20 bytes for Memory Profiling only
- 20 bytes for Runtime Tracing only
- 28 bytes for Performance Profiling only
- 36 bytes for all Runtime Analysis features together

Memory Profiling Overhead

Note This applies to Memory Profiling for C, C++ and Ada. Memory Profiling for Java does not use source code insertion.

Any call to an allocation function is replaced by a call to the Memory Profiling Library. See the **Target Deployment Guide** for more information.

These calls aim to track allocated blocks of memory. For each memory block, $16+12*n$ bytes are allocated to contain a reference to it, as well as to contain link references and the call stack observed at allocation time. n depends on the Call Stack Size Setting, which is 6 by default.

If ABWL errors are to be detected, the size of each tracked, allocated block is increased by $2*s$ bytes where s is the Red Zone Size Setting (16 by default).

If FFM or FMWL errors are to be detected, a Free Queue is created whose size depends on the Free Queue Length and Free Queue Size Settings. Queue Length is the maximum number of tracked memory blocks in the queue. Queue Size is the maximum number of bytes, which is the sum of the sizes of all tracked blocks in the queue.

Performance Profiling Overhead

For any source file containing at least one observed routine, a 24 byte structure is declared at the beginning of the file.

The size of the global data storing the profiling results of an instrumented routine is $4+3*t$ bytes where t is the size of the type returned by the clock retrieving function. See the **Target Deployment Guide** for more information.

Runtime Tracing Overhead

Implicit default constructors, implicit copy constructors and implicit destructors are explicitly declared in any instrumented classes that permits it.

Where C++ rules forbid such explicit declarations, a 4 byte class is declared as an attribute at the end of the class.

Reducing Instrumentation Overhead

Rational's Source Code Insertion (SCI) technology is designed to reduce both performance and memory overhead to a minimum. Nevertheless, for certain cross-platform targets, it may need to be reduced still further. There are three ways to do this.

Limiting Code Coverage Types

When using the Code Coverage feature, procedure input and simple and implicit block code coverage are enabled by default. You can reduce instrumentation overhead by limiting the number of coverage types.

Note The Code Coverage report can only display coverage types among those selected for instrumentation.

Instrumenting Calls (C Language)

When calls are instrumented, any instruction that calls a C user function or library function constitutes a *branch* and thus generates overhead. You can disable call instrumentation on a set of C functions using the Selective Code Coverage Instrumentation Settings.

For example, you can usually exclude calls to standard C library functions such as **printf** or **fopen**.

Code Coverage Information

In C++, use compact mode to decrease the data size overhead for targets where the code size is less critical.

In Ada, you can use pass mode to reduce the data amount overhead in the instrumented program. When using CLI mode, you can also use the -**instrumentation** option of the Instrumentor command line.

Information Modes

The Information Mode is the method used by Code Coverage to code the trace output. This has a direct impact of the size of the trace file as well as on CPU overhead.

You can change the information mode used by Code Coverage in the Coverage Type settings. There are three information modes:

- Default mode
- Compact mode
- Hit Count mode

Default Mode

When using **Default** or **Pass** mode, each branch generates one byte of memory. Which offers the best compromise between code size and speed overhead.

Compact Mode

The **Compact** mode is functionally equivalent to Pass mode, except that each branch needs only one bit of storage instead of one byte. This implies a smaller requirement for data storage in memory, but produces a noticeable increase in code size (shift/bits masks) and execution time.

Hit Count Mode

In **Hit Count** mode, instead of storing a Boolean value indicating coverage of the branch, a specific count is maintained of the number of times each branch is executed.

This information is displayed in the Code Coverage report. Count totals are given for each branch, for all trace files transferred to the report generator as parameters.

In the Code Coverage report, branches that have never been executed are highlighted with asterisk '*' characters.

The maximum count in the report generator depends on the machine on which tests are executed. If this maximum count is reached, the report signals it with a **Maximum reached** message.

Generating SCI Dumps

By default, the system call `atexit()` or `on_exit()` invokes the Target Deployment Port (TDP) function that dumps the trace data. You can therefore instrument either all or a portion of the application as required.

When instrumenting embedded or specialized applications that never terminate, it is sometimes impractical to generate a dump on the `atexit()` or `on_exit()` functions. If you exit such applications unexpectedly, traces may not be generated. In this case, you must either:

- Specify one or several explicit dump points in your source code, or
- Use an external signal to call a dump routine, or
- Produce an snapshot when a specific function is encountered.

Explicit Dump

Code Coverage, Memory Profiling and Performance Profiling allow you to explicitly invoke the TDP dump function by inserting a call to the `_ATCPQ_DUMP(<int>)` macro definition, where `<int>` is either 0 or 1.

- Use `_ATCPQ_DUMP(1)` to reset the internal trace table.
- Use `_ATCPQ_DUMP(0)` to preserve the internal trace table. This produces redundant information.

Explicit dumps should not be placed in the main loop of the application. The best location for an explicit dump call is in a secondary function, for example called by the user when sending a specific event to the application.

The explicit dump method is sometimes incompatible with watchdog constraints. If such incompatibilities occur, you must:

- Deactivate any hardware or software watchdog interruptions
- Acknowledge the watchdog during the dump process, by adding a specific call to the Data Retrieval customization point of the TDP.

Dump on Signal

Code Coverage allows you to dump the traces at any point in the source code by using the `ATC_SIGNAL_DUMP` environment variable.

When the signal specified by `ATC_SIGNAL_DUMP` is received, the Target Deployment Port function dumps the trace data and resets the signal so that the same signal can be used to perform several trace dumps.

Before starting your tests, set `ATC_SIGNAL_DUMP` to the following value:

```
<number>[:0|1]
```

where `<number>` is the number of the signal that is to trigger the trace dump.

The second parameter (**0** or **1**) after the separator character indicates whether the internal tables should be reset, so as to generate separate traces for successive independent tests (parameter 1) or cumulative traces (parameter **0**). For example:

```
16:0
17:1
```

The signal must be redirectable signal, such as **SIGUSR1** or **SIGINT** for example.

Instrumentor Snapshot

The Instrumentor snapshot option enables you to specify the functions of your application that will dump the trace information on entry, return or call.

In snapshot mode, the Runtime Tracing feature starts dumping messages only if the Partial Message Dump setting is activated. Code Coverage, Memory Profiling and Performance Profiling features all dump their internal trace data.

Use the **_ATCPQ_RESET** macro definition to specify whether the internal table reset must be done. By default, the **_ATCPQ_RESET** value is **1** (reset will be done). If you do not want to reset the tables, you must insert the compiler option to set this macro value to **0**.

Target Deployment Ports

Rational's Target Deployment Technology is a versatile, low-overhead technology enabling target-independent tests and run-time analysis despite limitless target support. Used by all Test RealTime features, the Target Deployment Port (TDP) technology is constructed to accommodate your compiler, linker, debugger, and target architecture. Tests are independent of the TDP, so tests don't change when the environment does. Test script

deployment, execution and reporting remain easy to use.

Key Capabilities and Benefits

- Compiler dialect-aware and linker-aware, for transparent test building.
- Easy download of the test harness environment onto the target via the user's IDE, debugger, simulator or emulator.
- Painless test and run-time analysis results download from the target environment using JTAG probes, emulators or any available communication link, such as serial, Ethernet or file system.
- Powerful test execution monitoring to distribute, start, synchronize and stop test harness components, as well as to implement communication and exception handling.
- Versatile communication protocol adaptation to send and receive test messages.
- XML-based TDP editor enabling simple, in-house TDP customization

Downloading Target Deployment Ports

Target Deployment technology was designed to adapt to any embedded or native target platform. This means that you need a particular TDP to deploy Test RealTime to your target.

A wide array of TDPs has already been developed by Rational to suit most target platforms. You can freely download available TDPs from the following page:

<http://www.rational.com/products/testrt/tdport/index.jsp>

Alternatively, from the **Help** menu, select **Download Target Deployment Ports**.

Downloaded TDPs can be freely used and modified with the TDP Editor.

Obtaining New Target Deployment Ports

If there is no existing TDP for your particular target platform, you have two options:

- You can choose to create, unassisted, a TDP tailored for your embedded environment. This requires extensive knowledge of your development environment and the product. This also requires some knowledge of the scripting language Perl.
- Rational can provide Professional Services and create a tailored TDP for you.

To create a TDP, see the Target Deployment Guide provided with the TDP Editor. The Target Deployment Guide provides an overview and detailed information on setting up a TDP, and using the TDP Editor.

For Rational's Professional Services, please contact Rational via one of these methods:

- Contact your Rational Sales Representative directly.
- Submit a contact request via this link:
http://www.rational.com/products/testrt/forms/test_rt.jsp
- If you don't know your Sales Representative, contact Rational Customer Support.

Launching the TDP Editor

The TDP Editor provides a user interface designed to help you customize and create unified Target Deployment Ports (TDP).

Please refer to the **Target Deployment Guide**, accessible from the **Help** menu of the Target Deployment Port Editor, for information about customizing Target Deployment Ports and using the editor.

To run the TDP Editor from Windows:

- From the Windows Tools menu, select Target Deployment Port Editor and Start.

Updating a Target Deployment Port

The Target Deployment Port (TDP) settings are read or loaded when a Test RealTime project is opened, or when a new TDP is used.

If you make any changes to a TDP with the TDP Editor, these will not be taken into account until the TDP has been reloaded in the project.

To reload the TDP in Test RealTime:

1. From the **Project** menu, select **Configurations**.
2. Select the TDP and click **Remove**.
3. Click **New**, select the TDP and click **OK**.

Reconfiguring a TDP for a Compiler or JDK

During installation of Rational Test RealTime:

- **on Windows:** A local Microsoft Visual Studio compiler and JDK are located, based on registry settings. Only the compiler and JDK located during installation will be accessible within Test RealTime.
- **on Unix platforms:** The user is confronted by two interactive dialogs. These dialogs serve to clarify the location of the local GNU compiler and (if present) local JDK. Only the GNU compiler and JDK specified within these dialogs will be accessible within Test RealTime.

To make a different compiler or JDK accessible in Test RealTime:

1. From the Tools menu, select the Target Deployment Port Editor and Start.
2. In the TDP Editor, from the **File** menu, select **Open**.

3. Open the **.xdp** file corresponding to the new compiler or JDK for which you would like to generate support
4. From the **File** menu, select **Save**.
5. Close the TDP Editor

To update an existing project to use the newly supported compiler or JVM:

1. Open the existing project in Test RealTime.
2. From the **Project** menu, select **Configuration**.
3. In the **Configurations** window, click **New**.
4. In the **New Configuration** window, select the newly supported compiler or JDK in the dropdown list and click **OK**.
5. In the **Configurations** window, click **Close**.

Unified Modeling Language

UML Sequence Diagrams

A sequence diagram is a Unified Modeling Language (UML) diagram that provides a view of the chronological sequence of messages between instances (objects or classifier roles) that work together in an interaction or interaction instance. A sequence diagram consists of a group of instances (represented by lifelines) and the messages that they exchange during the interaction. You line up instances participating in the interaction in any order from left to right, and then you position the messages that they exchange in sequential order from top to bottom. Activations sometimes appear on the lifelines.

A sequence diagram belongs to an interaction in a collaboration or an interaction instance in a collaboration instance.

Model Elements and Relationships in Sequence Diagrams

The UML sequence diagrams produced by the UML/SD Viewer illustrate program interactions with an emphasis on the chronological order of messages.

Activations

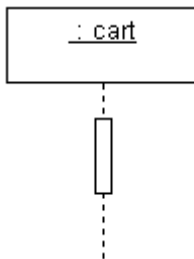
An activation (also known as a focus of control) is a notation that can appear on a lifeline to indicate the time during which an instance (an actor instance, object, or classifier role) is active. An active instance is performing an action, such as executing an operation or a subordinate operation. The top of the activation represents the time at which the activation begins, and the bottom represents the time at which the activation ends.

For example, in a sequence diagram for a "Place Online Order" interaction, there are lifelines for a ":Cart" object and ":Order" object. An "updateTotal" message points from the ":Order" object to the ":Cart" object. Each lifeline has an activation to indicate how long it is active because of the "updateTotal" message.

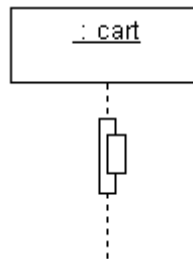
Shape

An activation appears as a thin rectangle on a lifeline. You can stack activations to indicate nested stack frames in a calling sequence.

Activation



Nested Activations



Using Activations

Activations can appear on your sequence diagrams to represent the following:

- On lifelines depicting instances (actors, classifier roles, or objects), an activation typically appears as the result of a message to indicate the time during which an instance is active.
- On lifelines involved in complex interactions, nested activations (also known as stacked activations or nested focuses of control) are displayed to indicate nested stack frames in a calling sequence, such as those that happen during recursive calls.
- On lifelines depicting concurrent operations, the entire lifeline may appear as an activation (thin rectangles) instead of dashed lines.

Naming Conventions

An activation is usually identified by the incoming message that initiates it. However, you may add text labels that identify activations either next to the activation or in the left margin of the diagram.

Classifier Roles

A classifier role is a model element that describes a specific role played by a classifier participating in a collaboration without specifying an exact instance of a classifier. A classifier role is neither a class nor an object. Instead, it is a model element that specifies the kind of object that must ultimately fulfill the role in the collaboration. The classifier role limits the kinds of classifier that can be used in the role by referencing a base classifier. This reference identifies the operations and attributes that an instance of a classifier will need in order to fulfill its responsibilities in the collaboration.

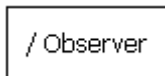
Classifier roles are commonly used in collaborations that represent patterns.

For example, a subject-observer pattern may be used in a system. One classifier role would represent the subject, and one would represent the observer. Each role would reference a base class that identifies the attributes and operations that are needed to participate in the subject-observer collaboration. When you use the pattern in the system, any class that has the specified operations and behaviors can fill the role.

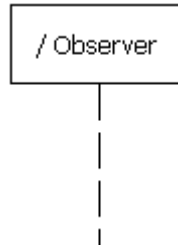
Shape

A classifier role appears as a rectangle. Its name is prefixed with a slash and is not underlined. In sequence diagrams, a lifeline (a dashed, vertical line) is attached to the bottom of a classifier role to represent its life over a period of time. For details about lifelines, see Lifelines.

Classifier Role



Classifier Role with Lifeline



Using Classifier Roles

You can add classifier roles to your model to represent the following:

- In models depicting role-based interactions, a classifier role represents an instance in an interaction. Using classifier roles instead of objects can provide two advantages: First, a class can serve as the base classifier for multiple classifier roles. Second, instances of a class can realize multiple classifier roles in one or more collaborations.
- In models depicting patterns, a classifier role specifies the kind of object that must ultimately fulfill a role in the pattern. The classifier role shows

how the object will participate in the pattern, and its reference to a base class defines the attributes and operations that are required for participation in the pattern. When the pattern is used in the model, classes are bound to the collaboration to identify the type of objects that realize the classifier roles.

The classifier roles in a model are usually contained in a collaboration and usually appear in sequence diagrams.

Naming Conventions

The name of a classifier role consists of a role name and base class name. You can omit one of the names. The following table identifies the variations of the naming convention.

Convention	Example	Description
/rolename:baseclass	/courseOffering:course	The courseOffering role is based on the course class.
/rolename	/courseOffering	Role name. The base class is hidden or is not defined.
:baseclass	:course	Unnamed role based on the course class.

Destruction Markers

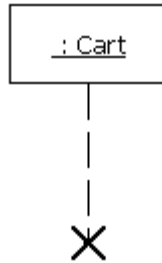
A destruction marker (also known as a termination symbol) is a notation that can appear on a lifeline to indicate that an instance (object or classifier role) has been destroyed. Usually, the destruction of an object results in the memory occupied by the data members of the object being freed.

For example, when a customer exits the Web site for an e-commerce application, the ":Cart" object that held information about the customer's activities is destroyed, and the memory that it used is freed. The destruction of the ":Cart" object can be shown in a sequence diagram by adding a

destruction marker on the ":Cart" object's lifeline.

Shape

A destruction marker appears as an X at the end of a lifeline.



Naming Conventions

Destruction markers do not have names.

Lifelines

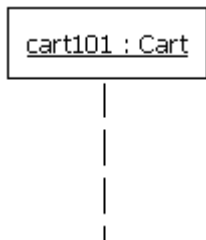
A lifeline is a notation that represents the existence of an object or classifier role over a period of time. Lifelines appear only in sequence diagrams, where they show how each instance (object or classifier role) participates in the interaction.

For example, a "Place Online Order" interaction in an e-commerce application includes a number of lifelines in a sequence diagram, including lifelines for a ":Cart" object, ":OnlineOrder" object, and ":CheckoutCart" object. As the interaction is developed, stimuli are added between the lifelines.

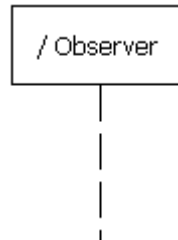
Shape

A lifeline appears as a vertical dashed line in a sequence diagram.

Lifeline for an Object



Lifeline for a Classifier Role



Using Lifelines

When you add a classifier role or object to a sequence diagram, it will automatically have a lifeline. You can use lifelines to indicate the following:

- Creation – If an instance is created during the interaction, its lifeline starts at the level of the message or stimulus that creates it; otherwise, its lifeline starts at the top of the diagram to indicate that it existed prior to the interaction.
- Communication – Messages or stimuli between instances are illustrated with arrows. A message or stimulus is drawn with its end on the lifeline of the instance that sends it and its arrowhead on the lifeline of the instance that receives it.
- Activity – The time during which an instance is active (either executing an operation directly or through a subordinate operation) can be shown with activations.
- Destruction – If an instance is destroyed during the interaction, its lifeline ends at the level of the message or stimulus that destroys it, and a destruction marker appears; otherwise, its lifeline extends beyond the final message or stimulus to indicate that it exists during the entire interaction.

Naming Conventions

A lifeline has the name of an object or classifier role. For details, see Objects or Classifier Roles.

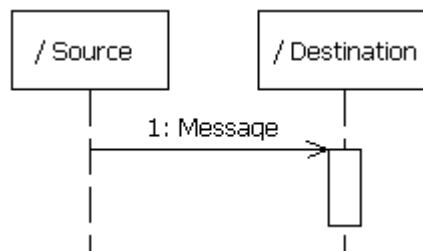
Messages

A message is a model element that specifies a communication between classifier roles and usually indicates that an activity will follow. The types of communications that messages model include calls to operations, signals to classifier roles, the creation of classifier roles, and the destruction of classifier roles. The receipt of a message is an instance of an event.

For example, in the observer pattern, the instance that is the subject sends an "Update" message to instances that are observing it. You can illustrate this behavior by adding "Subject" and "Observer" classifier roles and then adding an "Update" message between them.

Shape




A message appears as a line with an arrow. The direction of the arrow indicates the direction in which the message is sent. In a sequence diagram, messages usually connect two classifier role lifelines.



Message shapes can be adorned with names and sequence numbers.

Types of Messages

Different types of messages can be used to model different flows of control.

Type	Shape	Description
Procedure Call or Nested Flow of Control		Models either a call to an operation or a call to a nested flow of control. When calling a nested flow of control, the system waits for the nested flow of control to complete before continuing with the outer flow.
Asynchronous Flow of Control		Models an asynchronous message between two objects. The source object sends the message and immediately continues with the next step.
Return From a Procedure Call		Models a return from a call to a procedure. This type of message can be omitted from diagrams because it is assumed that every call has a return.

Using Messages

You can add messages to your model to represent the communications exchanged between classifier roles during dynamic interactions.

Note Both messages and stimuli are supported. Stimuli are added to collaboration instances, and messages are added to collaborations. For details about stimuli, see Stimuli.

The messages in a model are usually contained in collaborations and usually appear in sequence diagrams.

Naming Conventions

Messages can be identified by a name or operation signature.

Type	Example	Description
Name	// Get the Password	A name identifies only the name of the message. Simple names are often used in diagrams developed during analysis because the messages are identified by their responsibilities and not operations. One convention uses double slashes (//) to indicate that the stimulus name is not associated with an operation.

Signature getPassword(String)

When an operation is assigned to a message, you can display the operation signature to identify the name of the operation and its parameters. Signatures are often used in diagrams developed during design because they provide the detail that developers need when they code the design.

Objects

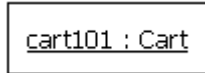
An object is a model element that represents an instance of a class. While a class represents an abstraction of a concept or thing, an object represents an actual entity. An object has a well-defined boundary and is meaningful in the application. Objects have three characteristics: state, behavior, and identity. State is a condition in which the object may exist, and it usually changes over time. The state is implemented with a set of attributes. Behavior determines how an object responds to requests from other objects. Behavior is implemented by a set of operations. Identity makes every object unique. The unique identity lets you differentiate between multiple instances of a class if each has the same state.

The behaviors of objects can be modeled in sequence and activity diagrams. In sequence diagrams, you can display how instances of different classes interact with each other to accomplish a task. In activity diagrams, you can show how one or more instances of an object changes states during an activity. For example, an e-commerce application may include a "Cart" class. An instance of this class that is created for a customer visit, such as "cart100:Cart." In a sequence diagram, you can illustrate the stimuli, such as "addItem()," that the "cart100:Cart" object exchanges with other objects. In an activity diagram, you can illustrate the states of the "cart100:Cart" object, such as empty or full, during an activity such as a user browsing the online catalog.

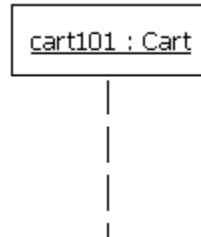
Shape

In sequence and activity diagrams, an object appears as a rectangle with its name underlined. In sequence diagrams, a lifeline (a dashed, vertical line) is attached to the bottom of an object to represent the existence of the object over a period of time. For details about lifelines, see Lifelines.

Object

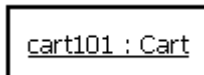


Object with Lifeline

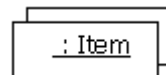


There are two notable variations of the object shape. First, active objects appear with thicker borders than other types of objects. Second, multiobjects appear as two overlapped rectangles. (These types of objects are defined later in this topic.)

Active Object



Multiobject



In addition, the object shape may include adornments for properties, such as persistence and concurrency. It may display a stereotype with an icon or the display of the stereotype name in guillemets (« »). Finally, it may show an attribute compartment. In activity diagrams, an object shape can display the state of the object under the name.

Types of Objects

The following table identifies three types of objects.

Types of Objects	Description
------------------	-------------

Active	Owens a thread of control and may initiate control activity. Processes and tasks are kinds of active objects.
Passive	Holds data, but does not initiate control.
Multiobject	Is a collections of object or multiple instances of the same class. It is commonly used to show that a set of objects interacts with a single stimulus.

Using Objects

You can add objects to your model to represent concrete and prototypical instances. A concrete instance represents an actual person or thing in the real world. For example, a concrete instances of a "Customer" class would represent an actual customer. A prototypical instance represents an example person or thing. For example, a prototypical instance of a "Customer" class would contain the data that a typical customer would provide.

The objects in a model usually appear in activity and sequence diagrams.

Naming Conventions

Each object must have a unique name. A full object name includes an object name, role name, and class name. You may use any combination of these three parts of the object name. The following table identifies the variations of object names.

Syntax	Example	Description
<u>object/role:class</u>	<u>cart100/storage:cart</u>	Named instance (cart100) of the cart class that is playing the storage role during an interaction.
<u>object:class</u>	<u>cart100:cart</u>	Named instance (cart100) of the cart class.
<u>/role:class</u>	<u>/storage:cart</u>	Anonymous instance of the cart class playing the storage role in an interaction.
<u>object/role</u>	<u>cart/storage</u>	An object named cart playing the storage role. This object is either an object that is

		hiding the name of the class or an instance that is not associated with a class.
<u>object</u>	<u>cart100</u>	An object named cart100. This object is either an instance that is hiding the name of the class or an instance that is not associated with a class.
<u>/role</u>	<u>/storage</u>	An anonymous instance playing the storage role. This object is either an instance that is hiding the name of the object and class or an instance that is not associated with an object or class.
<u>:class</u>	<u>:cart</u>	Anonymous instance of the customer class.

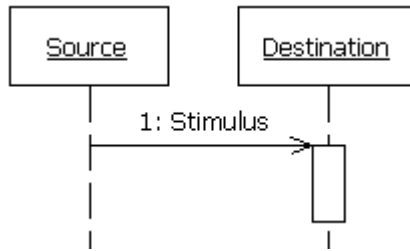
Stimuli

A stimulus is a model element that represents a communication between objects in a sequence diagram and usually indicates that an activity will follow. The types of communications that stimuli model include calls to operations, signals to objects, the creation of objects, and the destruction of objects. The receipt of a stimulus is an instance of an event.

For example, in an e-commerce application, you can model how a customer logs in to the application. A "Customer" actor instance sends a stimulus containing a name and password to a "LoginForm" object, and the "LoginForm" object sends a stimulus to itself to verify the input.

Shape

A stimulus appears as a line with an arrow. The direction of the arrow indicates the direction in which the stimulus is sent. In a sequence diagram, a stimulus usually connects two object lifelines.



Stimulus shapes can be adorned with names and sequence numbers.

Types of Stimuli

Different types of stimuli can be used to model different flows of control.

Type	Shape	Description
Procedure Call or Nested Flow of Control	→	Models either a call to an operation or a call to a nested flow of control. When calling a nested flow of control, the system waits for the nested flow of control to complete before continuing with the outer flow.
Asynchronous Flow of Control	→	Models an asynchronous stimulus between two objects. The source object sends the stimulus and immediately continues with the next step.
Return from a Procedure Call	- - →	Models a return from a call to a procedure. This type of stimulus can be omitted from diagrams because it is assumed that every call has a return.

Using Stimuli

You can add stimuli to your model to represent the communications exchanged between objects during dynamic interaction instances.

Note Both messages and stimuli are supported. Stimuli are added to collaboration instances, and messages are added to collaborations. For details about messages, see Messages.

The stimuli in a model are contained in collaboration instances and appear in

sequence diagrams.

Naming Conventions

Stimuli can have either names or signatures.

Type	Example	Description
Name	// Get the Password	A name identifies only the name of the stimulus. Simple names are often used in diagrams developed during analysis because the stimuli are identified by their responsibilities and not by their operations. One convention uses double slashes (//) to indicate that the stimulus name is not associated with an operation.
Signature	getPassword(String)	When an operation is assigned to a stimulus, you can display the operation signature to identify the name of the operation and its parameters. Signatures are often used in diagrams developed during design because they provide the detail that developers need when they code the design.

Actions

An action is represented as shown below:

```
Action/Ins#1
```

The action box displays the name of the action.

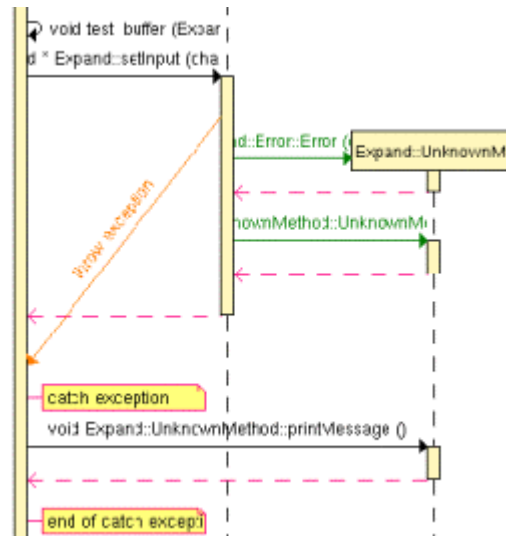
The action is linked to its source file. In the UML/SD Viewer, click an action to open the Text Editor at the corresponding line in the source code.

Exceptions

When tracing C++ exceptions, Runtime Tracing locates the throw point of the

exception (the throw keyword in C++) as well as its catch point.

Exceptions are displayed as a slanted red line, as shown in the example below, generated by Runtime Tracing.



To jump to the corresponding portion of source code:

Click an instance to open the Text Editor at the line in the source code where the exception is thrown.

Click the **catch exception** or **end of catch exception** notes to open the Text Editor at the line where the exception is caught.

To filter an instance out of the UML sequence diagram:

Right-click an exception and select **Filter instance** in the pop-up menu.

Actors

An actor is a model element that describes a role that a user plays when interacting with the system being modeled. Actors, by definition, are external

to the system. Although an actor typically represents a human user, it can also represent an organization, system, or machine that interacts with the system. An actor can correspond to multiple real users, and a single user may play the role of multiple actors.

Shape

An actor usually appears as a "stick man" shape.



In models depicting software applications, actors represent the users of the system. Examples include end users, external computer systems, and system administrators.

Naming Conventions

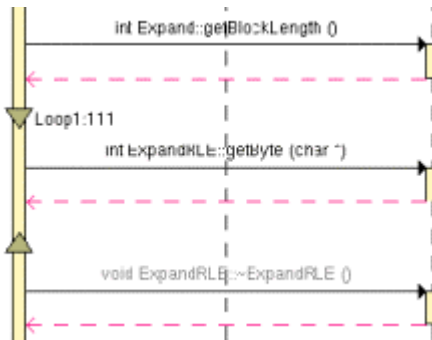
Each actor has a unique name that describes the role the user plays when interacting with the system.

Loops

Loop detection simplifies UML sequence diagrams by summarizing repeating traces into a loop symbol.

Note Loops are a Rational extension to UML Sequence Diagrams and are not supported by the UML standard.

A loop is represented as shown below:



A tag displays the name of the loop and the number of executions.

The loop is linked to its source file. In the UML/SD Viewer, click a loop to open the Text Editor at the corresponding line in the source code.

To configure Runtime Tracing to detect loops:

1. From the Project Explorer, select the highest level node to which you want to apply the option, such as the Workspace.
2. Right-click the node, and select **Settings...** from the pop-up menu.
3. In the **Configuration Settings** dialog, select the **Runtime Tracing** node, and **Trace Control**.
4. From the options box, set the **Automatic Loop Detection** to **Yes**.
5. Click **OK**.

Synchronizations

Synchronizations are an extension to the UML standard that only apply when using the split trace file feature of Runtime Tracing. They are used to show that all instance lifelines are synchronized at the beginning and end of each split TDF file.

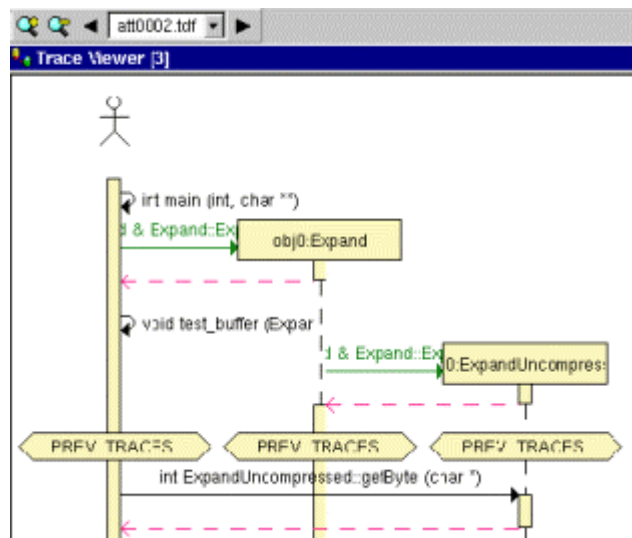
Shape

A synchronization is represented as shown below:



The synchronization box displays the name of the synchronization.

The synchronization is linked to its source file. In the UML/SD Viewer, click a synchronization to open the Text Editor at the corresponding line in the source code.



When the Split Trace capability is enabled, the UML/SD Viewer displays the list of TDF files generated in the UML/SD Viewer toolbar.

At the beginning of each diagram, before the Synchronization, the Viewer displays the context of the previous file.

Another synchronization is displayed at the end of each file, to insure that all instance lifelines are together before viewing the next file.

Notes

Notes appear as shown below and are centered on, and attached to, the element to which they apply:



UML notes can be associated to messages and instances.

The note is linked to its source file. In the UML/SD Viewer, click a note to open the Text Editor at the corresponding line in the source code.

Upgrading from a Previous Version

The current version of Rational Test RealTime is capable of importing from Rational Test RealTime v2001A or from any of the following discontinued ATTOL Testware products.

- ATTOL UniTest
- ATTOL SystemTest
- ATTOL Coverage

Files from later versions of Test RealTime are directly compatible with the current version.

Upgrading from ATTOL UniTest

Test RealTime v2002.05 and later provides an **Import** function to import **.prj**, **.cmp**, and **.ses** files from ATTOL UniTest and Rational Test RealTime v2001A. See Importing v2001 Component Testing Files for more information.

Upgrading from ATTOL SystemTest

There is no import facility for ATTOL SystemTest files. The recommended approach is to manually import the existing source and **.pts** test script files into a Test RealTime workspace with the System Testing Wizard.

Project files created in ATTOL SystemTest Studio are not compatible with Rational Test RealTime.

To import an ATTOL SystemTest project into Test RealTime:

1. Use the **Activity Wizard** to create a new workspace and System Testing test node.
When prompted, specify your existing source files.
2. Follow the indications until the **System Testing Wizard** appears.
3. In the **System Testing Wizard**, clear the **Create a new test script** option and click the ... button to import an existing **.pts** test script. Click **Add** to add the **.h** interface file.
4. Follow the standard System Testing instructions to configure and deploy the Virtual Testers.

Upgrading from ATTOL Coverage

The Code Coverage Viewer in Test RealTime can open and display **.fdc** and **.tio** files from ATTOL Coverage versions.

To open a Code Coverage Report:

1. From the **File** menu, select **Open**.
2. In the **File type** list, select **Code Coverage Viewer files (*.fdc,*.tio)**.
3. Locate and select the **.fdc** and **.tio** files from the older version.
4. Click **OK**.

Runtime Analysis

2

The runtime analysis feature set allows you to closely monitor the behavior of your application for debugging and validation purposes. Each feature *instruments* the source code providing real-time analysis of the application while it is running, either on a native or embedded target platform.

Using Runtime Analysis Features

The runtime analysis features of Test RealTime allow you to closely monitor the behavior of your application for debugging and validation purposes.

These features use Rational's unique SCI technology to *instrument* the source code providing real-time analysis of the application while it is running, either on a native or embedded target platform.

- Memory Profiling analyzes memory usage and detects memory leaks
- Performance Profiling provides performance load monitoring
- Code Coverage performs code coverage analysis
- Runtime Tracing draws a real-time UML Sequence Diagram of your application

In Test RealTime, each of these runtime analysis features can be used together with any of the automated testing features providing, for example, test coverage information.

Note SCI instrumentation of the source code generates a certain amount

of overhead, which can impact application size and performance. See Source Code Insertion Technology for more information.

How to use the runtime analysis features:

Here is a basic rundown of the main steps to using the runtime analysis feature set:

1. From the Start page, set up a new project. This can be done automatically with the New Project Wizard.
2. Follow the Activity Wizard to add your application source files to the workspace.
3. Select the source files under analysis in the wizard to create the application node. The wizard guides you through the process of selecting the right test feature for your needs.
4. Select the runtime analysis features to be applied to the application in the Build options.
5. Use the Project Explorer to set up the test campaign and add any additional runtime analysis or test nodes.
6. Run the application node to build and execute the instrumented application.
7. View and analyze the generated test reports.

In Test RealTime, runtime analysis options can be run within a test by simply adding the runtime analysis setting to an existing test node.

Code Coverage

Source-code coverage consists of identifying which portions of a program are executed or not during a given test case. Source-code coverage is recognized as one of the most effective ways of assessing the efficiency of

the test cases applied to a software application.

The Code Coverage feature brings efficient, easy-to-use robust coverage technologies to real-time embedded systems. Code Coverage provides a completely automated and proven solution for C, C++ and Ada software coverage based on optimized source-code instrumentation.

Coverage Types

The Code Coverage feature provides the capability of reporting of various source code units and branches, depending on the coverage type selected.

By default, Code Coverage implements full coverage analysis, meaning that all coverage types are instrumented by source code insertion (SCI). However, in some cases, you might want to reduce the scope of the Code Coverage report, such as to reduce the overhead generated by SCI for example.

Branches

When referring to the Code Coverage feature, a *branch* denotes a generic unit of enumeration. For each branch, you specify the coverage type. Code Coverage instruments each branch when you compile the source under test.

Coverage Levels

The following table provides details of each coverage type as used in each language supported by the product

Coverage Level	Languages			
Block Coverage	C	Ada	C++	Java
Call Coverage	C	Ada		
Condition Coverage	C	Ada		

Function, Unit or Method Coverage	C	Ada	C++	Java
Link Files		Ada		
Templates			C++	
Additional statements	C	Ada	C++	Java

To select a coverage level:

1. Right-click the application or test node concerned by the Code Coverage report.
2. From the pop-up menu, select **Settings**.
3. In the Configuration list, expand **Code Coverage** and select **Instrumentation Control**.
4. Select or clear the coverage levels as required.
5. Click **OK**.

Ada Coverage

Ada Block Coverage

When analyzing Ada source code, Code Coverage can provide the following block coverage types:

- Statement Blocks
- Statement Blocks and Decisions
- Statement Blocks, Decisions, and Loops

Statement Blocks (or Simple Blocks)

Simple blocks are the main blocks within units as well as blocks introduced by decisions, such as:

- **then** and **else (elsif)** of an **if**

- **loop...end loop** blocks of a **for...while**
- **exit when...end loop** or **exit when** blocks at the end of an instruction sequence
- **when** blocks of a **case**
- **when** blocks of exception processing blocks
- **do...end** block of the **accept** instruction
- **or** and **else** blocks of the **select** instruction
- **begin...exception** blocks of the **declare** block that contain an exceptions processing block.
- **select...then abort** blocks of an **ATC** statement
- sequence blocks: instructions found after a potentially terminal statement.

A simple block constitutes one branch. Each unit contains at least one simple block corresponding to its body, except packages that do not contain an initialization block.

Decision Coverage (Implicit Blocks)

An if statement without an else statement introduces an implicit block.

```
-- Function power_10
-- -block=decision or -block=implicit
function power_10 ( value, max : in integer) return integer
is
    ret, i : integer ;
begin
    if ( value == 0 ) then
        return 0;
    -- implicit else block
    end if ;
    for i in 0..9
    loop
        if ( (max /10) < ret ) then
            ret := ret *10 ;
        else
            ret := max ;
        end if ;
    end loop ;
end power_10 ;
```

```

        end if ;
    end loop ;
    return ret;
end ;

```

An implicit block constitutes one branch.

Implicit blocks refer to simple blocks to describe possible decisions. The Code Coverage report presents the sum of these decisions as an absolute value and a ratio.

Loop Coverage (Logical Blocks)

A **for** or **while** loop constitutes three branches:

- The simple block contained in the loop is never executed: the exit condition is *true* immediately
- The simple block is run only once: the exit condition is *false*, and then *true* on the next iteration
- The simple block run at least twice: the exit condition is *false* at least twice, then finally *true*)

A **loop...end loop** block requires only two branches because the exit condition, if it exists, is tested within the loop:

- The simple block is played only once: the exit condition is *true* on the first iteration, if the condition exists
- The simple block is played at least twice: the exit condition *false* at least once and then finally *true*, if the condition exists

In the following example, you need to execute the function **try_five_times()** several times for 100 % coverage of the three logical blocks induced by this while loop.

```

-- Function try_five_times
function try_five_times return integer is
    result, i : integer := 0 ;
begin

```

```

-- try is any function
while ( i < 5 ) and then ( result <= 0 ) loop
    result := try ;
    i := integer'succ(i);
end loop ;
return result;
end ; -- 3 logical blocks

```

Logical blocks are attached to the **loop** introduction keyword.

Asynchronous Transfer of Control (ATC) Blocks

This coverage type is specific to the ADA 95 Asynchronous Transfer of Control (ATC) block statement (see your ADA documentation).

The ATC block contains tree branches:

- **Control immediately transferred:** The sequence of control never passes through the block then abort /end select, but is immediately transferred to the block select/then abort.
- **Control transferred:** The sequence of control starts at the block then abort/end select, but never reaches the end of this block. Because of trigger event appearance, the sequence is transferred to the block select/then abort.
- **Control never transferred:** Because the trigger event never appears, the sequence of control starts and reaches the end of the block then abort/end select, and was never transferred to the block select/then abort.

In the following example, you need to execute the **compute_done** function several times to obtain full coverage of the three ATC blocks induced by the select statement:

```

function compute_done return boolean is
    result : boolean := true ;
begin
    -- if computing is not done before 10s ...
    select
        delay 10.0;

```

```
    result := false ;
  then abort
    compute;
  end select;
  return result;
end ; -- 3 logical blocks
```

Code Coverage blocks are attached to the **Select** keyword of the ATC statement.

Ada Call Coverage

When analyzing Ada source code, Code Coverage can provide coverage of function, procedure, or entry calls.

Code Coverage defines as many branches as it encounters function, procedure, or entry calls.

This type of coverage ensures that all the call interfaces can be shown to have been exercised for each Ada unit (procedure, function, or entry). This is sometimes a pass/fail criterion in the software integration test phase.

Ada Condition Coverage

Basic Conditions

Basic conditions are operands of logical operators (standard or derived, but not overloaded) or, xor, and, not, or else, or and then, wherever they appear in ADA units. They are also the conditions of if, while, exit when, when of entry body, and when of select statement, even if these conditions do not contain logical operators. For each of these basic conditions, two branches are defined: the sub-condition is true and the sub-condition is false.

A basic condition is also defined for each when of a case statement, even each sub-expression of a compound when, that is when A | B: two branches.

```
-- power of 10 function
-- -cond
```

```

Function power_of_10( value, max : in integer )
is
  result : integer ;
Begin
  if value = 0 then
    return 0;
  end if ;
  result := value ;
  for i in 0..9 loop
    if ( max > 0 ) and then (( max / value ) < result )
then
      result := result * value;
    else
      result := max ;
    end if ;
  end loop;
  return result ;
end ; -- there are 3 basic conditions (and 6 branches).
-- Near_Color function
Function Near_Color ( color : in ColorType ) return
ColorType
is
Begin
  case color is
    when WHITE | LIGHT_GRAY => return WHITE ;
    when RED | LIGHT_RED .. PURPLE => return RED ;
  end case ;
End ; -- there are 4 basics conditions (and 4 branches).

```

Two branches are enumerated for each Boolean basic condition, and one per case basic condition.

Forced Conditions

A forced condition is a multiple condition in which any occurrence of the or else operator is replaced with the or operator, and the and then operator is replaced with the and operator. This modification forces the evaluation of the second member of these operators. You can use this coverage type after modified conditions have been reached to ensure that all the contained basic conditions have been evaluated. With this coverage type, you can be sure that only the considered basic condition value changes between both condition vectors.

```

-- Original source :
-- -cond=forceevaluation

```

```

    if ( a and then b ) or else c then
-- Modified source :
    if ( a and      b ) or      c then

```

Note This replacement modifies the code semantics. You need to verify that using this coverage type does not modify the behavior of the software.

Example

```

procedure P ( A : in tAccess ) is
begin
    if A /= NULL and then A.value > 0      -- the evaluation of
A.value will raise an                    -- exception when
using forced conditions                  -- if the A pointer
is nul
    then
        A.value := A.value - 1;
    end if;
end P;

```

Modified Conditions

A modified condition is defined for each basic condition enclosed in a composition of logical operators (standard or derived, but not overloaded). It aims to prove that this condition affects the result of the enclosing composition. To do that, find a subset of values affected by the other conditions, for example, if the value of this condition changes, the result of the entire expression changes.

Because compound conditions list all possible cases, you must find the two cases that can result in changes to the entire expression. The modified condition is covered only if the two compound conditions are covered.

```

-- State_Control state
-- -cond=modified
Function State_Condtol return integer
is
Begin
    if ( ( flag_running and then ( process_count > 10 ) )
        or else flag_stopped )

```

```

        then
            return VALID_STATE ;
        else
            return INVALID_STATE ;
        end if ;
End ;
-- There are 3 basic conditions, 5 compound conditions
-- and 3 modified conditions :
--   flag_running : TTX=T and FXF=F
--   process_count > 10 : TTX=T and TFF=F
--   flag_stopped : TFT=T and TFF=F, or FXT=T and FXF=F
-- 4 test cases are enough to cover all the modified
conditions :
--   TTX=T
--   FXF=F
--   TFF=F
--   FTF=F or FXT=T

```

Note You can associate a modified condition with more than one case, as shown in this example for **flag_stopped**. In this example, the modified condition is covered if the two compound conditions of at least one of these cases are covered.

Code Coverage calculates cases for each modified condition.

The same number of modified conditions as Boolean basic conditions appear in a composition of logical operators (standard or derived, but not overloaded).

Multiple Conditions

A multiple condition is one of all the available cases of logical operators (standard or derived, but not overloaded) wherever it appears in an ADA unit. Multiple conditions are defined by the concurrent values of the enclosed basic boolean conditions.

A multiple condition is noted with a set of T, F, or X letters, which means that the corresponding basic condition evaluates to true or false, or it was not evaluated, respectively. Such a set of letters is called a condition vector. The right operand of or else or and then logical operators is not evaluated if

the evaluation of the left operand determines the result of the entire expression.

```
-- State_Control Function
-- -cond=compound
Function State_Control return integer
is
Begin
    if ( ( flag_running and then ( process_count > 10 ) )
        or else flag_stopped
    then
        return VALID_STATE ;
    else
        return INVALIDE_STATE ;
    end if ;
End ;
-- There are 3 basic conditions
-- and 5 compound conditions :
--     TTX=T <=> ((T and then T) or else X ) = T
--     TFT=T
--     TFF=F
--     FXT=T
--     FXF=F
```

Code Coverage calculates the computation of every available case for each composition.

The number of enumerated branches is the number of distinct available cases for each composition of logical operators (standard or derived, but not overloaded).

Ada Unit Coverage

Unit Entries

Unit entries determine which units are executed and/or evaluated.

```
-- Function factorial
-- -proc
function factorial ( a : in integer ) return integer is
begin
    if ( a > 0 ) then
        return a * factorial ( a - 1 );
    else
        return 1;
```



```

        end if;
    end factorial ;

```

One branch is defined for each defined and instrumented unit. In the case of a package, the unit entry only exists if the package body contains the begin/end instruction block.

For Protected units, no unit entry is defined because this kind of unit does not have any statements blocks.

Unit Exits and Returns

These are the standard exit (if it is coverable), each return instruction (from a procedure or function), and each exception-processing block in the unit.

```

-- Function factorial
-- -proc=ret
function factorial ( a : in integer ) return integer is
begin
    if ( a > 0 ) then
        return a * factorial ( a - 1 );
    else
        return 1;
    end if ;
end factorial ; -- the standard exit is not coverable
-- Procedure divide
procedure divide ( a,b : in integer; c : out integer ) is
begin
    if ( b == 0 ) then
        text_io.put_line("Division by zero" );
        raise CONSTRAINT_ERROR;
    end if ;
    if ( b == 1 ) then
        c := a;
        return;
    end if ;
    c := a / b;
exception
    when PROGRAM_ERROR => null ;
end divide ;

```

For Protected units, no exit is defined because this kind of unit does not have any statements blocks.

In general, at least two branches per unit are defined; however, in some

cases the coding may be such that:

- There are no unit entries or exits (a package without an instruction block (begin/end), protected units case).
- There is only a unit entry (an infinite loop in which the exit from the task cannot be covered and therefore the exit from the unit is not defined).

The entry is always numbered if it exists. The exit is also numbered if it is coverable. If it is not coverable, it is preceded by a terminal instruction containing return or raise instructions; otherwise, it is preceded by an infinite loop.

A raise is considered to be terminal for a unit if no processing block for this exception was found in the unit.

Ada Link Files

Link files are the library management system used for Ada Coverage. These libraries contain the entire Ada compilation units contained by compiler sources, the predefined Ada environment and the source files of your projects. You must use link files when using Code Coverage in Ada for the Ada Coverage analyzer to correctly analyze your source code.

You can include a link file within another link file, which is an easy way to manage your source code.

Link File Syntax

Link files have a line-by-line syntax. Comments start with a double hyphen (--), and end at the end of the line. Lines can be empty.

There are two types of configuration lines:

- **Link file inclusion:** The link filename can be relative to the link file

that contains this line or absolute.

```
<link filename> LINK
```

- **Compilation unit description:** The source filename is the file containing the described compilation unit (absolute or relative to the link filename). The full unit name is the Ada full unit name (beware of separated units, or child units).

```
<source filename> <full unit name> <type> [ada83]
```

The *<type>* is one of the following flags:

- **SPEC** for specification
- **BODY** for a body
- **PROC** for procedure or function

Use the optional **ada83** flag if the source file cannot be compiled in Ada 95 mode, and must be analyzed in Ada 83 mode.

Generating a Link File

The link file can be generated either manually or automatically with the Ada Link File Generator (**attolalk**) tool. See the **Test RealTime Reference Manual** for more information about command line tools.

Sending the Link File to the Instrumentor

The loading order of link files is important. If the same unit name is found twice or more in one (or more) loaded link files, the Instrumentor issues a warning and uses the last encountered unit.

Included link files are analyzed when the file including the link file is loaded.

In Ada, Code Coverage loads the link files in the following order:

- By default, either **adalib83.alk** or **adalib95.alk** is loaded. These files are

part of the Target Deployment Port.

- If you use the **-STDLINK** command line option, the specified standard link file is loaded first. See the **Test RealTime Reference Manual** for more information
- The link file specified by the **ATTOLCOV_ADALINK** environment variable is loaded.
- The link files specified by the **-Link** option is loaded.

Now, you can start analyzing the file instrument.

Loading A Permanent Link File

You can ask Code Coverage to load the link file at each execution. To do that, set the environment variable **ATTOLCOV_ADALINK** with the link filename separated by ':' on a UNIX system, or ';' in Windows. For example:

```
ATTOLCOV_ADALINK="compiler.alk/projects/myproject/myproject.alk"
```

A Link file specified on the command line is loaded after the link file specified by this environment variable.

Ada Additional Statements

Terminal Statements

An ADA statement is terminal if it transfers control of the program anywhere other than to a sequence (*return, goto, raise, exit*).

By extension, a decision statement (*if, case*) is also terminal if all its branches are terminal (i.e., *if, then* and *else* blocks and non-empty *when* blocks contain a terminal instruction). An *if* statement without an *else* statement is never terminal, since one of the blocks is empty and therefore transfers control in sequence.

Potentially Terminal Statements

An Ada statement is potentially terminal if it contains a decision choice that transfers control of the program anywhere other than after it (*return*, *goto*, *raise*, *exit*).

Non-coverable Statements

An Ada statement is detected as being not coverable if it is not a *goto* label and if it is in a terminal statement sequence. Statements that are not coverable are detected by the feature during the instrumentation. A warning is generated to signal each one, which specifies its location source file and line. This is the only action Code Coverage takes for statements that cannot be covered.

Note Ada units whose purpose is to terminate execution unconditionally are not evaluated. This means that Code Coverage does not check that procedures or functions terminate or return.

Similarly, exit conditions for loops are not analyzed statistically to determine whether the loop is infinite. As a result, a *for*, *while* or *loop*/*exit* when loop is always considered non-terminal (i.e., able to transfer control in its sequence). This is not applicable to *loop*/*end loop* loops without an *exit* statement (with or without condition), which are terminal.

C Coverage

C Block Coverage

When running the Code Coverage feature on Ada source code, Test RealTime can provide the following coverage types for code blocks:

- Statement Blocks
- Statement Blocks and Decisions

- Statement Blocks, Decisions, and Loops

Statement Blocks (or Simple Blocks)

Simple blocks are the C function main blocks, blocks introduced by decision instructions:

- **THEN** and **ELSE FOR IF**
- **FOR, WHILE** and **DO ... WHILE** blocks
- non-empty blocks introduced by switch case or default statements
- true and false outcomes of ternary expressions (*<expr>? <expr> : <expr>*)
- blocks following a potentially terminal statement.

```

/* Power_of_10 Function */
/* -block */
int power_of_10 ( int value, int max )
{
    int retval = value, i;
    if ( value == 0 ) return 0; /* potentially terminal
statement */
    for ( i = 0; i < 10; i++ ) /* start of a sequence block
*/
    {
        retval = ( max / 10 ) < retval ? retval * 10 : max;
    }
    return retval;
} /* The power_of_10 function has 6 blocks */
/* Near_color function */
ColorType near_color ( ColorType color )
{
    switch ( color )
    {
        case WHITE :
        case LIGHT_GRAY :
            return WHITE;
        case RED :
        case PINK :
        case BURGUNDY :
            return RED;
        /* etc ... */
    }
} /* The near_color function has at least 3 simple blocks */

```

Each simple block is a branch. Every C function contains at least one simple block corresponding to its main body.

Decisions (Implicit Blocks)

Implicit blocks are introduced by an **IF** statement without an **ELSE** or a **SWITCH** statement without a **DEFAULT**.

```
/* Power_of_10 function */
/* -block=decision */
int power_of_10 ( int value, int max )
{
  int retval = value, i;
  if ( value == 0 ) return 0; else ;
  for ( i =0;i <10;i++)
  {
    retval = ( max / 10 ) < retval ? retval * 10 : max;
  }
  return retval;
}
/* Near_color function */
ColorType near_color ( ColorType color )
{
  switch ( color )
  {
    case WHITE :
    case LIGHT_GRAY :
      return WHITE;
    case RED :
    case PINK :
    case BURGUNDY :
      return RED;
    /* etc ... with no default */
    default : ;
  }
}
```

Each implicit block represents a branch.

Because the sum of all possible decision paths includes implicit blocks as well as statement blocks, reports provide the total number of simple and implicit blocks as a figure and as a percentage. Code Coverage places this information in the **Decisions** report.

Loops (Logical Blocks)

A typical **FOR** or **WHILE** loop can reach three different conditions:

- The statement block contained within the loop is executed zero times, therefore the output condition is *True* from the start
- The statement block is executed exactly once, the output condition is *False*, then *True* the next time
- The statement block is executed at least twice. (The output condition is *False* at least twice, and becomes *True* at the end)

In a **DO...WHILE** loop, because the output condition is tested after the block has been executed, two further branches are created:

- The statement block is executed exactly once. The output is condition *True* the first time.
- The statement block is executed at least twice. (The output condition is *False* at least once, then true at the end)

In this example, the function **try_five_times ()** must run several times to completely cover the three logical blocks included in the **WHILE** loop:

```
/* Try_five_times function */
/* -block=logical */
int try_five_times ( void )
{
  int result,i =0;
  /*try () is a function whose return value depends
  on the availability of a system resource, for example */
  while ( ( ( result = try ( ) ) != 0 ) &&
  ( ++i < 5 ) );
  return result;
} /* 3 logical blocks */
```

C Call Coverage

When analyzing Ada source code, Code Coverage can provide coverage of function or procedure calls.

Code Coverage defines as many branches as it encounters function calls.

Procedure calls are made during program execution.

This type of coverage ensures that all the call interfaces can be shown to have been exercised for each C function. This may be a pass or failure criterion in software integration test phases.

You can use the **-EXCALL** option to select C functions whose calls you do not want to instrument, such as C library functions for example.

Example

```
/* Evaluate function */
/* -call */
int evaluate ( NodeTypeP node )
{
    if ( node == (NodeTypeP)0 ) return 0;
    switch ( node->Type )
    {
        int tmp;
        case NUMBER :
            return node->Value;
        case IDENTIFIER :
            return current value ( node->Name );
        case ASSIGN :
            set ( node->Child->Name,
                tmp = evaluate ( node->Child->Sibling )
            );
        case ADD :
            return evaluate ( node->Child ) +
                evaluate ( node->Child->Sibling );
        case SUBTRACT :
            return evaluate ( node->Child ) -
                evaluate ( node->Child->Sibling );
        case MULTIPLY :
            return evaluate ( node->Child ) *
                evaluate ( node->Child->Sibling );
        case DIVIDE :
            tmp = evaluate ( node->Child->Sibling );
            if ( tmp == 0 ) fatal error ( "Division by zero" );
            else return evaluate ( node->Child ) / tmp;
    }
} /* There are twelve calls in the evaluate function */
```

C Condition Coverage

When analyzing C source code, Test RealTime can provide the following condition coverage:

- Basic Coverage
- Forced Coverage

Basic Conditions

Conditions are operands of either `||` or `&&` operators wherever they appear in the body of a C function. They are also if and ternary expressions, tests for **for**, **while**, and **do/while** statements even if these expressions do not contain `||` or `&&` operators. Two branches are involved in each condition: the sub-condition being true and the sub-condition being false.

Basic conditions also enable different case or default (which could be implicit) in a switch to be distinguished even when they invoke the same simple block. A basic condition is associated with every case and default (written or not).

```
/* Power_of_10 function */
/* -cond */
int power_of_10 ( int value, int max )
{
    int result = value, i;
    if ( value == 0 ) return 0;
    for ( i = 0; i < 10; i++ )
    {
        result = max > 0 && ( max / value ) < result ?
                result * value :
                max;
    }
    return result ;
} /* There are 4*2 basic conditions in this function */
/* Near_color function */
ColorType near_color ( ColorType color )
{
    switch ( color )
    {
        case WHITE :
        case LIGHT_GRAY :
```

```

        return WHITE;
    case RED :
    case PINK :
    case BURGUNDY :
        return RED;
    /* etc ... */
}
} /* There are at least 5 basic conditions here */

```

Two branches are enumerated for each condition, and one per case or default.

Forced Conditions

Forced conditions are multiple conditions in which any occurrence of the | and && operators has been replaced in the code with | and & binary operators. Such a replacement done by the Instrumentor enforces the evaluation of the right operands. You can use this coverage type after modified conditions have been reached to be sure that every basic condition has been evaluated. With this coverage type, you can be sure that only the considered basic condition changed between the two tests.

```

/* User source code */
cond=forceevaluation */
if ( ( a && b ) || c ) ...
/* Replaced with the Code Coverage feature with : */
if ( ( a & b ) | c ) ...
/* Note : Operands evaluation results are enforced to one if
different from 0 */
/* -

```

Note This replacement modifies the code semantics. You need to verify that using this coverage type does not modify the behavior of the software.

```

int f ( MyStruct *A )
{
    if ( A && A->value > 0 ) /* the evaluation of
A->value will cause a program error using forced conditions
if A pointer is null */
    {
        A->value -= 1;
    }
}

```

```
}
```

Modified Conditions

A modified condition is defined for each basic condition enclosed in a composition of `|` or `&&` operators. It aims to prove that this condition affects the result of the enclosing composition. To do that, find a subset of values affected by the other conditions, for example, if the value of this condition changes, the result of the entire expression changes.

Because compound conditions list all possible cases, you must find the two cases that can result in changes to the entire expression. The modified condition is covered only if the two compound conditions are covered.

```
/* state_control function */
int state_control ( void )
{
    if ( ( ( flag & 0x01 ) &&
          ( instances_number > 10 ) ) ||
        ( flag & 0x04 ) )
        return VALID_STATE;
    else
        return INVALID_STATE;
} /* There are 3 basic conditions, 5 compound conditions
   and 3 modified conditions :
   flag & 0x01 : TTX=T and FXF=F
   nb_instances > 10 : TTX=T and TFF=F
   flag & 0x04 : TFT=T and TFF=F, or FXT=T and FXF=F
   4 test cases are enough to cover all those modified
   conditions :
   TTX=T
   FXF=F
   TFF=F
   TFT=T or FXT=T
*/
```

Note You can associate a modified condition with more than one case, as shown in this example for `flag & 0x04`. In this example, the modified condition is covered if the two compound conditions of at least one of these cases are covered.

Code Coverage calculates matching cases for each modified condition.

The same number of modified conditions as Boolean basic conditions appears in a composition of `||` and `&&` operators.

Multiple Conditions

A multiple (or compound) condition is one of all the available cases for the `||` and `&&` logical operator's composition, whenever it appears in a C function. It is defined by the simultaneous values of the enclosed Boolean basic conditions.

A multiple condition is noted with a set of T, F, or X letters. These mean that the corresponding basic condition evaluated to true, false, or was not evaluated, respectively. Remember that the right operand of a `||` or `&&` logical operator is not evaluated if the evaluation of the left operand determines the result of the entire expression.

```
/* state_control function */
/* -cond=compound */
int state_control ( void )
{
    if ( ( ( flag & 0x01 ) &&
           ( instances_number > 10 ) ) ||
         ( flag & 0x04 ) )
        return VALID_STATE;
    else
        return INVALID_STATE;
} /* There are 3 basic conditions
   and 5 compound conditions :
   TTX=T <=> (( T && T ) || X ) = T
   TFT=T
   TFF=F
   FXT=T
   FXF=F
*/
```

Code Coverage calculates every available case for each composition.

The number of enumerated branches is the number of distinct available cases for each composition of `||` or `&&` operators.

C Function Coverage

When analyzing C source code, Test RealTime can provide the following function coverage:

- Procedure Entries
- Procedure Entries and Exits

Procedure Entries

Inputs identify the C functions that are executed.

```
/* Factorial function */
/* -proc */
int factorial ( int a )
{
    if ( a > 0 ) return a * factorial ( a - 1 );
    else return 1;
}
```

One branch is defined per C function.

Procedure Entries and Exits (Returns and Terminal Statements)

These include the standard output (if coverable), and all return instructions, exits, and other terminal instructions that are instrumented, as well as the input.

```
/* Factorial function */
/* -proc=ret */
int factorial ( int a )
{
    if ( a > 0 ) return a * factorial ( a - 1 );
    else return 1;
} /* standard output cannot be covered */
/* Divide function */
void divide ( int a, int b, int *c )
{
    if ( b == 0 )
    {
        fprintf ( stderr, "Division by zero\n" );
        exit ( 1 );
    };
};
```

```
    if ( b == 1 )
    {
        *c = a;
        return;
    };
    *c = a / b;
}
```

At least two branches are defined per C function.

The input is always enumerated, as is the output if it can be covered. If it cannot, it is preceded by a terminal instruction involving returns or an exit.

In addition to the terminal instructions provided in the standard definition file, you can define other terminal instructions using the pragma **attol_exit_instr**.

C Additional Statements

Terminal Statements

A C statement is *terminal* if it transfers program control out of sequence (**RETURN**, **GOTO**, **BREAK**, **CONTINUE**), or stops the execution (**EXIT**).

By extension, a decision statement (**IF** or **SWITCH**) is terminal if all branches are terminal; that is if the non-empty **THEN ... ELSE**, **CASE**, and **DEFAULT** blocks all contain terminal statements. An **IF** statement without an **ELSE** and a **SWITCH** statement without a **DEFAULT** are never terminal, because their empty blocks necessarily continue program control in sequence.

Potentially Terminal Statements

The following decision statements are potentially terminal if they contain at least one statement that transfers program control out of their sequence (**RETURN**, **GOTO**, **BREAK**, **CONTINUE**), or that terminates the execution (**EXIT**):

- **IF** without an **ELSE**
- **SWITCH**
- **FOR**
- **WHILE** or **DO ... WHILE**

Non-coverable Statements in C

Some C statements are considered *non-coverable* if they follow a terminal instruction, a **CONTINUE**, or a **BREAK**, and are not a **GOTO** label. Code Coverage detects non-coverable statements during instrumentation and produces a warning message that specifies the source file and line location of each non-coverable statement.

Note User functions whose purpose is to terminate execution unconditionally are not evaluated. Furthermore, Code Coverage does not statically analyze exit conditions for loops to check whether they are infinite. As a result, **FOR ... WHILE** and **DO ... WHILE** loops are always assumed to be *non-terminal*, able to resume program control in sequence.

C++ Coverage

C++ Block Code Coverage

When analyzing C++ source code, Code Coverage can provide the following block coverage types:

- Statement Blocks
- Statement Blocks and Decisions
- Statement Blocks, Decisions, and Loops

Statement Blocks

Statement blocks are the C++ function or method main blocks, blocks introduced by decision instructions:

- **THEN** and **ELSE FOR IF, WHILE** and **DO ... WHILE** blocks
- non-empty blocks introduced by **SWITCH CASE** or **DEFAULT** statements
- true and false outcomes of ternary expressions (*<expr>? <expr>: <expr>*)
- **TRY** blocks and any associated catch handler
- blocks following a potentially terminal statement.

```
int main ( )                                     /*
-BLOCK */
{
    try {
        if ( 0 )
        {
            func ( "Hello" );
        }
        else
        {
            throw UnLucky ( );
        }
    }
    catch ( Overflow & o ) {
        cout << o.String << '\n';
    }
    catch ( UnLucky & u ) {
        throw u;
    }
    return 0; /* potentially terminal statement */
} /* sequence block */
```

Each simple block is a branch. Every C++ function and method contains at least one simple block corresponding to its main body.

Decisions (Implicit Blocks)

Implicit blocks are introduced by **IF** statements without an **ELSE** statement, and a **SWITCH** statements without a **DEFAULT** statement.

```

/* Power_of_10 function */
/* -BLOCK=DECISION or -BLOCK=IMPLICIT */
int power_of_10 ( int value, int max )
{
    int retval = value, i;
    if ( value == 0 ) return 0; else ;
    for ( i = 0; i < 10; i++ )
    {
        retval = ( max / 10 ) < retval ? retval * 10 : max;
    }
    return retval;
}
/* Near_color function */
ColorType near_color ( ColorType color )
{
    switch ( color )
    {
        case WHITE :
        case LIGHT_GRAY :
            return WHITE;
        case RED :
        case PINK :
        case BURGUNDY :
            return RED;
        /* etc ... with no default */
        default : ;
    }
}

```

Each implicit block represents a branch.

Since the sum of all possible decision paths includes implicit blocks as well as simple blocks, reports provide the total number of simple and implicit blocks as a figure and a percentage after the term decisions.

Loops (Logical Blocks)

Three branches are created in a for or while loop:

- The first branch is the simple block contained within the loop, and that is executed zero times (the entry condition is false from the start).
- The second branch is the simple block executed exactly once (entry condition true, then false the next time).

- The third branch is the simple block executed at least twice (entry condition true at least twice, and false at the end).

Two branches are created in a **DO/WHILE** loop, as the output condition is tested after the block has been executed:

- The first branch is the simple block executed exactly once (output condition true the first time).
- The second branch is the simple block executed at least twice (output condition false at least once, then true at the end).

```
/* myClass::tryFiveTimes method */ /* -
BLOCK=LOGICAL */
int myClass::tryFiveTimes ()
{
    int result, i = 0;
    /* letsgo ( ) is a function whose return value depends
       on the availability of a system resource, for example
    */
    while ( ( ( result = letsgo ( ) ) != 0 ) &&
            ( ++i < 5 ) );
    return result;
} /* 3 logical blocks */
```

You need to execute the method **tryFiveTimes ()** several times to completely cover the three logical blocks included in the while loop.

C++ Method Code Coverage

Inputs to Procedures

Inputs identify the C++ methods executed.

```
/* Vector::getCoord() method */ /* -PROC
*/
int Vector::getCoord ( int index )
{
    if ( index >= 0 && index < size ) return Values[index];
    else return -1;
}
```

One branch per C++ method is defined.

Procedure Inputs, Outputs and Returns, and Terminal Instructions

These include the standard output (if coverable), all return instructions, and calls to `exit()`, `abort()`, or

`terminate()`, as well as the input.

```
/* Vector::getCoord() method */ /* -PROC=RET */
int Vector::getCoord ( int index )
{
    if ( index >= 0 && index < size ) return Values[index];
    else return -1;
}
/* Divide function */
void divide ( int a, int b, int *c )
{
    if (b ==0 )
    {
        fprintf ( stderr, "Division by zero\n" );
        exit (1 );
    };
    if (b ==1 )
    {
        *c =a;
        return;
    };
    *c =a /b;
}
```

At least two branches per C++ method are defined. The input is always enumerated, as is the output if it can be covered. If it cannot, it is preceded by a terminal instruction involving returns or by a call to `exit()`, `abort()`, or `terminate()`.

Potentially Terminal Statements

The following decision statements are potentially terminal if they contain at least one statement that transfers program control out of its sequence (**RETURN**, **THROW**, **GOTO**, **BREAK**, **CONTINUE**) or that terminates the execution (**EXIT**).

- **IF** without an **ELSE**

- **SWITCH, FOR**
- **WHILE** or **DO...WHILE**

C++ Template Instrumentation

Code Coverage performs the instrumentation of templates, functions, and methods of template classes, considering that all instances share their branches. The number of branches computed by the feature is independent of the number of instances for this template. All instances will cover the same once-defined branches in the template code.

Files containing template definitions implicitly included by the compiler (no specific compilation command is required for such source files) are also instrumented by the Code Coverage feature and present in the instrumented files where they are needed.

For some compilers, you must specifically take care of certain templates (for example, static or external linkage). You must verify if your Code Coverage Runtime installation contains a file named **templates.txt** and, if it does, read that file carefully.

- To instrument an application based upon Rogue Wave libraries , you must use the **-DRW_COMPILE_INSTANTIATE** compilation flag that suppresses the implicit include mechanism in the header files. (Corresponding source files are so included by pre-processing.)
- To instrument an application based upon ObjectSpace C++ Component Series , you must use the **-DOS_NO_AUTO_INSTANTIATE** compilation flag that suppresses the implicit include mechanism in the header files. (Corresponding source files are so included by pre-processing.)
- Any method (even unused ones) of an instantiated template class is analyzed and instrumented by the Instrumentor. Some compilers do not try to analyze such unused methods. It is possible that some of these methods are not fully compliant with C++ standards. For

example, a template class with a formal class template argument named T can contain a compare method that uses the == operator of the T class. If the C class used for T at instantiation time does not define an == operator, and if the compare method is never used, compilation succeeds but instrumentation fails. In such a situation, you can declare an == operator for the C class or use the **-instantiationmode=used** Instrumentor option.

C++ Additional Statements

Non-coverable Statements in C++

A C++ statement is *non-coverable* if the statement can never possibly be executed. Code Coverage detects non-coverable statements during instrumentation and produces a warning message that specifies the source file and line location of each non-coverable statement.

Java Coverage

Java Block Coverage

When analyzing Java source code, Code Coverage can provide the following block coverage:

- Statement Blocks
- Statement Blocks and Decisions
- Statement Blocks, Decisions, and Loops

Statement Blocks

Statement blocks are the Java method blocks, blocks introduced by control instructions:

- **THEN** for **IF** and **ELSE** for **IF**, **WHILE** and **DO ... WHILE** blocks

- non-empty blocks introduced by **SWITCH CASE** or **DEFAULT** statements
- true and false outcomes of ternary expressions (*<expr>? <expr> : <expr>*)
- **TRY** blocks and any associated catch handler
- blocks following a potentially terminal statement.

Example

```
public class StatementBlocks
{
    public static void func( String _message )
    throws UnsupportedOperationException
    {
        throw new UnsupportedOperationException(_message);
    }
    public static void main( String[] args )
    throws Exception
    {
        try {
            if ( false )
            {
                func( "Hello" );
            }
            else
            {
                throw new Exception("bad luck");
            }
        }
        catch ( UnsupportedOperationException _E )
        {
            System.out.println( _E.toString() );
        }
        catch ( Exception _E )
        {
            System.out.println( _E.toString() );
            throw _E ;
        } //potentially terminal statement
        return ; //sequence block
    }
}
```

Each simple block is a branch. Every Java method contains at least one simple block corresponding to its main body.

Decisions (Implicit Blocks)

Implicit blocks are introduced by **IF** statements without an **ELSE** statement, and a **SWITCH** statement without a **DEFAULT** statement.

Example

```
public class MathOp
{
    static final int WHITE=0;
    static final int LIGHTGRAY=1;
    static final int RED=2;
    static final int PINK=3;
    static final int BLUE=4;
    static final int GREEN=5;
    // power of 10
    public static int powerOf10( int _value, int _max )
    {
        int result = _value, i;
        if( _value==0 ) return 0; //implicit else
        for( i = 0; i < 10; i++ )
        {
            result = ( _max / 10 ) < result ? 10*result : _max ;
        }
        return result;
    }
    // Near color function
    int nearColor( int _color )
    {
        switch( _color )
        {
            case WHITE:
            case LIGHTGRAY:
                return WHITE ;
            case RED:
            case PINK:
                return RED;
            //implicit default:
        }
        return _color ;
    }
}
```

Each implicit block represents a branch.

Since the sum of all possible decision paths includes implicit blocks as well

as simple blocks, reports provide the total number of simple and implicit blocks as a figure and a percentage after the term decisions.

Loops (Logical Blocks)

Three branches are created in a **FOR** or **WHILE** loop:

- The first branch is the simple block contained within the loop, and that is executed zero times (the entry condition is false from the start).
- The second branch is the simple block executed exactly once (entry condition true, then false the next time).
- The third branch is the simple block executed at least twice (entry condition true at least twice, and false at the end).
- Two branches are created in a **DO/WHILE** loop, as the output condition is tested after the block has been executed:
 - The first branch is the simple block executed exactly once (output condition false the first time).
 - The second branch is the simple block executed at least twice (output condition false at least once, then true at the end).

Example

```
public class LogicalBlocks
{
    public static int tryFiveTimes()
    {
        int result, i=0;
        while ( ( ( result=resourcesAvailable() ) <= 0)
            && ( ++i < 5 ) );
        // while define 3 logical blocks
        return result;
    }
    public static int resourcesAvailable()
    {
        return ( _free_resources_++ );
    }
}

public static int _free_resources_=0;
```

```

public static void main( String[] argv )
{
    //first call: '0 loop' block is reach
    _free_resources_=1;
    tryFiveTimes();
    //second call: '1 loop' blocks are reach
    _free_resources_=0;
    tryFiveTimes();
    //third call: '2 loops or more' blocks are reach
    _free_resources_=-10;
    tryFiveTimes();
}
}

```

Java Method Coverage

Inputs to Procedures

Inputs identify the Java methods executed.

Example

```

public class Inputs
{
    public static int method()
    {
        return 5;
    }
    public static void main( String[] argv )
    {
        System.out.println("Value:"+method());
    }
}

```

One branch per Java method is defined.

Procedure Inputs, Outputs and Returns, and Terminal Instructions

These include the standard output (if coverable), all return instructions, and calls to exit(), abort(), or terminate(), as well as the input.

Example

```

public class InputsOutputsAndReturn

```

```

{
    public static void method0( int _selector )
    {
        if ( _selector < 0 )
        {
            return ;
        }
    }
    public static int method1( int _selector )
    {
        if( _selector < 0 ) return 0;
        switch( _selector )
        {
            case 1: return 0;
            case 2: break;
            case 3: case 4: case 5: return 1;
        }
        return ( _selector/2);
    }
    public static void main( String[] argv )
    {
        method0( 3 );
        System.out.println("Value:"+method1( 5 ));
        System.exit( 0 );
    }
}

```

At least two branches per Java method are defined. The input is always enumerated, as is the output if it can be covered.

Potentially Terminal Statements

The following decision statements are potentially terminal if they contain at least one statement that transfers program control out of its sequence (**RETURN**, **THROW**, **GOTO**, **BREAK**, **CONTINUE**) or that terminates the execution (**EXIT**).

- **IF** without an **ELSE**
- **SWITCH**, **FOR**
- **WHILE** or **DO...WHILE**

Java Additional Statements

Non-coverable Statements in Java

A Java statement is *non-coverable* if the statement can never possibly be executed. Code Coverage detects non-coverable statements during instrumentation and produces an error message that specifies the source file and line location of each non-coverable statement.

Code Coverage Viewer

About the Code Coverage Viewer

The Code Coverage Viewer allows you to view code coverage reports generated by the Code Coverage feature. Select a tab at the top of the Code Coverage Viewer window to select the type of report:

- A Source Report, showing the source code under analysis, highlighted with the actual coverage information.
- A Rates Report, providing detailed coverage rates for each activated coverage type.

You can use the Report Explorer to navigate through the report. Click a source code component in the Report Explorer to go to the corresponding line in the Report Viewer.

You can jump directly to the next or previous Failed test in the report by using the **Next Failed Test** or **Previous Failed Test** buttons from the Code Coverage toolbar.

You can jump directly to the next or previous Uncovered line in the Source report by using the **Next Uncovered Line** or **Previous Uncovered Line** buttons in the Code Coverage feature bar.

When viewing a Source coverage report, the Code Coverage Viewer provides several additional viewing features for refined code coverage analysis.

To open a Code Coverage report:

1. Right-click a previously executed test or application node
2. If a Code Coverage report was generated during execution of the node, select **View Report** and then **Code Coverage**.

Coverage Types

For Ada, C, and C++ the Code Coverage feature offers:

- **Function or Method code coverage:** select between function **Entries**, **Entries and exits**, or **None**. See the Function, Unit or Method Code Coverage Ada, C, and C++ for more information.
- **Call code coverage:** select **Yes** or **No** to toggle call coverage for Ada and C.
- **Block code coverage:** select the desired block coverage method. See the Block Code Coverage for Ada, C, and C++ for details.
- **Condition code coverage:** select condition coverage for Ada, C.

Please refer to the related topics for details on using each coverage type with each language.

Any of the Code Coverage types selected for instrumentation can be filtered out in the Code Coverage report stage if necessary.

To filter coverage types from the report:

1. From the **Code Coverage** menu, select **Coverage Type**.
2. Toggle each coverage type in the menu.

Alternatively, you can filter out coverage types from the Code Coverage toolbar by toggling the Code Coverage type filter buttons.

Test by Test Analysis Mode

The *Test-by-Test* analysis mode allows you to refine the coverage analysis by individually selecting the various tests that were generated during executions of the test or application node. In Test-by-Test mode, a **Tests** node is available in the Report Explorer.

When Test-by-Test analysis is disabled, the Code Coverage Viewer displays all traces as one global test.

To toggle Test-by-Test mode:

1. In the **Code Coverage Viewer** window, select the **Source** tab.
2. From the **Code Coverage** menu, select **Test-by-Test**.

To select the Tests to display in Test-by-Test mode:

1. Expand the Tests node at the top of the **Report Explorer**.
2. Select one or several tests. The **Code Coverage Viewer** provides code coverage information for the selected tests.

Reloading a Report

If a Code Coverage report has been updated since the moment you have opened it in the **Code Coverage Viewer**, you can use the **Reload** command to refresh the display:

To reload a report:

From the **Code Coverage** menu, select **Reload**.

Resetting a Report

When you run a test or application node several times, the Code Coverage results are appended to the existing report. The **Reset** command clears previous Code Coverage results and starts a new report.

To reset a report:

From the **Code Coverage** menu, select **Reset**.

Exporting a Report to HTML

Code Coverage results can be exported to an HTML file.

To export results to an HTML file:

From the **File** menu, select **Export**.

Source Report

You can use the standards keys (arrow keys, home, end, etc.) to move about and to select the source code.

Hypertext Links

The Source report provides hypertext navigation throughout the source code:

- Click a plain underlined function call to jump to the definition of the function.
- Click a dashed underlined text to view additional coverage information in a pop-up window.
- Right-click any line of code and select **Edit Source** to open the source file in the **Text Editor** at the selected line of code.

Macro Expansion

Certain macro-calls are preceded with a magnifying glass icon.

Click the magnifying glass icon to expand the macro in a pop-up window with the usual Code Coverage color codes.

Hit Count

The Hit Count tool-tip is a special capability that displays the number of times that a selected branch was covered.

Hit Count is only available when Test-by-Test analysis is disabled and when the Hit Count option has been enabled for the selected Configuration.

To activate the Hit Count tool-tip:

1. In the **Code Coverage Viewer** window, select the **Source** tab.
2. From the **Code Coverage** menu select **Hit**. The mouse cursor changes shape.
3. In the **Code Coverage Viewer** window, click a portion of covered source code to display the Hit Count tool-tip.

Cross Reference

The Cross Reference tool-tip displays the name of tests that executed a selected branch.

Cross Reference is only available in Test-by-Test mode.

To activate the Cross Reference tool-tip:

1. In the **Code Coverage Viewer** window, select the **Source** tab.
2. From the **Code Coverage** menu select **Cross Reference**. The mouse cursor changes shape.
3. In the **Code Coverage Viewer** window, click a portion of covered source code to display the **Cross Reference** tool-tip.

Comment

You can add a short comment to the generated Code Coverage report by using the Comment option in the Misc. Options Settings for Code Coverage. This can be useful to distinguish different reports generated with different

Configurations.

Comments are displayed as a magnifying glass symbol at the top of the source code report. Click the magnifying glass icon to display the comment.

Rates Report

From the Code Coverage Viewer window, select the **Rates** tab to view the coverage rate report.

Select a source code component in the Report Explorer to view the coverage rate for that particular component and the selected coverage type. Select the Root node to view coverage rates for all current files.

Code Coverage rates are updated dynamically as you navigate through the **Report Explorer** and as you select various coverage types.

Code Coverage Toolbar

The Code Coverage toolbar is useful for navigating through code coverage reports generated by the Code Coverage feature of Test RealTime.

These buttons are available when the Code Coverage Viewer is active.

- The **Previous Link** and **Next Link** buttons allow you to quickly navigate through the Failed items.
- The **Previous Uncovered Line** and **Next Uncovered Line** buttons allow you to quickly navigate through the Failed items.
- The **Failed Tests Only** or **All Tests** button toggles between the two display modes.
- The **F** button allows you to hide or show functions
- The **E** button allows you to hide or show function exits
- The **B** button allows you to hide or show statement blocks

- The **I** button allows you to hide or show implicit blocks
- The **L** button allows you to hide or show loops.

Code Coverage Viewer Preferences

The **Preferences** dialog box allows you to change the appearance of your Code Coverage reports.

To choose Code Coverage report colors and attributes:

1. Select the **Code Coverage Viewer** node:
 - **Background color:** This allows you to choose a background color for the **Code Coverage Viewer** window.
 - **Stroud Number:** This parameter modifies the results of Halstead Metrics.
2. Expand the **Code Coverage Viewer** node, and select **Styles:**
 - **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
 - **Font:** This allows you to change the font type and size for the selected style.
 - **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
 - **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.
3. Click **OK** to apply your changes.

Code Coverage Dump Driver

In C and C++, you can dump coverage trace data without using standard I/O functions by using the Code Coverage Dump Driver API contained in

the **atcapi.h** file, which is part of the Target Deployment Port

To customize the Code Coverage Dump Driver, open the Target Deployment Port directory and edit the **atcapi.h**. Follow the instructions and comments included in the source code.

Static Metrics

Source code profiling is an extremely important matter when you are planning a test campaign or for project management purposes. The graphical user interface (GUI) provides a Metrics Viewer, which provides detailed source code complexity data and statistics for your project.

Static Metric Viewer

Use the Metrics Viewer to view static testability measurements of the source files of your project. Source code metrics are created each time a source file is added to the project. Metrics are updated each time a file is modified.

The metrics are stored in **.met** metrics files alongside the actual source files.

To open the Metrics Viewer:

1. Right-click a node in the **Asset Browser** of the **Project Explorer**.
2. From the pop-up menu, select **View Metrics**.

To manually open a report file:

1. From the **File** menu, select **Open...** or click the **Open** icon in the main toolbar.
2. In the **Type** box of the File Selector, select the **.met Metrics File** file type.
3. Locate and select the metrics files that you want to open.

4. Click **OK**.

Report Explorer

The Report Explorer displays the scope of the selected nodes, or selected **.met** metrics files. Select a node to switch the Metrics Window scope to that of the selected node.

Metrics Window

Depending on the language of the analyzed source code, different pages are available:

- **Root Page - File View:** contains generic data for the entire scope
- **Root Page - Object View:** contains object related generic data for C++ and Java only
- **Component View:** displays detailed component-related metrics for each file, class, method, function, unit, procedure, etc...

The metrics window offer hyperlinks to the actual source code. Click the name of a source component to open the Text Editor at the corresponding line.

Static Metrics

The Source Code Parsers provide static metrics for the analyzed C and C++ source code.

File Level Metrics

The scope of the metrics report depends on the selection made in the Report Explorer window. This can be a file, one or several classes or any other set of source code components.

- **Comment only lines:** the number of comment lines that do not contain

any source code

- **Comments:** the total number of comment lines
- **Empty lines:** the number of lines with no content
- **Source only lines:** the number of lines of code that do not contain any comments
- **Source and comment lines:** the number of lines containing both source code and comments
- **Lines:** the number of lines in the source file
- **Comment rate:** percentage of comment lines against the total number of lines
- **Source lines:** the total number of lines of source code and empty lines

File, Class or Package, and Root Level Metrics

These numbers are the sum of metrics measured for all the components of a given file, class or package.

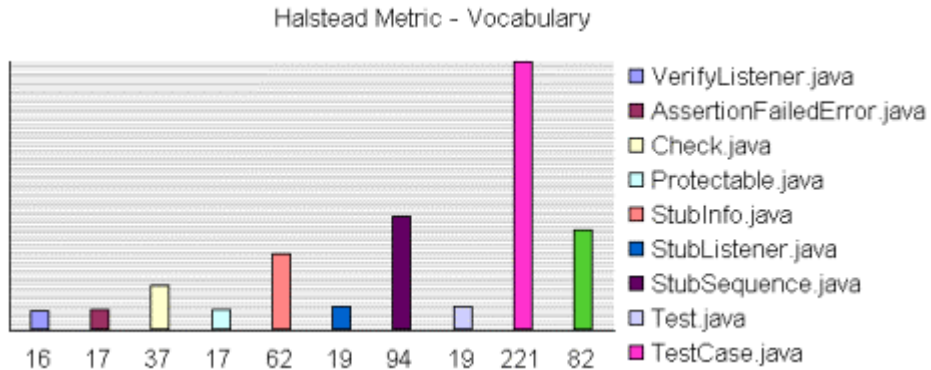
- **Total statements:** total number of statement in child nodes
- **Maximum statements:** the maximum number of statements
- **Maximum level:** the maximum nesting level
- **Maximum V(g):** the highest encountered cyclomatic number
- **Mean V(g):** the average cyclomatic number
- **Standard deviation from V(g):** deviation from the average V(g)
- **Sum of V(g):** total V(g) for the scope.

Root Level File View

At the top of the Root page, the Metrics Viewer displays a graph based on Halstead data.

On the Root page, the scope of the Metrics Viewer is the entire set of nodes below the Root node.

Halstead Graph



The following display modes are available for the Halstead graph:

- VocabularySize
- Volume
- Difficulty
- Testing Effort
- Testing Errors
- Testing Time

See the Halstead Metrics section for more information.

Metrics Summary

The scope of the metrics report depends on the selection made in the Report Explorer window. This can be a file, one or several classes or any other set of source code components.

Below the Halstead graph, the Root page displays a metrics summary table, which lists for for the source code component of the selected scope:

- **V(g):** provides a complexity estimate of the source code component
- **Statements:** shows the number of statements within the component
- **Nested Levels:** shows the highest nesting level reached in the component
- **Ext Comp Calls:** measures the number of calls to methods defined outside of the component class (C++ and Java only)
- **Ext Var Use:** measures the number of uses of attributes defined outside of the component class (C++ and Java only)

To select the File View:

1. Select **File View** in the View box of the Report Explorer.
2. Select the **Root** node in the Report Explorer to open the Root page.

Note With C and Ada source code, File View is the only available view for the Root page.

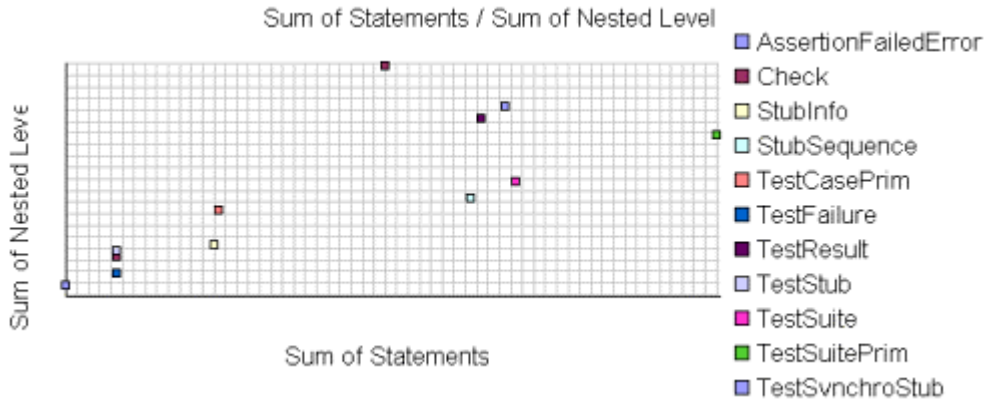
To change the Halstead Graph on the Root page:

1. From the Metrics menu, select Halstead Graph for Root Page.
2. Select another metric to display.

Root Level Object View

At the top of the Root page, the Metrics Viewer displays a graph based on the sum of data.

On the Root page, the scope of the Metrics Viewer is the entire set of nodes below the Root node.



File View is the only available view with C or Ada source code. When viewing metrics for C++ and Java, an Object View is also available.

Two modes are available for the data graph:

- Vocabulary
- Size
- Volume
- Difficulty
- Testing Effort
- Testing Errors
- Testing Time

See the Halstead Metrics section for more information.

Metrics Summary

Below the Halstead graph, the Root page displays a metrics summary table, which lists for each source code component:

- **V(g)**: provides a complexity estimate of the source code component

- **Statements:** shows the total number of statements within the object
- **Nested Levels:** shows the highest statement nesting level reached in the object
- **Ext Comp Calls:** measures the number of calls to components defined outside of the object
- **Ext Var Use:** measures the number of uses of variables defined outside of the object

Note The result of the metrics for a given object is equal to the sum of the metrics for the methods it contains.

To select the Object View:

1. Select the **Root** node in the Report Explorer to open the Root page.
2. Select **Object View** in the **View** box of the Report Explorer.

To switch the object graph mode:

1. From the Metrics menu, select Object Graph for Root Page.
2. Select ExtVarUse by ExtCompCall or Nested Level by Statement.

Halstead Metrics

Halstead complexity measurement was developed to measure a program module's complexity directly from source code, with emphasis on computational complexity. The measures were developed by the late Maurice Halstead as a means of determining a quantitative measure of complexity directly from the operators and operands in the module.

Halstead provides various indicators of the module's complexity

Halstead metrics allow you to evaluate the testing time of any C/C++ source code. These only make sense at the source file level and vary with the following parameters:

Parameter	Meaning
n_1	Number of distinct operators
n_2	Number of distinct operands
N_1	Number of operators instances
N_2	Number of operands instances

When a source file node is selected in the Metrics Viewer, the following results are displayed in the Metrics report:

Metric	Meaning	Formula
n	Vocabulary	$n_1 + n_2$
N	Size	$N_1 + N_2$
V	Volume	$N * \log_2 n$
D	Difficulty	$n_1/2 * N_2/n_2$
E	Effort	$V * D$
B	Errors	$V / 3000$
T	Testing time	E / k

In the above formulas, k is the *Stroud* number, and has a default value of **18**. You can change the value of k in the Metrics Viewer Preferences. Adjustment of the Stroud number allows you to adapt the calculation of T to the testing conditions: team background, criticity level, and so on.

When the Root node is selected, the Metrics Viewer displays the total testing time for all loaded source files.

V(g) or Cyclomatic Number

The V(g) or cyclomatic number is a measure of the complexity of a function which is correlated with difficulty in testing. The standard value is between 1 and 10.

A value of 1 means the code has no branching.

A function's cyclomatic complexity should not exceed 10.

The Metrics Viewer presents $V(g)$ of a function in the Metrics tab when the corresponding tree node is selected.

When the type of the selected node is a source file or a class, the sum of the $V(g)$ of the contained function, the mean, the maximum and the standard deviation are calculated.

At the Root level, the same statistical treatment is provided for every function in any source file.

Metrics Viewer Preferences

The **Preferences** dialog box allows you to change the appearance of your Code Coverage reports.

To choose Metrics Viewer report colors and attributes:

1. Select the **Metrics Viewer** node:
 - **Background color:** This allows you to choose a background color for the **Metrics Viewer** window.
 - **Stroud number:** This parameter modifies the results of Halstead Metrics.
2. Expand the **Metrics Viewer** node, and select **Styles**:
 - **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
 - **Font:** This allows you to change the font type and size for the selected style.
 - **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.

- **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.
3. Click **OK** to apply your changes.

Memory Profiling for C and C++

Run-time memory errors and leaks are among the most difficult errors to locate and the most important to correct. The symptoms of incorrect memory use are unpredictable and typically appear far from the cause of the error. The errors often remain undetected until triggered by a random event, so that a program can seem to work correctly when in fact it's only working by accident.

That's where the Memory Profiling feature can help you get ahead.

- You associate Memory Profiling with an existing test node or application code.
- You compile and run your application.
- The application with the Memory Profiling feature, then directs output to the Memory Profiling Viewer, which provides a detailed report of memory issues.

Memory Profiling uses Source Code Insertion Technology for C and C++.

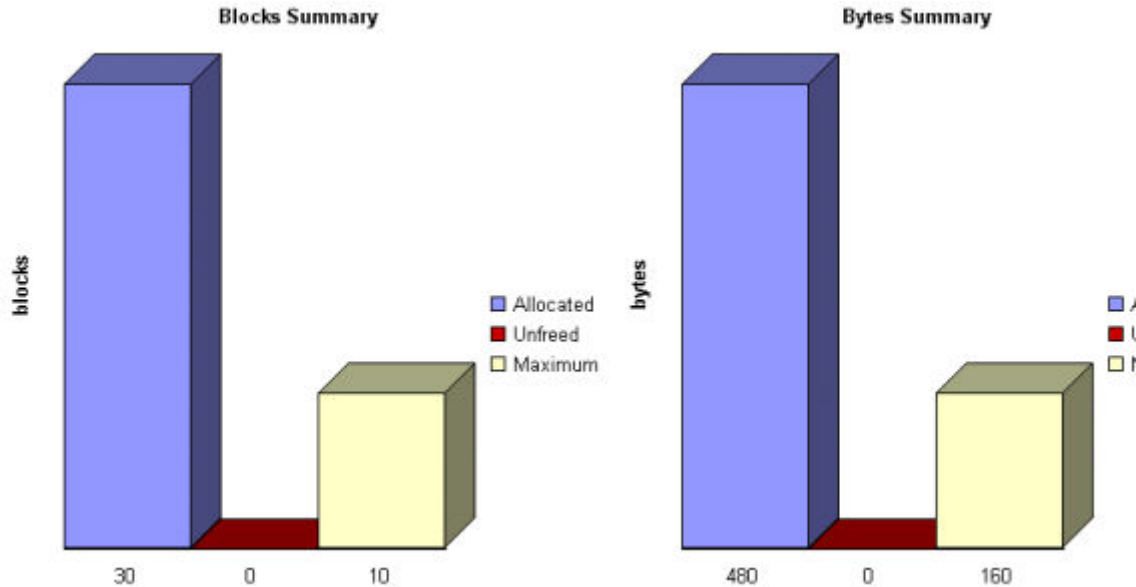
Because of the different technologies involved, Memory Profiling for Java is covered in a separate section.

Memory Profiling Results for C and C++

After execution of an instrumented application, the Memory Profiling report provides a list of the following sections:

Summary diagrams

The summary diagrams give you a quick overview of memory usage in blocks and bytes.



Where:

- **Allocated** is the total memory allocated during the execution of the application
- **Unfreed** is the memory that remains allocated after the application was terminated
- **Maximum** is the highest memory usage encountered during execution

Detailed Report

The detailed section of the report lists memory usage events, including the following errors and warnings:

- Error messages
- Warning messages

Memory Profiling Errors

Error messages indicate invalid program behavior. These are serious issues you should address before you check in code.

List of Memory Profiling Error Messages

- Freeing Freed Memory (FFM)
- Freeing Unallocated Memory (FUM)
- Late Detect Array Bounds Write (ABWL)
- Late Detect Free Memory Write (FMWL)
- Memory Allocation Failure (MAF)
- Core Dump (COR)

Freeing Freed Memory (FFM)

An FFM message indicates that the program is trying to free memory that has previously been freed.

This message can occur when one function frees the memory, but a data structure retains a pointer to that memory and later a different function tries to free the same memory. This message can also occur if the heap is corrupted.

Memory Profiling maintains a *free queue*, whose role is to actually delay memory free calls in order to compare with upcoming free calls. The length of the delay depends on the **Free queue length** and **Free queue threshold** Memory Profiling Settings. A large deferred free queue length and threshold increases the chances of catching FFM errors long after the block

has been freed. A smaller deferred free queue length and threshold limits the amount of memory on the deferred free queue, taking up less memory at run time but providing a lower level of error detection.

Freeing Unallocated Memory (FUM)

An FUM message indicates that the program is trying to free unallocated, or invalid, memory.

This message can occur when the memory is not yours to free. In addition, trying to free the following types of memory causes a FUM error:

- Memory on the stack
- Program code and data sections

Late Detect Array Bounds Write (ABWL)

An ABWL message indicates that the program wrote a value before the beginning or after the end of an allocated block of memory. Because Memory Profiling instrumented one or more components with minimal instrumentation, it cannot determine the exact location of the error. Instead, Memory Profiling performs a late detect scan after every 200 heap operations or if 10 seconds have elapsed between the currently active heap operation and the last heap operation, whichever comes first.

This message can occur when you:

- Make an array too small. For example, you fail to account for the terminating NULL in a string.
- Forget to multiply by `sizeof(type)` when you allocate an array of objects.
- Use an array index that is too large or is negative.
- Fail to NULL terminate a string.

- Are off by one when you copy elements up or down an array.

Memory Profiling actually allocates a larger block by adding a Red Zone at the beginning and end of each allocated block of memory in the program. Memory Profiling monitors these Red Zones to detect ABWL errors.

Increasing the size of the Red Zone helps Test RealTime catch bounds errors before or beyond the block.

The ABWL error does not apply to local arrays allocated on the stack.

Note Unlike Rational PurifyPlus, the ABWL error in the Rational Test RealTime Memory Profiling feature only applies to heap memory zones and not to global or local tables.

Late Detect Free Memory Write (FMWL)

An FMWL message indicates that the program wrote to memory that was freed.

This message can occur when you:

- Have a dangling pointer to a block of memory that has already been freed (caused by retaining the pointer too long or freeing the memory too soon)
- Index far off the end of a valid block
- Use a completely random pointer which happens to fall within a freed block of memory

Memory Profiling maintains a *free queue*, whose role is to actually delay memory free calls in order to compare with upcoming free calls. The length of the delay depends on the **Free queue length** and **Free queue threshold** Memory Profiling Settings. A large deferred free queue length and threshold increases the chances of catching FMWL errors. A smaller deferred free queue length and threshold limits the amount of memory on

the deferred free queue, taking up less memory at run time but providing a lower level of error detection.

Memory Allocation Failure (MAF)

An MAF message indicates that a memory allocation call failed. This message typically indicates that the program ran out of paging file space for a heap to grow. This message can also occur when a non-spreadable heap is saturated.

After Memory Profiling displays the MAF message, a memory allocation call returns *NULL* in the normal manner. Ideally, programs should handle allocation failures.

Core Dump (COR)

A COR message indicates that the program generated a UNIX core dump. This message can only occur when the program is running on a UNIX target platform.

Memory Profiling Warnings

Warning messages indicate a situation in which the program might not fail immediately, but might later fail sporadically, often without any apparent reason and with unexpected results. Warning messages often pinpoint serious issues you should investigate before you check in code.

List of Memory Profiling Warning Messages

- Memory in Use (MIU)
- Memory Leak (MLK)
- Potential Memory Leak (MPK)
- File in Use (FIU)

- Signal Handled (SIG)

Memory in Use (MIU)

An MIU message indicates heap allocations to which the program has a pointer.

Note At exit, small amounts of memory in use in programs that run for a short time are not significant. However, you should fix large amounts of memory in use in long running programs to avoid out-of-memory problems.

Memory Profiling generates a list of memory blocks in use when you activate the **MIU Memory In Use** option in the Memory Profiling Settings.

Memory Leak (MLK)

An **MLK** message describes leaked heap memory. There are no pointers to this block, or to anywhere within this block.

Memory Profiling generates a list of leaked memory blocks when you activate the **MLK Memory Leak** option in the Memory Profiling Settings.

This message can occur when you allocate memory locally in some function and exit the function without first freeing the memory. This message can also occur when the last pointer referencing a block of memory is cleared, changed, or goes out of scope. If the section of the program where the memory is allocated and leaked is executed repeatedly, you might eventually run out of swap space, causing slow downs and crashes. This is a serious problem for long-running, interactive programs.

To track memory leaks, examine the allocation location call stack where the memory was allocated and determine where it should have been freed.

Memory Potential Leak (MPK)

An **MPK** message describes heap memory that might have been leaked. There are no pointers to the start of the block, but there appear to be pointers pointing somewhere within the block. In order to free this memory, the program must subtract an offset from the pointer to the interior of the block. In general, you should consider a potential leak to be an actual leak until you can prove that it is not by identifying the code that performs this subtraction.

Memory in use can appear as an **MPK** if the pointer returned by some allocation function is offset. This message can also occur when you reference a substring within a large string. Another example occurs when a pointer to a C++ object is cast to the second or later base class of a multiple-inherited object and it is offset past the other base class objects.

Alternatively, leaked memory might appear as an **MPK** if some non-pointer integer within the program space, when interpreted as a pointer, points within an otherwise leaked block of memory. However, this condition is rare.

Inspection of the code should easily differentiate between different causes of **MPK** messages.

Memory Profiling generates a list of potentially leaked memory blocks when you activate the **MPK Memory Potential Leak** option in the Memory Profiling Settings.

File in Use (FIU)

An FIU message indicates a file that was opened, but never closed. An FIU message can indicate that the program has a resource leak.

Memory Profiling generates a list of files in use when you activate the **FIU Files In Use** option in the Memory Profiling Settings.

Signal Handled (SIG)

A **SIG** message indicates that a system signal has been received.

Memory Profiling generates a list of received signals when you activate the **SIG Signal Handled** option in the Memory Profiling Settings.

Memory Profiling User Heap in C and C++

When using Memory Profiling on embedded or real-time target platforms, you might encounter one of the following situations:

- **Situation 1:** There are no provisions for **malloc**, **calloc**, **realloc** or **free** statements on the target platform.

Your application uses custom heap management routines that may use a user API. Such routines could, for example, be based on a static buffer that performs allocation and free actions.

In this case, you need to customize the memory heap parameters **RTRT_DO_MALLOC** and **RTRT_DO_FREE** in the TDP to use the custom **malloc** and **free** functions.

In this case, you can access the custom API functions.

- **Situation 2:** There are partial implementations of **malloc**, **calloc**, **realloc** or **free** on the target, but other functions provide methods of allocating or freeing heap memory.

In this case, you do not have access to any custom API. This requires customization of the Target Deployment Port. Please refer to the **Target Deployment Guide** provided with the TDP Editor.

In both of the above situations, Memory Profiling can use the heap management routines to detect memory leaks, array bounds and other memory-related defects.

Note Application pointers and block sizes can be modified by Memory Profiling in order to detect ABWL errors (Late Detect Array Bounds

Write). Actual-pointer and actual-size refer to the memory data handled by Memory Profiling, whereas user pointer and user-size refer to the memory handled natively by the application-under-analysis. This distinction is important for the Memory Profiling ABWL and Red zone settings.

Target Deployment Port API

The Target Deployment Port library provides the following API for Memory Profiling:

```
void * _PurifyLTHeapAction ( _PurifyLT_API_ACTION, void *,
RTRT_U_INT32, RTRT_U_INT8 );
```

In the function **_PurifyLTHeapAction** the first parameter is the type of action that will be or has been performed on the memory block pointed by the second parameter. The following actions can be used:

```
typedef enum {
    _PurifyLT_API_ALLOC,
    _PurifyLT_API_BEFORE_REALLOC,
    _PurifyLT_API_FREE
} _PurifyLT_API_ACTION;
```

The third parameter is the size of the block. The fourth parameter is either of the following constants:

```
#define _PurifyLT_NO_DELAYED_FREE    0
#define _PurifyLT_DELAYED_FREE      1
```

If an allocation or free has a size of 0 this fourth parameter indicates a delayed free in order to detect FWML (Late Detect Free Memory Write) and FFM (Freeing Freed Memory) errors. See the section on Memory Profiling Configuration Settings for Detect FFM, Detect FMWL, Free Queue Length and Free Queue Size.

A freed delay can only be performed if the block can be freed with **RTRT_DO_FREE** (situation 1) or ANSI **free** (situation 2). For example, if a function requires more parameters than the pointer to de-allocate, then the FMWL and FFM error detection cannot be supported and FFM errors will

be indicated by an FUM (Freeing Unallocated Memory) error instead.

The following function returns the size of an allocated block, or 0 if the block was not declared to Memory Profiling. This allows you to implement a library function similar to the `msize` from Microsoft Visual 6.0.

```
RTRT_SIZE_T _PurifyLTheapPtrSize ( void * );
```

The following function returns the actual-size of a memory block, depending on the size requested. Call this function before the actual allocation to find out the quantity of memory that is available for the block and the contiguous red zones that are to be monitored by Memory Profiling.

```
RTRT_SIZE_T _PurifyLTheapActualSize ( RTRT_SIZE_T );
```

Examples

In the following examples, **my_malloc**, **my_realloc**, **my_free** and **my_msize** demonstrate the four supported memory heap behaviors.

The following routine declares an allocation:

```
void *my_malloc ( int partId, size_t size )
{
    void *ret;
    size_t actual_size = _PurifyLTheapActualSize(size);
    /* Here is any user code making ret a pointer to a heap or
       simulated heap memory block of actual_size bytes */
    ...
    /* After comes Memory Profiling action */
    return _PurifyLTheapAction ( _PurifyLT_API_ALLOC, ret,
    size, 0 );
    /* The user-pointer is returned */
}
```

In situation 2, where you have access to a custom memory heap API, replace the "... " with the actual *malloc* API function.

For a **my_calloc**(*size_t nelem, size_t elsize*), pass on *nelem*elsize* as the third parameter of the **_PurifyLTheapAction** function. In this case, you might need to replace this operation with a function that takes into account

the alignments of elements.

To declare a reallocation, two operations are required:

```
void *my_realloc ( int partId, void * ptr, size_t size )
{
    void *ret;
    size_t actual_size = _PurifyLTheapActualSize(size);
    /* Before comes first Memory Profiling action */
    ret = _PurifyLTheapAction ( _PurifyLT_API_BEFORE_REALLOC,
ptr, size, 0 );
    /* ret now contains the actual-pointer */
    /* Here is any user code making ret a reallocated pointer
to a heap or
    simulated heap memory block of actual_size bytes */
    ...
    /* After comes second Memory Profiling action */
    return _PurifyLTheapAction ( _PurifyLT_API_ALLOC, ret,
size, 0 );
    /* The user-pointer is returned */
}
```

To free memory without using the delay:

```
void my_free ( int partId, void * ptr )
{
    /* Memory Profiling action comes first */
    void *ret = _PurifyLTheapAction ( _PurifyLT_API_FREE, ptr,
0, 0 );
    /* Any code insuring actual deallocation of ret */
}
```

To free memory using a delay:

```
void my_free ( int partId, void * ptr )
{
    /* Memory Profiling action comes first */
    void *ret = _PurifyLTheapAction ( _PurifyLT_API_FREE, ptr,
0, 1 );
    /* Nothing to do here */
}
```

To obtain the user size of a block:

```
size_t my_msize ( int partId, void * ptr )
{
    return _PurifyLTheapPtrSize ( ptr );
}
```

Use the following macros to save customization time when dealing with

functions that have the same prototypes as the standard ANSI functions:

```

#define _PurifyLT_MALLOC_LIKE(func) \
void *RTRT_CONCAT_MACRO(usr_,func) ( RTRT_SIZE_T size ) \
{ \
    void *ret; \
    ret = func ( _PurifyLTHeapActualSize ( size ) ); \
    return _PurifyLTHeapAction ( _PurifyLT_API_ALLOC, ret, \
size, 0 ); \
}
#define _PurifyLT_CALLOC_LIKE(func) \
void *RTRT_CONCAT_MACRO(usr_,func) ( RTRT_SIZE_T nelem, \
RTRT_SIZE_T elsize ) \
{ \
    void *ret; \
    ret = func ( _PurifyLTHeapActualSize ( nelem * elsize ) ); \
    return _PurifyLTHeapAction ( _PurifyLT_API_ALLOC, ret, \
nelem * elsize, 0 ); \
}
#define _PurifyLT_REALLOC_LIKE(func,delayed_free) \
void *RTRT_CONCAT_MACRO(usr_,func) ( void *ptr, RTRT_SIZE_T \
size ) \
{ \
    void *ret; \
    ret = func ( _PurifyLTHeapAction ( \
_PurifyLT_API_BEFORE_REALLOC, \
ptr, size, delayed_free \
), \
_PurifyLTHeapActualSize ( size ) ); \
    return _PurifyLTHeapAction ( _PurifyLT_API_ALLOC, ret, \
size, 0 ); \
}
#define _PurifyLT_FREE_LIKE(func,delayed_free) \
void RTRT_CONCAT_MACRO(usr_,func) ( void *ptr ) \
{ \
    if ( delayed_free ) \
    { \
        _PurifyLTHeapAction ( _PurifyLT_API_FREE, ptr, 0, \
delayed_free ); \
    } \
    else \
    { \
        func ( _PurifyLTHeapAction ( _PurifyLT_API_FREE, ptr, 0, \
delayed_free ) ); \
    } \
}

```


Using the Memory Profiling Viewer

Memory Profiling results for C, C++ and Ada are displayed in the Memory Profiling Viewer.

Memory Profiling for Java uses the Report Viewer

Error and Warning Filter

The Memory Profiling Viewer for C, C++ and Ada allows you to filter out any particular type of Error or Warning message from the report.

To filter out error or warning messages:

1. Select an active **Memory Profiling Viewer** window.
2. From the Memory Profiling menu, select Errors and Warnings.
3. Select or clear the type of message that you want to show or hide.

Reloading a Report

If a Memory Profiling report has been updated since the moment you have opened it in the Memory Profiling Viewer, you can use the Reload command to refresh the display:

To reload a report:

1. From the View Toolbar, click the **Reload** button.

Resetting a Report

When you run a test or application node several times, the Memory Profiling results are appended to the existing report. The **Reset** command clears previous Memory Profiling results and starts a new report.

To reset a report:

1. From the View Toolbar, click the **Reset** button.

Exporting a Report to HTML

Memory Profiling results can be exported to an HTML file.

To export results to an HTML file:

From the **File** menu, select **Export**.

Memory Profiling Viewer Preferences

The **Preferences** dialog box allows you to change the appearance of your Memory Profiling reports for C, C++ and Ada.

To choose Memory Profiling report colors and attributes:

1. Select the Memory Profiling Viewer node:
 - **Background color:** This allows you to choose a background color for the **Memory Profiling Viewer** window.
2. Expand the **Memory Profiling Viewer** node, and select **Styles:**
 - **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
 - **Font:** This allows you to change the font type and size for the selected style.
 - **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
 - **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.
3. Click **OK** to apply your changes.

Memory Profiling for Java

Run-time memory problems are among the most difficult errors to locate and the most important to correct. The symptoms of incorrect memory use are unpredictable and typically appear far from the cause of the error. The issue often remain undetected until triggered by a random event, so that a program can seem to work correctly when in fact it's only working by accident.

That's where the Memory Profiling feature can help you get ahead.

- You associate Memory Profiling with an existing test node or Application code.
- You compile and run your application.
- The application with the Memory Profiling feature, then directs output to the Memory Profiling Viewer, which provides a detailed report of memory issues.

The Java language differs from other programming languages, among other aspects, by the way memory is managed by the Java Virtual Machine (JVM).

The technique used is the JVMPI Agent technology for Java.

Memory Profiling Results for Java

After execution of an instrumented application, the Memory Profiling report displays:

- In the **Report Explorer** window: a list of available snapshots
- In the **Memory Profiling** window: the contents of the selected Memory Profiling snapshot

Report Explorer

The Report Explorer window displays a **Test** for each execution of the application node, or for a test node when using Component Testing for Java in Rational Test RealTime. Inside each test, a **Snapshot** report is created for each Memory Profiling snapshot.

Method Snapshots

The Memory Profiling report displays snapshot data for each method that has performed an allocation. If the Java CLASSPATH is correctly set, you can click blue method names to open the corresponding source code in the Text Editor. System methods are displayed in black and cannot be clicked.

Method data is reset after each snapshot.

For each method, the report lists:

- **Method:** The method name. Blue method names are hyperlinks to the source code under analysis
- **Allocated Objects:** The number of objects allocated since the previous snapshot
- **Allocated Bytes:** The total number of bytes used by the objects allocated by the method since the previous snapshot
- **Local + D Allocated Objects:** The number of objects allocated by the method since the previous snapshot as well as any descendants called by the method
- **Local + D Allocated Bytes:** The total number of bytes used by the objects allocated by the method since the previous snapshot and its descendants

Referenced Objects

If you selected the **With objects** filter option in the JVMPI Settings dialog box, the report can display, for each method, a list of objects created by the method and object-related data.

From the **Memory Profiling** menu, select **Hide/Show Referenced Objects**.

For each object, the report lists:

- **Reference Object Class:** The name of the object class. Blue class names are hyperlinks to the source code under analysis.
- **Referenced Objects:** The number of objects that exist at the moment the snapshot was taken
- **Referenced Bytes:** The total number of bytes used by the referenced objects

Differential Reports

The Memory Profile report can display differential data between two snapshots within the same Test. This allows you to compare the referenced objects. There are two *diff* modes:

- Automatic differential report with the previous snapshot
- User differential report

Differential reports add the following columns to the current Memory Profiling snapshot report:

- **Referenced Objects Diff AUTO:** Shows the difference in the number of referenced objects for the same method in the current snapshot as compared to the previous snapshot
- **Referenced Bytes Diff AUTO :** Shows the difference in the memory used by the referenced objects for the same method in the current

snapshot as compared to the previous snapshot

- **Referenced Objects Diff USER:** Shows the difference in the number of referenced objects for the same method in the current snapshot as compared to the user-selected snapshot
- **Referenced Bytes Diff USER:** Shows the difference in the memory used by the referenced objects for the same method in the current snapshot as compared to the user-selected snapshot

To add or remove data to the report:

1. From the **Memory Profiling** menu, select **Hide/Show Data**.
2. Toggle the data that you want to hide or display

To sort the report:

- In the **Memory Profiling** window, click a column label to sort the table on that value.

To obtain a differential report:

- From the **Memory Profiling** menu, select **Diff with Previous Referenced Objects**.

To obtain a user differential report:

1. In the **Report Explorer**, select the current snapshot
2. Right-click another snapshot in the same Test node and select **Diff Report**.

JVMPI Technology

Memory Profiling for Java uses a special dynamic library, known as the Memory Profiling Agent, to provide advanced reports on Java Virtual Machine (JVM) memory usage.

Garbage Collection

JVMs implement a heap that stores all objects created by the Java code.

Memory for new objects is dynamically allocated on the heap. The JVM automatically frees objects that are no longer referenced by the program, preventing many potential memory issues that exist in other languages. This process is called *garbage collection*.

In addition to freeing unreferenced objects, a garbage collector may also reduce heap fragmentation, which occurs through the course of normal program execution. On a virtual memory system, the extra paging required to service an ever growing heap can degrade the performance of the executing program.

JVMPI Agent

Because of the memory handling features included in the JVM, Memory Profiling for Java is quite different from the feature provided for other languages. Instead of Source Code Insertion technology, the Java implementation uses a JVM Profiler Interface (JVMPI) Agent whose task is to monitor JVM memory usage and to provide a memory dump upon request.

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

- Method entries and exits
- Object and primitive type allocations

The JVMPI Agent is a dynamic library **DLL** or **lib.so** depending on the platform used that is loaded as an option on the command line that launches the Java program.

During execution, when the agent receives a snapshot trigger request, it can either 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. In some cases, such information may be difficult to interpret correctly.

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.

Performance Profiling

The Performance Profiling feature puts successful performance engineering within your grasp. It provides complete, accurate performance data—and provides it in an understandable and usable format so that you can see exactly where your code is least efficient. Using Performance Profiling, you can make virtually any program run faster. And you can measure the results.

Performance Profiling measures performance for every component in C or C++ source code, in real-time, and on both native or embedded target platforms. Performance Profiling works by instrumenting the C and C++ source code of your application. After compilation, the instrumented code reports back to Test RealTime after the execution of the application.

- You associate Performance Profiling with an existing test or application code.
- You build and execute your code in Test RealTime.

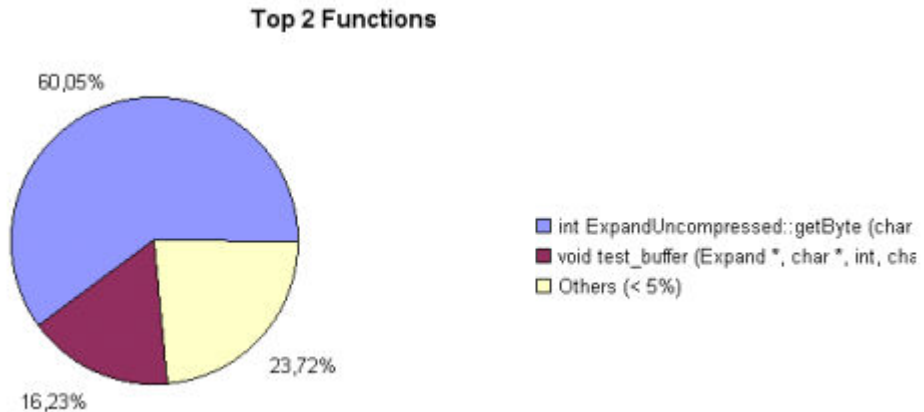
- The application under test, instrumented with the Performance Profiling feature, then directs output to the Performance Profiling Viewer, which provides a detailed report of memory issues.

Performance Profiling Results

The Performance Profiling report provides function profiling data for your program and its components so that you can see exactly where your program spends most of its time.

Top Functions Graph

This section of the report provides a high level view of the largest time consumers detected by Performance Profiling in your application.



Performance Summary

This section of the report indicates, for each instrumented function, procedure or method (collectively referred to as functions), the following data:

- **Calls:** The number times the function was called

- **Function (F) time:** The time required to execute the code in a function exclusive of any calls to its descendants
- **Function+descendant (F+D) time:** The total time required to execute the code in a function and in any function it calls.

Note that since each of the descendants may have been called by other functions, it is not sufficient to simply add the descendants' F+D to the caller function's F. In fact, it is possible for the descendants' F+D to be larger than the calling function's F+D. The following example demonstrates three functions **a**, **b** and **c**, where both **a** and **b** each call **c** once:

function	F	F+D
a	5	15
b	5	15
c	20	20

The F+D value of **a** is less than the F+D of **c**. This is because the F+D of **a** (15) equals the F of **a** (5) plus one half the F+D of **c** ($20/2=10$).

- **F Time (% of root)** and **F+D Time (% of root):** Same as above, expressed in percentage of total execution time
- **Average F Time:** The average time spent executing the function each time it was called

Performance Profiling SCI Dump Driver

In C and C++, you can dump profiling trace data without using standard I/O functions by using the Performance Profiling Dump Driver API contained in the **atqapi.h** file, which is part of the Target Deployment Port

To customize the Performance Profiling Dump Driver, open the Target

Deployment Port directory and edit the **atqapi.h**. Follow the instructions and comments included in the source code.

Performance Profiling Viewer Preferences

The **Preferences** dialog box allows you to change the appearance of your Performance Profiling reports.

To choose Performance Profiling report colors and attributes:

1. Select the Performance Profiling Viewer node:
 - **Background color:** This allows you to choose a background color for the **Performance Profiling Viewer** window.
2. Expand the Performance Profiling Viewer node, and select Styles:
 - **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
 - **Font:** This allows you to change the font type and size for the selected style.
 - **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
 - **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.
3. Click **OK** to apply your changes.

Using the Performance Profiling Viewer

The product GUI displays Performance Profiling results in the Performance Profiling Viewer.

Reloading a Report

If a Performance Profiling report has been updated since the moment you have opened it in the Performance Profiling Viewer, you can use the Reload command to refresh the display:

To reload a report:

1. From the View Toolbar, click the **Reload** button.

Resetting a Report

When you run a test or application node several times, the Performance Profiling results are appended to the existing report. The Reset command clears previous Performance Profiling results and starts a new report.

To reset a report:

1. From the View Toolbar, click the **Reset** button.

Exporting a Report to HTML

Performance Profiling results can be exported to an HTML file.

To export results to an HTML file:

From the File menu, select Export

Runtime Tracing

About Runtime Tracing

Runtime Tracing is a feature for monitoring real-time dynamic interaction analysis. Runtime Tracing uses exclusive source-code instrumentation technology to generate trace data, which is turned into UML sequence diagrams within the Test RealTime GUI.

In Test RealTime, Runtime Tracing can run either as a standalone product, or in conjunction with a Test RealTime component or system testing test node.

- You associate Performance Profiling with an existing test or application code.
- You build and execute your code in Test RealTime.
- The application under test, instrumented with the Runtime Tracing feature, then directs output to the UML/SD Viewer, which provides a real-time UML Sequence Diagram of your application's behavior.

Understanding Runtime Tracing UML Sequence Diagrams

Below are a series of examples of Runtime Tracing UML Sequence Diagram output:

- Object instances
- C++ exceptions
- File instances
- Loops

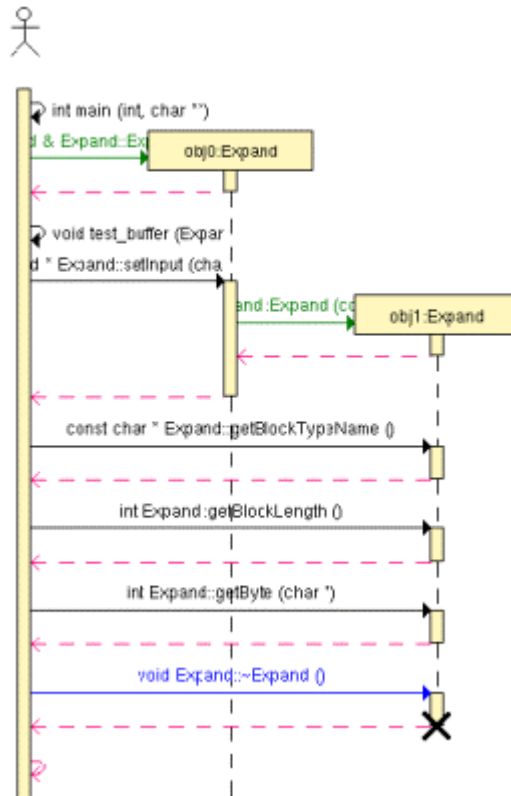
Object Instances and Routine I/O

The lifeline of an object is represented in the UML/SD Viewer as shown below.

The instance creation box displays the name of the instance.

Example

Below is an example of object lifelines generated by Runtime Tracing from a C++ application.



In this C++ example, functions and static methods are attached to the World instance.

Objects are labelled with **obj<number>:<classname>**

The black cross represents the destruction of the instance.

Constructors are displayed in green.

Destructors are blue.

Return messages are dotted red lines.

Other functions and methods are black.

The `main()` is a function of the World instance called by the same World instance.

To jump to the corresponding portion of source code:

- Double-click an element of the object lifeline to open the Text Editor at the corresponding line in the source code.

To jump to the beginning or to the end of an instance:

- Right-click an element of the object lifeline and select **Go to Head** or **Go to Destruction** in the pop-up menu.

To filter an instance out of the UML sequence diagram:

- Right-click an element of the object lifeline and select **Filter instance** in the pop-up menu.

Runtime Tracing with a Test Node

When Runtime Tracing is activated with a Component Testing or System Testing test node, monitoring a UML sequence diagram of the execution from Runtime Tracing is a matter of including Runtime Tracing in the Build options of an existing test node.

If however you are using Runtime Tracing on its own, you need to create an application node in the Project Explorer, and associate it with the source files that you want to monitor.

To set the Runtime Tracing option:

1. From the Build toolbar, click the **Options** button.
2. In the options list, select **Runtime Tracing**.
3. Click anywhere outside the options list to close it.

Next time you run a **Make** command on the selected test node, a Runtime Tracing UML sequence diagram will be produced simultaneously with the standard test output.

To View Runtime Tracing output:

- Runtime Tracing output is displayed, with the UML/SD Viewer, in the same UML sequence diagram as the standard test's graphical output.

Multi-Thread Support

Runtime Tracing can be configured for use in a multi-threaded environment such as Posix, Solaris and Windows.

Multi-thread mode protects Target Deployment Port global variables against concurrent access. This causes a significant increase in Target Deployment Port size as well as an impact on performance. Therefore, select this option only when necessary.

Multi thread settings:

These settings are ignored if you are not using a multi-threaded environment. To change these settings, use the Runtime Tracing Control Settings dialog box.

- **Maximum number of threads:** This value sets the size of the thread management table inside the Target Deployment Port. Lower values save memory on the target platform. Higher values allow more simultaneous threads.
- **Dump note on thread creation:** When selected, the UML Sequence Diagram displays a note ("**Thread Creation**") each time a new thread is created.
- **Dump note on thread schedule:** When selected, the UML Sequence Diagram displays a note ("**Thread Schedule**") each time a thread's schedule is changed.

Partial Trace Flush

When using this mode, the Target Deployment Port only sends messages related to instance creation and destruction, or user notes. All other events are ignored. This can be useful to reduce the output of trace.

When Partial Trace Flush mode is enabled, message dump can be toggled on and off during trace execution.

The Partial Trace Flush settings are located in the **Runtime Tracing Settings** dialog box.

To set Partial Trace Flush from the Node Settings:

1. In the Project Explorer, click the Open Settings... button.
2. Select one or several nodes in the Configuration pane.
3. Select the Runtime Analysis node and the Runtime Tracing node.
4. Select Runtime Tracing Control.
5. Set the **Partial trace flush** setting to **Yes** or **No** to activate or disable the mode.
6. When you have finished, click **OK** to validate the changes.

To toggle message dump from within the source code:

To do this, you can use the Runtime Tracing pragma user directives:

- `_ATT_START_DUMP`
- `_ATT_STOP_DUMP`
- `_ATT_TOGGLE_DUMP`
- `_ATT_DUMP_STACK`

See the **Reference Manual** for more information about pragma directives.

To control message dump through a user signal (native UNIX only):

This capability is available only when using a native UNIX target platform.

Under UNIX, the kill command allows you to send a user signal to a process. Runtime Tracing can use this signal to toggle message dump on and off.

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select one or several nodes in the Configuration pane.
3. Select the **Runtime Analysis** node and the **Runtime Tracing** node.
4. Select **Runtime Tracing Control**.
5. Set the **Partial trace flush** setting to **Yes** or **No** to activate or disable the mode.
6. When you have finished, click **OK** to validate the changes.

Note By default, the expected signal is **SIGUSR1**, but you can change this by setting the **ATT_SIGNAL_DUMP** environment variable to the desired signal number. See the **Reference Manual** for more information about environment variables.

Trace Item Buffer

Buffering allows you to reduce formatting and I/O processing at time-critical steps by telling the Target Deployment Port to only output trace information when its buffer is full or at user-controlled points.

This can prove useful when using Runtime Tracing on real-time applications, as you can control buffer flush from within the source-under-trace.

To activate or de-activate trace item buffering:

1. In the **Project Explorer**, click the **Open Settings...** button.

2. Select one or several nodes in the Configuration pane.
3. Select the **Runtime Analysis** node and the **Runtime Tracing** node.
4. Select **Runtime Tracing Control**.
5. Set the **Buffer trace items** setting to **Yes** or **No** to activate or disable the mode.
6. Set the size of the buffer in the **Items buffer size** box.
7. When you have finished, click **OK** to validate the changes.

A smaller buffer optimizes memory usage on the target platform, whereas a larger buffer improves performance of the real-time trace. The default value is 64.

To flush the trace buffer through a user directive:

It can be useful to flush the buffer before entering a time-critical part of the application-under-trace. You can do this by adding the `_ATT_FLUSH_ITEMS` user directive to the source-under-trace.

Note See Runtime Tracing pragma directives in the **Reference Manual** to control Target Deployment Port buffering from within the source code.

Splitting Trace Files

During execution, Runtime Tracing generates a **.tdf** dynamic file. When a large application is instrumented, the size of the **.tdf** file can impact performance of UML/SD Viewer.

Splitting trace files allows you to split the **.tdf** trace file into smaller files, resulting in faster display of the UML Sequence Diagram and to optimize memory usage. However, split trace files cannot be used simultaneously with On-the-Fly tracing.

When displaying split **.tdf** files, Runtime Tracing adds Synchronization elements to the UML sequence diagram to ensure that all instance lifelines are synchronized.

To set Split Trace mode:

1. From the Project Explorer, select the highest level node from which you want to activate split trace mode, the Workspace for instance.
2. Right-click the node, and select **Settings...** from the pop-up menu.
3. In the **Configuration Settings** dialog, select the **Runtime Tracing** tab
4. From the options box, select **Miscellaneous options**.
5. Select **Override parent settings** to allow modification of the node's settings.
6. Select **Split trace** in the Split Runtime Tracing area.
7. Set the **Size (Kb)** of each split **.tdf**. The default size is 5000 Kb.
8. Specify a **Prefix** for the split **.tdf** filenames. The prefix is followed by a 4-digit number that identifies each file.
9. Click **OK**.

Note The total size of split **.tdf** files is slightly larger than the size of a single **.tdf** file, because each file contains additional context information.

Automated Testing

3

The test features provided with Rational Test RealTime allow you to submit your application to a robust test campaign. Each feature uses a different approach to the software testing problem, from the use of test drivers stimulating the code under test, to source code instrumentation testing internal behavior from inside the running application.

Using Test Features

The test features provided with Test RealTime allow you to submit your application to a robust test campaign. Each feature uses a different approach to the software testing problem, from the use of test drivers stimulating the code under test, to source code instrumentation testing internal behavior from inside the running application.

- Component Testing for C and Ada performs black box or functional testing of software components independently of other units in the same system.
- Component Testing for C++ uses object-oriented techniques to address embedded software testing.
- System Testing for C is dedicated to testing message-based applications.

These test features each use a dedicated scripting language for writing specialized test cases. Test RealTime's test features can also be used together with any of the runtime analysis features.

To use a test feature:

Here is a rundown of the main steps to using the Test RealTime test features:

1. Set up a new project in Test RealTime. This can be done automatically with the New Project Wizard.
2. Follow the Activity Wizard to add your application source files to the workspace.
3. Select the source files under test with the Test Generation Wizard to create a test node. The Wizard guides you through process of selecting the right test feature for your needs.
4. Develop the test cases by completing the automatically generated test scripts with the corresponding script language and native code.
5. Use the Project Explorer to set up the test campaign and add any additional runtime analysis or test nodes.
6. Run the test campaign to builds and execute a test driver with the application under test.
7. View and analyze the generated test reports.

Component Testing for C and Ada

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.

Overview

Basically, Component Testing for C and Ada interacts with your source code through a scripting language. There are three scripting languages, each corresponding to the language used by your source code:

- C Test Script Language
- Ada Test Script Language

The Rational Test RealTime Reference Manual contains full reference information about each of these languages.

You use the Test RealTime GUI to set up your test campaign, write a C or Ada **.ptu** test script, run your tests and view the results.

When the test is executed, Component Testing compiles both the test scripts and the source under test, then instruments the source code if necessary and generates a test driver.

The test driver, TDP, stubs and dependency files make up a test harness. The test harness interacts with the source code under test and produces test results.

Both the instrumented application and the test driver provide output data which is displayed within Test RealTime.

Note If your are upgrading from a previous version of Test RealTime, please see how to import V2001 Component Testing files.

C and Ada Testing Overview

Integrated, Simulated and Additional Files

Component Testing for C and Ada

When creating a Component Testing test node for C and Ada, the Component Testing wizard offers the following options for specifying dependencies of the source code under test:

- Integrated files

- Simulated files
- Additional files

Integrated Files

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

The Component Testing wizard analyzes integrated 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 file in the **.ptu** test script.

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

This option gives the Component Testing wizard a list of source files to simulate or stub upon execution of the test.

A stub is a dummy software component designed to replace a component that the code under test relies on, but cannot use for practicality or availability reasons. A stub can simulate the response of the stubbed component.

The Component Testing 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 in the **.ptu** test script.

By default, no simulation instructions are generated.

Additional Files

Additional files are merely dependency files that are added to the Component Testing test node, but ignored by the source code parser. Additional files are compiled with the rest of the test node but are not instrumented.

For example, Microsoft Visual C resource files can be compiled inside a test node by specifying them as additional files.

You can toggle a source file from *under test* to *additional* by using the Properties Window dialog box.

Tester Configuration

The Tester Configuration dialog box allows you to configure the Component Testing test driver.

To open the Tester Configuration dialog box:

1. In the **Project Explorer**, right-click a **.ptu** test script.
2. From the pop-up menu, select **Tester Configuration**.

Service/Test Tab

Use this tab to select one or several **SERVICES** or **TESTS** as defined in the **.ptu** test script. During execution, the Component Testing node plays the selected **SERVICES** or **TESTS**.

Family Tab

Use this tab to select one or several families as defined in the **.ptu** test script. During execution, the Component Testing node plays the selected families.

Importing V2001A Component Testing Files

The file format of ATTOL UniTest and Test RealTime v2001A Component Testing for C and Ada is not compatible with the current v2002 *Release 2* file format.

This means that the **.prj**, **.cmp**, and **.ses** files must be imported and converted in order to be used in a v2002 *Release 2* Test RealTime project.

The **Import** feature creates a new workspace with the updated Component Testing script files.

Note This problem only affects the Component Testing for C and Ada feature. Use previous Component Testing for C++ and System Testing tests in your current v2002 *Release 2* projects without importing them.

1. From the **File** menu, select **Import**.
2. In the window **Import V2001A Component Testing Files Into a New Workspace**, select the **Add...** button and then select those V2001 Component Testing files (with the extension **.prj**, **.cmp**, and **.ses**) you wish to import.
3. Click the **OK** button
4. In the window **Name Workspace**, type in a name for the new workspace and click **OK**.

Limitations

This feature imports the session, project and campaign data from the old version of Component Testing, including references to and from test scripts as well as tested and integrated source files.

After the importation, you must manually check and update the following items:

- **Target Deployment Port:** Use the TDP Editor to reconfigure any custom ATTOL Target Package settings. The Target Deployment Guide contains advanced information about upgrading from an old Target Package.
- **Configuration Settings:** The Import feature retrieves **-D** condition information and *include* directories. Check the **General**, **Build** and **Unit Testing** tabs of the **Configuration Settings** dialog box to identify any other settings that need updating.
- **Service and Family parameters:** These are not imported and require manual updating with the Tester Configuration function.

Options and Settings

Array and Structure Display

The Array and Structure Display option indicates the way in which Component Testing processes variable array and structure statements. This option is part of the Component Testing Settings for C dialog box.

Standard Array and Structure Display

This option processes arrays and structures according to the statement with which they are declared. This is the default operating mode of Component Testing. The default report format is the **Standard** editing.

Extended Array and Structure Display

Arrays of variables may be processed after the keywords **VAR** or **ARRAY**, and structured variables after the keywords **VAR**, **ARRAY**, or **STRUCTURE**:

- After a **VAR** statement, each element in the array is initialized and tested one by one. Likewise, each member of a structure that is an array is initialized and tested element by element.

- After an **ARRAY** statement, the entire array is initialized and checked. Likewise, each member of a structure is initialized and checked element by element.
- After a **STRUCTURE** statement, the whole of the structure is initialized and checked.

When **Extended editing** is selected, Component Testing interprets **ARRAY** and **STRUCTURE** statements as **VAR** statements.

The output records in the unit test report are then detailed for each element in the array or structure.

Note This setting slightly slows down the test execution because checks are performed on each element in the array.

Packed Array and Structure Display

This command has the opposite effect of the Extended editing option. When **Packed editing** is selected, Component Testing interprets **VAR** statements as **ARRAY** or **STRUCTURE** statements.

Array and structure contents are fully tested, only the output records are more concise.

Note This setting slightly improves speed of execution because checks are performed on each array as a whole.

Initial and Expected Values

The Initial and Expected Value settings are part of the Component Testing Settings for C dialog box and describes how values assigned to each variable are displayed in the Component Testing report. Component Testing allows three possible evaluation strategy settings.

Variable Only

This evaluation strategy setting generates both the initial and expected values of each variable evaluated by the program during execution.

This is possible only for variables whose expression of initial or expected value is not reducible by the Test Compiler. For arrays and structures in which one of the members is an array, this evaluation is not given for the initial values. For the expected values, however, it is given only for *Failed* items.

Value Only

With this setting, the test report displays for each variable both the initial value and the expected value defined in the test script.

Combined evaluation

The combined evaluation setting combines both settings. The test report thus displays the initial value, the expected value defined in the test script, and the value found during execution if that value differs from the expected value.

Test Script Compiler Macro Definitions

You can specify a list of conditions to be applied when starting the Test Script Compiler. You can then generate the test harness conditionally. In the test script, you can include blocks delimited with the keywords **IF**, **ELSE**, and **END IF**.

If one of the conditions specified in the **IF** instruction is present, the code between the keywords **IF** and **ELSE** (if **ELSE** is present), and **IF** and **END IF** (if **ELSE** is not present) is analyzed and generated. The **ELSE / END IF** block is eliminated.

If none of the conditions specified in the IF instruction is satisfied, the code between the keywords **ELSE** and **END IF** is analyzed and generated.

By default, no generation condition is specified, and the code between the keywords **ELSE** and **END IF** is analyzed and generated.

Pointers

Testing Pointers against Pointer Structure Elements

To test pointers against structure elements which are also pointers, specify for each pointer the variable it is pointing to.

For example, consider the following code:

```
typedef struct st_Test
{
    int a;
    int b;
    struct st_Test *Ptr1;
}st_Toto;
int FunctionTest (st_Toto *p_toto)
{
    int res=0;
    if (p_toto != 0)
    {
        if(p_toto->Ptr1 == 0)
        {
            res = 1;
        }
    }
    else
    {
        res = 2;
    }
    return(res);
}
```

To test the pointer p_toto, write the following test script:

```
SERVICE TestFunction
SERVICE_TYPE extern
-- Tested service parameter declarations
    #st_toto *p_toto;
-- By function returned type declaration
```



```

#int ret_TestFunction;
ENVIRONMENT ENV_TestFunction
VAR ret_TestFunction,  init = 0,  ev = init
END ENVIRONMENT -- ENV_TestFunction
USE ENV_TestFunction
TEST 1
FAMILY nominal
ELEMENT
    STR *p_toto,  init = { a => 0, b => 0, Ptr1 => NIL
}, ev= init
    STR *p_toto->Ptr1, init = {a=>2,b=>32, Ptr1=>NIL},
ev= init
    VAR ret_TestFunction,  init = 0, ev = init
    #ret_TestFunction = TestFunction(p_toto);
    END ELEMENT
END TEST -- TEST 1
FIN SERVICE -- TestFunction

```

Pointer and Array Ambiguities

Use the **string_ptr** keyword on a **VAR** line to work around the ambiguity of the C language between arrays and pointers.

For example the following **VAR** line (supposing the declaration *char* string*;) will generate C code that will copy the string into the memory location pointed by string.

```

VAR string,  init = "foo", ev = init
-- This is the "traditional" way

```

Of course, if no memory was allocated to the variable, this is not possible.

The following alternative approach causes the string to point to the memory location containing **foo**. The string is then compared to **foo** using a string comparison function:

```

VAR string, string_ptr, init = "foo", ev = init
-- Note the additional field in the line

```

This syntax allows you to initialize the variable to **NIL**, and to compare its contents to a given string after the test.

Testing an Array Whose Elements are Unions

When testing an array of unions, detail your tests for each member of the array, using **VAR** lines in the **ELEMENT** block.

Example

Considering the following variables:

```
#typedef struct {
# int test1;
# int test2;
# int test3;
# int test4;
# int test5;
# int test6;
# } Test;
#typedef struct {
# int champ1;
# int champ2;
# int champ3;
# } Champ;
#typedef struct {
# int toto1;
# int toto2;
# } Toto;
#typedef union {
# Test A;
# Champ B;
# Toto C;
# } T_union;
#extern T_union Tableau[4];
```

The test must be written element per element:

```
TEST 1
FAMILY nominal
ELEMENT
  VAR Tableau[0], init = {A => { test1 => 0, test2 => 0,
test3 => 0, test4 => 0,
&                                test5 => 0, test6 =>
0} }, ev = init
  VAR Tableau[1], init = {B => { champ1 => 0, champ2 => 0,
champ3 => 0} }, ev = init
  VAR Tableau[2], init = {B => { champ1 => 0, champ2 => 0,
champ3 => 0} }, ev = init
  VAR Tableau[3], init = {B => { champ1 => 0, champ2 => 0,
champ3 => 0} }, ev = init
  #ret_fct;
END ELEMENT
```

```
END TEST -- TEST 1
```

Initializing Pointer Variables while Preserving the Pointed Value

To initialize a variable as a pointer while keeping the ability to test the value of the pointed element, use the **FORMAT string_ptr** statement in your **.ptu** test script.

This allows you to initialize your variable as a pointer and still perform string comparisons using **str_comp**.

Example:

```
TEST 1

FAMILY nominal
ELEMENT
  FORMAT pointer_name = string_ptr
-- Then your variable pointer_name will be first initialized
as a pointer
  ....
  VAR pointer_name, INIT="l11c01pA00", ev=init
-- It is initialized as pointing at the string "l11c01pA00",
--and then string comparisons are done with the expected
values using str_comp.
```

Functions

Testing Main Functions

You can use the Component Testing feature to test C language *main* functions. To do so, you must rename those functions.

Example

```
#ifdef ATTOL
int test_main (int argc, char** argv)
#else
int main (int argc, char** argv)
#endif
{
  ...
}
```

If you are running an runtime analysis feature on the Component Testing test node, you can also use the **-rename** command line option to rename the *main* function name.

See the Instrumentor Line Command Reference section in the **Rational Test RealTime Reference Manual**.

Functions Using a Variable Number of Parameters

Some functions accept a variable number of parameters on each call.

You can still stub these functions with the Component Testing feature by using the *'...'* syntax indicating that there may be additional parameters of unknown type and name.

In this case, Component Testing can only test the validity of the first parameter.

Example

The standard *printf* function is a good a example of a function that can take a variable number of parameters:

```
int printf (const char* param, ...);
```

Here is an example including a STUB of the *printf* function:

```
HEADER add, 1, 1
#extern int add(int a, int b);
#include <stdio.h>
BEGIN
DEFINE STUB MulitParam
#int printf (const char param[200], ...);
END DEFINE
SERVICE add
  #int a, b, c;
  TEST 1
  FAMILY nominal
    ELEMENT
      VAR a, init = 1, ev = init
      VAR b, init = 3, ev = init
      VAR c, init = 0, ev = 4
```

```

        STUB printf("hello %s\n")12
        #c = add(a,b);
    END ELEMENT
END TEST
END SERVICE

```

Functions Taking void* Parameters

When stubbing a function that takes void* type parameters, the Source Code Parser generates incomplete code that might not compile.

When you are stubbing functions that take void* parameters, you must check and edit the **.ptu** test script accordingly.

Example

Consider the following test script generated by the C Source Code Parser:

```

DEFINE STUB fct_sim_c
#int fct_sim(double_in c, void_inout d);
END DEFINE
You must modify the .ptu script like this:
DEFINE STUB fct_sim_c
#int fct_sim(double_in c, unsigned char_inout d);
END DEFINE
Or, if testing the parameters is not required:
DEFINE STUB fct_sim_c
#int fct_sim(double_no c, unsigned char_no d);
END DEFINE

```

Functions Using const Parameters

Functions using *const* parameters sometimes produce compilation errors when stubbed with Test RealTime.

This is because the preprocessor generates variables that are used for testing calls to the **STUBs**. These variables have the same type as the parameter to the function being stubbed: *const int*. These *const* variables cannot be modified, causing the compilation errors.

To work around this problem, you can to indicate that type modifiers for a STUB parameter should be used in the function definition, but not in the

declaration of the variables used to control the STUBs.

To do this, add an `@` character as a prefix to the the type modifier. If your function takes a *const* pointer, then you don't need the `@` prefix:

This technique can be used with any type modifier.

Example

Consider the following function:

```
extern int ConstParam(const int param);
```

To stub the function, you would normally write the following lines in the **.ptu** test script. These will produce compilation error messages:

```
DEFINE STUB Example
#int ConstParam(const int _in param);
END DEFINE
```

Instead, use the following syntax to define the stub:

```
DEFINE STUB Example
#int ConstParam(@const int _in param);
END DEFINE
```

If your function takes a *const* pointer:

```
DEFINE STUB Example
#int ConstParam(const int _in *param);
END DEFINE
```

Functions Containing Type Modifiers

Type modifiers can appear in the signature of the function but should not be used when manipulating any passed variables. When using type modifiers, add `@` prefix to the type modifier keyword.

Test RealTime recognizes `@`-prefixed type modifiers in the function prototype, but ignores them when dealing internally with the parameters passed to and from the function.

This behaviour is the default behaviour for the "const" keyword, the '@' is

not necessary for const.

Example

Consider a type modifier **__foo**

```
DEFINE STUB tst_cst
#int ModifParam(@__foo float _in param);
END DEFINE
```

Note In this example, **__foo** is not a standard ANSI-C feature. To force Test RealTime to recognize this keyword as a type modifier, you must add the following line to the **.ptu** test script:

```
##pragma attol type_modifier = __foo
```

Functions Using *_inout* Mode Arrays

To stub a function taking an array in **_inout** mode, you must provide storage space for the actual parameters of the function.

The function prototype in the **.ptu** test script remains as usual:

```
#extern void function(unsigned char *table);
```

The **DEFINE STUB** statement however is slightly modified:

```
DEFINE STUB Funct
#void function(unsigned char _inout table[10]);
END DEFINE
```

The declaration of the pointer as an array with explicit size is necessary to memorize the actual parameters when calling the stubbed function. For each call you must specify the exact number of required array elements.

```
ELEMENT
STUB Funct.function 1 => (({'a', 'b', 'c', 'd', 'e', 'f',
'g', 'h', 'i', 0x0},
& {'i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a', 0x0}))
#call_the_code_under_test();
END ELEMENT
```

This naming convention compares the actual values and not the pointers.

The following line shows how to pass **_inout** parameters:

```
{<in_parameter>}, {<out_parameter>}}
```

Functions Taking *char** Parameters

You can use Component Testing to stub functions that take a parameter of the *char** type.

The *char** type causes problems with the Component Testing feature because of the ambiguity built into the C programming language. The *char** type can represent:

- Pointers
- Pointers to a single *char*
- Arrays of characters of indeterminate size
- Arrays of characters of which the last character is the character `\0`, a C string.

By default, Component Testing treats all variables of this type as C strings. If this is not the desired behavior, then you must inform Component Testing with the use of other instructions. To force to another representation use one of the following methods.

Pointers

Define the stub as in the following example:

```
#int StubFunction(char* pChar);  
DEFINE STUB Stubs  
#int StubFunction(void* _in pChar)  
END DEFINE  
#char MyChar = 'A';  
STUB StubFunction(NIL) 0, (&MyChar) 1
```

Pointers to a Single *char*

Define the type as ***_inout***, as in the following example. The *** isn't necessary as the type must be a pointer to be an out parameter:


```
#int StubFunction(char* pChar);
DEFINE STUB Stubs
#int StubFunction(char _inout pChar)
END DEFINE
STUB StubFunction(('a','A'))0
```

Arrays of Characters of Indeterminate Size

Use the **FORMAT** statement to force the treatment as a pointer to an unsigned char and use the **TAB** instruction to test the variable as a table.

For example:

```
#int StubFunction(char* pChar);
DEFINE STUB Stubs
#int StubFunction(unsigned char _in Chars[4])
END DEFINE
STUB StubFunction({'a','b','c','d'})0, ({'A','B','C','D'})1
```

C strings

This is the default behavior:

```
#int StubFunction(char* pString);
DEFINE STUB Stubs
#int StubFunction(char* _in pString )
END DEFINE
STUB StubFunction("abcd")0, ("ABCD")1
```

C and Ada Test Script

Ada

Ada Records with Discriminants

You can use record types with discriminants, with the following ADA restrictions:

- Initialization part must be complete.
- Evaluation can omit every field except discriminant fields.

Initialization and expected value expressions are ADA aggregates

beginning with the value of the discriminant.

Example

```
type rec (discr:boolean:=TRUE)
  case discr is
  when TRUE =>
    ch2:float;
  when FALSE =>
    ch3:integer;
  end case;
end record;
#r1: rec(TRUE);
#r2: rec;
TEST 1
FAMILY nominal
ELEMENT
var r1, init = (TRUE, 0.0), ev ==
var r2, init = (FALSE, 1), ev = (TRUE, 1.0)
#func (r);
END ELEMENT
END TEST
```

Separate Compilation

You can make internal procedures and variables and the structure of private types visible from the test program, by including them in the body of the unit under test with a separate ADA instruction.

You must add the following line at the end of the body of the unit tested:

```
PACKAGE BODY <name>
...
PROCEDURE Test is separate;
END;
```

Defining the procedure **Test** this way allows you to access every element of the specification and also those defined in the body.

Generic Units

Types and objects in a generic unit depend on generic formal parameters that are not known by the Test Script Compiler. Therefore, you cannot test generic units.

On the other hand, you can test instances of generic units. Such instances must appear in compilation units or at the beginning of the test script, as follows:

```
WITH <generic>;  
PACKAGE <instance> IS NEW <generic> (...);
```

Unknown Values

In some cases, Component Testing for Ada is unable to produce a default value in the `.ptu` test script. When this occurs, Component Testing produces an invalid value with the prefix `_Unknown`.

Such cases include:

- Private values: `_Unknown_private_value`
- Function pointers: `_Unknown_access_to_function`
- Tagged limited private: `_Unknown_access_to_tagged_limited_private`

Before compiling you must manually replace these `_Unknown` values with valid values.

Test Program Entry Point

Since `ATTOL_TEST` is a sub-unit and not a main unit, Component Testing for Ada generates a main procedure at the end of the test program with the name provided on the command line.

Two methods are available to start the execution of the test program:

- Call during the elaboration of the unit under test.
- Call by the main procedure.

Call During the Elaboration of the Unit

In this case, you must add an additional line in the body of the unit tested:

```
PACKAGE <name>
```

```

...
END;
PACKAGE BODY <name>
...
  PROCEDURE ATTOL_TEST is SEPARATE;
BEGIN
...
  ATTOL_TEST;
END;

```

The package specification is not modified, but the test procedure is called at every elaboration of the package. Therefore, you need to remove or replace this call with an empty procedure after the test phase.

Call by the Main Procedure

In this case, you must add an additional line in the specification of the unit tested:

```

PACKAGE < name>
...
  PROCEDURE ATTOL_TEST;
...
END;
PACKAGE BODY <name> is
...
  PROCEDURE ATTOL_TEST is SEPARATE;
END;

```

Component Testing will then automatically generate a call to the **ATTOL_TEST** procedure in the main procedure of the test program. The test will be executed during the execution of the main program.

Limitations

Consider the following limitations:

- The unit under test must be of type *package*.
- The root body of **ATTOL_TEST** (procedure **ATTOL_TEST** is separate) cannot appear inside a generic package.

Testing Generic Packages

To test a generic package, you must instantiate it and then call the instance. This however causes problems in cases where the test driver procedure must be generated separately from the tested package.

To avoid this, you need to generate the test driver separately and call it as a procedure of the instance.

In the **.ptu** test script, replace the **BEGIN** line with the following statement:

```
BEGIN GENERIC(<Generic_Package>, <Instance>),  
             <Procedure_Name>
```

where *<Generic_Package>* is the name of the generic unit under test, and *<Instance>* is the name of the instantiated unit from the generic. The *<Procedure_Name>* parameter is not mandatory. Component Testing uses **Attol_Test** by default.

This Syntax automatically generates a separate procedure *<Procedure_Name>* of *<Generic_Package>* and then calls the procedure *<Instance>.<Procedure_Name>*, which is code generated by the Component Testing Preprocessor).

Note This technique also allows testing of private types within the generic package.

Declaring Global Variables for Testing

The Target Deployment Ports for Ada do not provide any variables which might be used freely by the tester.

To avoid having to modify the code under test, it is easier to add an extra Ada package, which is actually just the *spec* part of the package, to provide a set of globally accessible variables. You can do this directly in the **.ptu** test script.

Declaring Global Variables

Any code inserted between the **HEADER** and **BEGIN** keywords is copied into the generated code as is. For example:

```
Header Code_Under_Test, 1.0, 1.0
  #With Code_Under_Test; -- only if Code_Under_Test is
used within My_Globals
  -- this context clause goes into the package My_Globals
  #package My_Globals is
  #   Global_Var_Integer : Integer := 0;
  #end My_Globals;
  #with Code_Under_Test;
  #with My_Globals;
  -- these two context clauses go into the generated test
harness
Begin
-- etc..
```

Note Any Ada instruction between **HEADER** and the **BEGIN** instruction must be encapsulated into a procedure or a package. Context clauses are possible.

Accessing Global Variables

The extra global variable package is visible from within all units of the test driver.

Variables can be accessed like this:

```
#My_Globals.Global_Var_Integer := 1;
```

Variables can be accessed from a **DEFINE STUB** block for example:

```
Define Stub Another_Package
#with My_Globals;
#procedure some_proc (param : in out some_type) is
#begin
#   My_Globals.Global_Var_Integer := 2;
#end some_proc;
-- however, no "return" statement is possible within this
block
End Define
```

Variables can be accessed in the **ELEMENT** blocks, just like any other

variable:

```
VAR My_Globals.Global_Var_Integer, init = 0, EV = 1
```

Rational Test RealTime processes the **.ptu** test script in such a way that global variable package automatically becomes a separate compilable unit.

Generating a Separate Test Harness

Because of restrictions of the Ada language, Component Testing cannot generate a test harness which is a separate of more than one package.

You can however generate the main test harness as a separate of one of the packages and declare additional procedures as separates of other packages. This is done in the header of the **.ptu** test script, as in the following example:

```
Header Code_Under_Test, 1.0, 1.0
  #separate (Second_Package);
  #procedure Something is
  #begin
  # -- here internal variables of Second_Package are
  # -- visible; private types can be accessed etc.
  # null;
  #end Something;
  #with Second_Package;
  -- this is to gain visibility on the package
  -- from within the test harness
Begin First_Package, Test_Entry_Point
  -- this causes Test RealTime to generate a procedure
  -- "Test_Entry_Point" as a separate of "First_Package" as
  -- "main" procedure of the Test Harness
  -- etc.
```

If the test script requires access to items from *Second_Package*, it can call the corresponding procedure from within an ELEMENT block of this **.ptu** test script.

```
Element
  -- some VAR instructions here
  #Second_Package.Something;
  #-- here is the call to the tested procedure
End Element
```

Test Script Modification

The following modified **BEGIN** instruction tells Component Testing to generate a separate test procedure:

```
BEGIN <tested_unit> [, <separate_procedure>]
```

where:

- <tested_unit> is the name of the package under test (modified as shown in the preceding paragraph).
- <separate_procedure> is the name of the separate procedure to be generated. This parameter is optional. By default, Component Testing uses **ATTOL_TEST**.

The generation will then occur as follows:

```
SEPARATE (<unit_tested>)  
PROCEDURE <separate_procedure> is  
    . . .  
END ATTOL_TEST;
```

When the unit tested is a sub-unit, you must include the parent unit in ADA in its name.

The separate procedure will now be referred to as **ATTOL_TEST**.

Testing Ada Tasks

As a general matter, Test RealTime Component Testing for Ada was designed for synchronous programming. However, it is possible to achieve component testing even in an asynchronous environment.

The important detail is that any task which might be producing Runtime Analysis information (especially by calling stubbed procedures or functions) must be terminated when control reaches the **END ELEMENT** instruction in the **.ptu** test script.

If the code under test does not provide select statements or entry points in

order to request the termination of the task, an abort call to the task might be necessary. For tasks who terminate after a certain time (not entering a infinite loop), the tester might check the task's state and sleep until termination of the task. In the **.ptu** test script, this might read as follows:

```
#while not TaskX'Terminated loop
#   delay 1;
#end loop;
```

This instruction block is placed just before the **END ELEMENT** statement of the Test Script.

Example

The source files and complete **.ptu** script for following example are provided in the **examples/Ada_Task** directory.

In this example, the task calls a stubbed procedure. Therefore the task must be terminated from within the Test Script. Two different techniques of starting and stopping the task are shown here in **Test 1** and **Test 2**.

```
HEADER prg_ss_tst, 0.3, 0.0
#with Pck_Muet;
BEGIN Prg_Ss_Tst
DEFINE STUB Pck_Muet
#with Text_IO;
#procedure Proc_Bouchonnee is
#begin
# Text_IO.Put_Line("Stub called.");
#end;
END DEFINE
SERVICE UNE
SERVICE_TYPE extern
#Param_Un_ : duration;
#task1 : Prioritaire;
TEST 1
FAMILY nominal
ELEMENT
VAR Param_Un_, init = duration(0), ev = init
STUB Pck_Muet.Proc_Bouchonnee 1..1 => ()
#Task1.Unit_Testing_Exit_Loop;
#delay duration(5);
#Task1.Unit_Testing_Wait_Termination;
END ELEMENT
```

```

END TEST -- TEST 1
TEST 2
FAMILY nominal
ELEMENT
    VAR Param_One, init = duration(2), ev = init
STUB Pck_Muet.Proc_Bouchonnee 1..1 => ()
#declare
    # Task2 : T_Prio := new Prioritaire;
#begin
# Task2.Do_Something_Useful(Param_One);
# Task2.Unit_Testing_Exit_Loop;
# Task2.Unit_Testing_Wait_Termination;
#end;
    END ELEMENT
END TEST -- TEST 2
END SERVICE --UNE

```

In the **BEGIN** line of the script, it is not necessary to add the name of the separate procedure **Attol_Test**, as this is the default name;

The user code within the **STUB** contains a context clause and some custom native Ada instructions.

In both **Test 1** and **Test 2** it is necessary not only to stop the main loop of the task before reaching the **END ELEMENT** instruction, but also the task itself in order to have the tester return.

Task1 and **Task2** could run in parallel, however, the test Report would be unable to distinguish between the **STUB** calls coming in from either task, and would show the calls in a cumulative manner.

The entry points **Unit_Testing_Exit_Loop** and **Unit_Testing_Wait_Termination** can be considered as implementations for testing purposes only. They might not be used in the deployment phase.

The second test is *False* in the Report, the loop runs twice. This allows to check that the dump goes through smoothly.

Environments

About Environments

When drawing up a test script for a service, you usually need to write several test cases. It is likely that, except for a few variables, these scenarios will be very similar. You could avoid writing a whole series of similar scenarios by factorizing items that are common to all scenarios.

Furthermore, when a test harness is generated, there are often side-effects from one test to another, particularly as a result of unchecked modification of global variables.

To avoid these two problems and leverage your test script writing, the Test Script Language lets you define test environments introduced by the keyword **ENVIRONMENT**.

These test environments are effectively a set of default tests performed on one or more variables.

Declaring Environments

A test environment consists of a list of variables for which you specify:

- Default initialization conditions for before the test
- Default expected results for after the test

Use the **VAR**, **ARRAY**, and **STR** instructions described previously to specify the status of the variables before and after the test.

You can only use an environment once you have defined it.

Delimit an environment using the instructions **ENVIRONMENT** <environment_name> and **END ENVIRONMENT**. You must place it after the **BEGIN** instruction. When you have declared it, an environment is visible to the block in which it was declared and to all blocks included therein.

Example

The following example illustrates the use of environments:

```
HEADER histo, 1, 1
#include <math.h>
#include "histo.h"
BEGIN
ENVIRONMENT image
  ARRAY image, init = 0, ev = init
END ENVIRONMENT
USE image
SERVICE COMPUTE_HISTO
  #int x1, x2, y1, y2;
  #int status;
  #T_HISTO histo;
  #T_IMAGE image1;
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
```

Environment Override

To provide more flexibility in using environments, you can override the initialization and test specifications in an **ENVIRONMENT** block for one or more variables, one or more array elements, or one or more fields of a structured variable by using either of the following:

- A new environment
- The instructions **VAR**, **ARRAY**, or **STR** in the **ELEMENT** block

The **ENVIRONMENT** concept greatly improves test robustness. You can use this approach to group default initialization and test specifications with all the variables that are global to a module under test, allowing you to check that unexpected global variables in tests on a service are indeed not modified.

The following steps are used to handle environments:

- **VAR**, **ARRAY** and **STR** instructions are stored between **ENVIRONMENT** and **END ENVIRONMENT** instructions.
- When the Test Compiler comes across the instruction **USE**, it determines the scope of the environment that has been stored.
- At every **END ELEMENT** instruction, the Test Compiler browses through all visible environments beginning with the most recently declared one. The test compiler then checks every environment variable to see if it has been fully or partially tested. If it has only been partially tested, the test compiler generates the necessary tests to complete the testing of the variable.

This process means that:

- Tests linked to environments are always carried out last.
- The higher the environment's precedence, the earlier the tests it contains will be carried out.

Example

The following example illustrates an override of an array element in two tests:

```
TEST 1
FAMILY nominal
ELEMENT
VAR histo[0], init = 0, ev = SIZE_IMAGE*SIZE_IMAGE
#status = compute_histo(x1,y1,x2,y2, histo);
END ELEMENT
END TEST
TEST 2
FAMILY nominal
ELEMENT
ARRAY image, init = {others => {others => 100}}, ev = init
ARRAY histo[100], init = 0, ev = SIZE_IMAGE*SIZE_IMAGE
#status = compute_histo(x1,y1,x2,y2, histo);
END ELEMENT
END TEST
```

In the first test, only *histo[0]* has an override. Therefore, all the default tests were generated except for the test on the *histo* variable, which had its 0 element removed, and a test was generated on *histo[1..255]*.

In the second test, the override is more noticeable; the *histo[100]* element has been removed to generate two tests: one on *histo[0..99]*, and the other on *histo[101..255]*.

Specifying Parameters for Environments

You can specify parameters for environments.

Declare the parameters in the **ENVIRONMENT** instruction as you would for a service:

```
ENVIRONMENT compute_histo1(a,b,c,d)
  VAR x1, init = a, ev = init
  VAR x2, init = b, ev = init
  VAR y1, init = c, ev = init
  VAR y2, init = d, ev = init
  ARRAY histo[0..SIZE_HISTO?1], init = 0, ev = 0
  VAR status, init ==, ev = 0
END ENVIRONMENT
```

The parameters are identifiers, which you can use in variable status instructions, as follows:

- In initial or expected value expressions
- In expressions delimiting bounds of arrays in extended mode

The parameters are initialized when they are used:

```
USE compute_histo1(0,0,SIZE_IMAGE?1,SIZE_IMAGE?1)
```

The number of values must be strictly equal to the number of parameters defined for the environment. The values can be expressions of any type.

Using Environments

The **USE** keyword declares the use of an environment (in other words, the

beginning of that environment's visibility).

The impact or visibility of an environment is determined by the position at which you declare the environment's use with the **USE** statement.

The initial values and tests associated with the environment are applied as follows, depending on the position of the declaration:

- To all the tests in a program
- To all the tests in a service
- To all the **ELEMENT** blocks of a particular test
- Within one **ELEMENT** block of a given test.

Exceptions

Unexpected Exceptions

The generated test driver detects all raised exceptions. If a raised exception is not specified in the test script, it is displayed in the report.

When the exception is a standard ADA exception (**CONSTRAINT_ERROR**, **NUMERIC_ERROR**, **PROGRAM_ERROR**, **STORAGE_ERROR**, **TASKING_ERROR**), the exception name is displayed in the test report.

Overview

Declaring Parameters

ELEMENT blocks contain specific instructions that describe the test start-up procedures and the post-execution tests.

The hash character (#) at the beginning of a line indicates a native language statement written in C, C++, or ADA.

This declaration is introduced after the **SERVICE** instruction because it is local to the **SERVICE** block; it is invalid outside this block.

It is only necessary to declare parameters of the procedure under test. Global variables are already present in the module under test or in any integrated modules, and do not need to be declared locally.

Test Script Structure

The C++ Test Script Language allows you to structure tests to:

- Describe several test cases in a single test script,
- Select a subset of test cases according to different Target Deployment Port criteria.

A typical Component Testing **.ptu** test script looks like this:

```
HEADER add, 1, 1
BEGIN
SERVICE add
TEST 1
  FAMILY nominal
  ELEMENT
  END ELEMENT
END TEST
END SERVICE
```

All instructions in a test script have the following characteristics:

- All statements begin with a keyword.
- Statements are not case sensitive (except when C and C++ expressions are used).
- Statements start at the beginning of a line and end at the end of a line. You can, however, write an instruction over several lines using the ampersand (&) continuation character at the beginning of additional lines.
- Statements must be shorter than 2000 characters.

Structure Statements

The following statements allow you to describe the structure of a test.

HEADER

For documentation purposes, specifies the name and version number of the module being tested, as well as the version number of the tested source file. This information is displayed in the test report.

BEGIN

Marks the beginning of the generation of the actual test program.

SERVICE

Contains the test cases relating to a given service. A service usually refers to a procedure or function. Each service has a unique name (in this case add). A **SERVICE** block terminates with the instruction **END SERVICE**.

TEST

Each test case has a number or identifier that is unique within the block **SERVICE**.

The test case is terminated by the instruction **END TEST**.

You can execute the test case several times by adding the number of iterations at the end of instruction **TEST**, for example:

```
TEST <name> LOOP <number>
```

You can add other test cases to the current test case by using the instruction **NEXT_TEST**:

```
TEST <name>  
...  
NEXT_TEST  
...
```

END TEST

This instruction allows a new test case to be added that will be linked to the preceding test case. Each loop introduced by the instruction **LOOP** relates to the test case to which it is attached.

FAMILY

Qualifies the test case to which it is attached. The qualification is free (here nominal). A list of qualifications can be given (for example: family, nominal, structure).

ELEMENT

Describes a test phase in the current test case. The phase is terminated by the instruction **END ELEMENT**.

The different phases of the same test case cannot be dissociated after the tests are run, unlike the test cases introduced by the instruction **NEXT_TEST**. However, the test phases introduced by the instruction **ELEMENT** are included in the loops created by the instruction **LOOP**.

The three-level structure of the test scripts has been deliberately kept simple. This structure allows:

- A clear and structured presentation of the test script and report
- Tests to be run selectively on the basis of the service name, the test number, or the test family.

Simulations

C and Ada Syntax Extensions

For a large number of calls to a stub, use the following syntax for a more compact description:

```
<call i> .. <call j> =>
```

You can describe each call to a stub by adding the specific cases before the preceding instruction, for example:

```
<call i> =>
```

or

```
<call i> .. <call j> =>
```

The call count starts at 1 as the following example shows:

```
TEST 2
FAMILY nominal
COMMENT Reading of 100 identical lines
ELEMENT
VAR file1, init = "file1", ev = init
VAR file2, init = "file2", ev = init
VAR s, init == , ev = 1
STUB open_file 1=>("file1")3
STUB create_file 1=>("file2")4
STUB read_file 1..100(3,"line")1, 101=>(3,"")0
STUB write_file 1..100=>(4,"line")1
STUB close_file 1=>(3)1,2=>(4)1
#s = copy_file(file1,file2);
END ELEMENT
END TEST
```

Several Calls to a Stub

If a stub is called several times during a test, either of the following are possible:

- Describe the different calls in the same **STUB** instruction, as described previously.
- Use several **STUB** instructions to describe the different calls. (This allows a better understanding of the test script when the **STUB** calls are not consecutive.)

The following example rewrites the test to use this syntax for the call to the **STUB close_file**:

```
STUB close_file (3)1
STUB close_file (4)1
```

No Testing of the Number of Calls of a Stub

If you don't want to test the number of calls to a stub, you can use the keyword `others` in place of the call number to describe the behavior of the stub for the calls to the stub not yet described.

For example, the following instruction lets you specify the first call and all the following calls without knowing the exact number:

```
STUB write_file 1=>(4,"line")1,others=>(4,"")1
```

Creating Complex Stubs

If necessary, you can make stub operation more complex by inserting native code into the body of the simulated function. You can do this easily by adding the lines of native code after the prototype, as shown in the following example:

```
DEFINE STUB file
#int fic_errno;
#char fic_err_msg[100];
#int open_file(char _in f[100])
# { errno = fic_errno; }
#int create_file(char _in f[100])
# { errno = fic_errno; }
#int read_file(int _in fd, char _out l[100])
# { errno = fic_errno; }
#int write_file(int fd, char _in l[100])
# { errno = fic_errno; }
#int close_file(int fd)
# { errno = fic_errno; }
END DEFINE
```

Excluding a Parameter from a Stub

Stub Definition

You can specify in the stub definition that a particular parameter is not to be tested or given a value. You do this using a modifier of type `_no` instead of `_in`, `_out` or `_inout`, as shown in the following example:

```
DEFINE STUB file
#int open_file(char _in f[100]);
```

```

    #int create_file(char _in f[100]);
    #int read_file(int _no fd, char _out l[100]);
    #int write_file(int _no fd, char _in l[100]);
    #int close_file(int fd);
END DEFINE

```

In this example, the `fd` parameters to `read_file` and `write_file` are never tested.

Note You need to be careful when using `_no` on an output parameter, as no value will be assigned to it. It will then be difficult to predict the behavior of the function under test on returning from the stub.

Stub Usage

Parameters that have not been tested (preceded by `_no`) are completely ignored in the stub description. The two values of the input/output parameters are located between brackets as shown in the following example:

```

DEFINE STUB file
    #int open_file(char _in f[100]);
    #int create_file(char _in f[100]);
    #int read_file(int _no fd, char _inout l[100]);
    #int write_file(int _no fd, char _in l[100]);
    #int close_file(int _no fd);
END DEFINE
...
STUB open_file ("file1")3
STUB create_file ("file2")4
STUB read_file (("","line 1"))1, (("line 1","line 2"))1,
& (("line2",""))0
STUB write_file ("line 1")1, ("line 2")1
STUB close_file ()1, ()1

```

If a stub is called and if it has not been declared in a scenario, an error is raised in the report because the number of the calls of each stub is always checked.

Sizing Stubs

For each **STUB**, the Component Testing feature allocates memory to:

- Store the value of the input parameters during the test
- Store the values assigned to output parameters before the test

A stub can be called several times during the execution of a test. By default, when you define a **STUB**, the Component Testing feature allocates space for 10 calls. If you call the **STUB** more than this you must specify the number of expected calls in the **STUB** declaration statement.

In the following example, the script allocates storage space for the first 17 calls to the stub:

```
DEFINE STUB file 17
  #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
```

Note You can also reduce the size when running tests on a target platform that is short on memory resources.

Simulation of Generic Units

You can stub a generic unit like an ordinary unit with the following restrictions:

Parameters of a procedure or function, and function return types of a type declared in a generic unit or parameter of this unit must use the **_NO** mode.

For example, if you want to stub the following generic package:

```
GENERIC
  TYPE TYPE_PARAM is .....
```

Package GEN is

```
  TYPE TYPE_INT0 is ....;
  procedure PROC(x:TYPE_PARAM,y:in out TYPE_INT0,Z:out
integer);
  function FUNC return TYPE_INT0;
end GEN;
```

Use the following stub definition:

```

DEFINE STUB GEN
#  procedure PROC(x: _NO TYPE_PARAM,y: _NO TYPE_INT0,Z:out
integer);
#  function FUNC return _NO TYPE_INT0;
END DEFINE

```

You can add a body to procedures and functions to process any parameters that required the **_NO** mode.

Note With some compilers, when stubbing a unit by using a **WITH** operator on the generic package, cross dependencies may occur.

Separate Body Stub

In some cases, you might need to define the body stub separately, with a proprietary behavior. Declare the stub separately as shown in the following example, and then you can define a body for it:

```

DEFINE STUB <STUB NAME>
# procedure My_Procedure(...) is separate ;
END DEFINE

```

The Ada Test Script Compiler will not generate a body for the service *My_Procedure*, but will expect you to do so.

Stub Definition in C

The following simulation describes a set of function prototypes to be simulated in an instruction block called **DEFINE STUB ... END DEFINE**:

```

HEADER file, 1, 1
BEGIN
DEFINE STUB file
#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

```

The prototype of each simulated function is described in ANSI form. The following information is given for each parameter:

- The type of the calling function (**char f[100]** for example, meaning that the calling function supplies a character string as a parameter to the `open_file` function)
- The method of passing the parameter, which can take the following values:
 - **_in** for an input parameter
 - **_out** for an output parameter
 - **_inout** for an input/output parameter

These values describe how the parameter is used by the called function, and, therefore, the nature of the test to be run in the stub.

- The **_in** parameters only will be tested.
- The **_out** parameters will not be tested but will be given values by a new expression in the stub.
- The **_inout** parameters will be tested and then given values by a new expression.

Any returned parameters are always taken to be **_out** parameters.

The following example (for ADA) highlights the simulation of all functions and procedures declared in the specification of **file_io**. A new body is generated for `file_io` in file `<testname>_fct_simule.ada`.

```

HEADER file, 1, 1
BEGIN
DEFINE STUB file_io
END DEFINE

```

You must always define stubs after the `BEGIN` instruction and outside any **SERVICE** block.

Modifying Stub Variable Values

You can define stubs so that the variable pointed to is updated with

different values in each test case. For example, to stub the following function:

```
extern void function_b(unsigned char * param_1);
```

Declare the stub as follows:

```
DEFINE STUB code_c
    #void function_b(unsigned char _out param_1);
END DEFINE
```

Note Any **_out** parameter is automatically a pointer, therefore the asterisk is not necessary.

To return '255' in the first test case and 'a' in the second test case, you would write the following in your test script:

```
SERVICE function_a
SERVICE_TYPE extern
    -- By function returned type declaration
    #int ret_function_a;
TEST 1
    FAMILY nominal
        ELEMENT
            VAR ret_function_a, init = 0, ev = 1
            STUB function_b (255)
            #ret_function_a = function_a();
        END ELEMENT
    END TEST -- TEST 1
TEST 2
    FAMILY nominal
        ELEMENT
            VAR ret_function_a, init = 1, ev = 0
            STUB function_b ('a')
            #ret_function_a = function_a();
        END ELEMENT
    END TEST -- TEST 2
END SERVICE -- function_a
```

Simulating Global Variables

The simulated file can also contain global variables that are used by the functions under test. In this case, as with simulated functions, you can simulate the global variables by declaring them in the **DEFINE STUB** block, as shown in the following example:

```

DEFINE STUB file
  #int fic_errno;
  #char fic_err_msg[100];
  #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

```

The global variables are created as if they existed in the simulated file.

Stub Simulation Overview

Stub simulation is based on the idea that functions to be simulated are replaced with other functions generated in the test driver. These generated functions have the same interface as the simulated functions, but the body of the functions is replaced. The functions are called stubs.

The generated test harness can be represented as shown in the following figure:

These stubs have the following roles:

- Store input parameters to simulated functions
- Assign output parameters from simulated functions

To be able to generate these stubs, the Test Script Compiler needs to know the prototypes of the functions that are to be simulated and the method of passing each parameter (input, output, or input/output).

Passing parameters by pointer can lead to problems of ambiguity regarding the data actually passed to the function. For example, a parameter that is described in a prototype by `int *x` can be passed in the following way:

```

int *x as input ==> f(x)
int x as output or input/output ==> f(&x)
int x[10] as input ==> f(x)
int x[10] as output or input/output ==> f(x)

```

Therefore, to describe the stubs, you should specify the following:

- The data type in the calling function
- The method of passing the data

Stub Usage in Ada

Range of Values of STUB Parameters

When using stubs, you may need to define an authorized range for each **STUB** parameter. Furthermore, you can summarize several calls in one line associated with this parameter.

Write such **STUB** lines as follows:

```
STUB F 1..10 => (1<->5) 30
```

This expression means that the **STUB F** will be called 10 times with its parameter having a value between 1 and 5, and its return value is always 30.

You can combine this with several lines; the result looks like the following:

```
STUB F 1..10 => (1<->5) 30,
& 11..19 => (1<->5) 0,
& 20..30 => (<->) 1,
& others => (<->) -1
```

Raise-exception Stubs

You can force to raise a user-defined (or pre-defined) exception when a **STUB** is called with particular values.

The appropriate syntax is as follows:

```
STUB P(1E+307<->1E+308) RAISE STORAGE_ERROR
```

If the **STUB F** happens to be called with its parameter between **1E+307** and **1E+308**, the exception **STORAGE_ERROR** will be raised during execution of the application; the test will be **FALSE** otherwise.

Suppose that the current stubbed unit contains at least one overloaded sub-

program. When calling this particular **STUB**, you will need to qualify the procedure or function. You can do this easily by writing the **STUB** as follows:

```
STUB A.F (1<->2:REAL) RAISE STANDARD.CONSTRAINT_ERROR
```

The **STUB A.F** is called once and will raise a **CONSTRAINT_ERROR** if its parameter, of type **REAL**, has a value between **1** and **2**.

Compilation Sequence

The Ada Test Script Compiler generates three files:

- `<testname>_fct_simule.ada` for the body of simulated functions and procedures
- `<testname>_var_simule.ada` for the declaration of simulation variables
- `<testname>_var_simule_B.ada` for the body of test procedures

You must compile your packages in the following order:

1. Simulated unit (specification)
2. `<testname>_var_simule.ada`
3. `<testname>_var_simule_B.ada`
4. Test program
5. `<testname>_fct_simule.ada`

Replacing Stubs

Stubs can be used to replace a component that is still in development. Later in the development process, you might want to replaced a stubbed component with the actual source code.

To replace a stub with actual source code:

1. Right-click the test node and select Add Child and Files

2. Add the source code files that will replace the Stubbed functions.
3. If you do not want a new file to be instrumented, right-click the file select Properties. Set the Instrumentation property to No.
4. Open the .ptu test script, and remove the **STUB** sections from your script file.

Stub Usage in C

Stubs are used by means of the **STUB** instruction within environments or test scenarios.

This **STUB** instruction tests input parameters and assigns a value to output parameters each time the simulated function is called.

The following example illustrates how to use a file stub.

```
SERVICE copy_file
#char file1[100], file2[100];
#int s;
TEST 1
FAMILY nominal
ELEMENT
VAR file1, init = "file1", ev = init
VAR file2, init = "file2", ev = init
VAR s, init == , ev = 1
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 (3)1, (4)1
#s = copy_file(file1, file2);
END ELEMENT
END TEST
END SERVICE
```

The following information is required for every stub called in a scenario:

- Test values for the input parameters
- Return values for the output parameters
- Test and return values for the input/output parameters

- Where appropriate, the return value of the called stub

Replacing Stubs

Stubs can be used to replace a component that is still in development. Later in the development process, you might want to replaced a stubbed component with the actual source code.

To replace a stub with actual source code:

1. Right-click the test node and select Add Child and Files
2. Add the source code files that will replace the Stubbed functions.
3. If you do not want a new file to be instrumented, right-click the file select Properties. Set the Instrumentation property to No.
4. Open the .ptu test script, and remove the **STUB** sections from your script file.

Example

The following example specifies that you expect three calls of *foo*.

```
STUB STUB1.foo(1) 1, (2) 2, (3) 3
...
#foo(1);
#foo(2);
#foo(4);
```

The first call has a parameter of 1 and returns 1. The second has a parameter of 2 and returns 2 and the third has a parameter of 3 and returns 3. Anything that does not match is reported in the test report as a failure.

C and Ada Test Reports

Comparing Reports

The Component Testing comparison capability allows you to compare the results of the last two consecutive tests.

To activate the comparison mode, select **Display diff of last two test runs** in the Component Testing Settings for C dialog box.

In comparison mode an additional check is performed to identify possible regressions when compared with the previous test run.

The Component Testing Report displays an extra column named "Obtained Value Comparison" containing the actual difference between the current report and the previous report.

Understanding Component Testing Reports

Test reports for Component Testing are displayed in Test RealTime's Report Viewer.

The test report is a hierarchical summary report of the execution of a test node. Parts of the report that have *Passed* are displayed in green. *Failed* tests are shown in red.

Report Explorer

The Report Explorer displays each element of a test report with a *Passed* ✓ , *Failed* ✗ symbol.

Elements marked as *Failed* ✗ are either a failed test, or an element that contains at least one failed test.

Elements marked as *Passed* ✓ are either passed tests or elements that contain only passed tests.



Test results are displayed for each instance, following the structure of the **.ptu** test script.

Report Header

Each test report contains a report header with:

- The version of Test RealTime used to generate the test as well as the date of the test report generation
- The path and name of the project files used to generate the test
- The total number of test cases *Passed* and *Failed*. These statistics are calculated on the actual number of test elements listed in the sections below

Test Results

The graphical symbols in front of the node indicate if the test, item, or variable is *Passed*  or *Failed* .

- A test is *Failed* if it contains at least one failed variable. Otherwise, the test is considered *Passed*.

You can obtain the following data items if you click with the pointer on the Information node:

- Number of executed tests
- Number of correct tests
- Number of failed tests

A variable is incorrect if the expected value and the value obtained are not identical, or if the value obtained is not within the expected range.

If a variable belongs to an environment, an environment header is previously edited.

In the report variables are edited according to the value of the Display Variables setting of the Component Testing test node.

The following table summarizes the editing rules:

Results	Display Variable All Variables	Display Variable Incorrect Variables	Display Variable Failed Tests Only
✓ Passed test	Variable edited automatically	Variable not edited	Variable not edited
✗ Failed test	Variable edited automatically	Variable edited automatically	Variable edited if incorrect

The Initial and Expected Values option changes the way initial and expected values are displayed in the report.

Understanding Component Testing UML Sequence Diagrams for C and Ada

During the execution of the test, Component Testing generates trace data this is used by the UML/SD Viewer. The Component Testing sequence diagram uses standard UML notation to represent both Component Testing results.

When using Component Testing for C and Ada with Runtime Tracing or other Test RealTime features that generate UML sequence diagrams, all results are merged in the same sequence diagram.

You can click any element of the UML sequence diagram to open the test report at the corresponding line. Click again in the test report, and you will locate the line in the **.pts** test script.

Component Testing for C++

About Component Testing for C++

Component Testing for C++ is a fully integrated feature of Test RealTime

that uses object-oriented techniques to address automated testing of C++ embedded and native software.

Object-oriented testing does not mean that the Component Testing for C++ feature is designed solely for the testing object-oriented languages. Whether the target application is object-oriented or not, Component Testing for C++ adapts to the environment.

In fact, Component Testing for C++ can be used for:

- Software feature tests,
- Component integration tests,
- Software validation,
- Non-regression tests.

Overview

Basically, Component Testing for C++ interacts with your source code through a scripting language called *C++ Test Script Language*. You use the Test RealTime GUI or command line tools to set up your test campaign, write your test scripts, run your tests and view the test results. Object Testing's mode of operation is twofold:

- *C++ Test Driver* scripts describe a test harness that stimulates and checks basic I/O of the code under test.
- *C++ Contract Check* scripts, which instrument the code under test, verifying behavioral assertions during execution of the code.

When the test is executed, Component Testing for C++ compiles both the test scripts and the source under test, then instruments the source code and generates a test driver. Both the instrumented application and the test driver provide output data which is displayed within Test RealTime.

C++ Testing Overview

C++ Test Nodes

The project structure of the Rational Test RealTime GUI uses *test nodes* to represent your Component Testing test harness.

Test nodes created for Component Testing for C++ use the following structure

- **C++ Test Node:** represents the Component Testing for C++ test harness
- *<script>.otc*: is the Contract-Check test script
- *<script>.otd*: is the test driver script
- *<source>.cpp*: is the source file under test
- *<source>.cpp*: is an additional source file

C++ Contract-Check Script

The C++ Contract Check script allows you to test invariants and state charts as well as wraps for each method of the class.

The Contract Check script is contained in an **.otc** file, whose name matches the name of the file containing the class definition.

C++ Contract Check scripts are written in C++ Contract Check Language, which is part of the C++ Test Script language designed for Component Testing for C++.

Use the Component Testing wizard to set up a test node and create the C++ Contract Check script templates.

See the **Test RealTime Reference Manual** for the semantics of the C++ Contract Check Language.

C++ Test Driver Script

The C++ Test Driver Script stimulates the source code under test to test assertions on a cluster of classes.

The test driver script itself is contained in an **.otd** file and may call two optional files:

- A declaration file (**.dcl**) that contains C++ code that ensures the types, class, variables and functions needed by your test script will be available in your code.
- A stub file (**.stb**) whose purpose is to define variables, functions and methods which are to be stubbed.

Using a separate declaration and stub files is optional. It is possible to include all or certain declarations and stubs directly within the test driver script file.

C++ Contract Check scripts are written in C++ Contract Check Language, which is part of the C++ Test Script Language designed for Component Testing for C++.

Use the Component Testing wizard to set up a test node and create the C++ Test Driver script templates.

See the **Test RealTime Reference Manual** for the semantics of the C++ Contract Check Language.

Files and Classes Under Test

Source Files

The **Source under test** are source files containing the code you want to test. These files must contain either the definition of the classes targeted by the test, or method implementations of those classes.

Note Source files can be either body files (**.C**, **.cc**, **.cpp...**) or header files (**.h**), but it is usually recommended to select the body file. Specifying both header and body files as **Source under test** is unnecessary.

When using a C++ Test Driver Script, the wizard generates:

- A template test driver script (**.otd**) to test each class defined in the **Candidate classes** box.
- Declaration (**.dcl**) and stub (**.stb**) files to make the environment of the source under test available to the test script.

When using a C++ Contract Check script, the wizard generates:

- A template contract script (**.otc**) containing template code allowing you to add invariants and state charts as well as empty wraps for each method of the class.

Note If a source under test is a header file (a file containing only declarations, typically a **.h** file), the source file under test is automatically included in the C++ Test Driver script.

Candidate Classes

For source files containing several classes, you may only want to submit a restricted number of classes to testing.

If no classes are selected, the wizard automatically selects all classes that are defined or implemented in the source(s) under test as follow:

- The class is defined within the source file (*i.e.* the sequence class `<name> { };`).
- At least one of the methods of the class is defined within the source file (*i.e.* a method's body).

Note Classes can only be selected if you have refreshed the File View

before running the Test Generation Wizard.

Simulated, Additional and Included Files

Component Testing for C++

When creating a Component Testing test node for C++, the Component Testing wizard offers the following options for specifying dependencies of the source code under test:

- Simulated files
- Additional files
- Included files

Simulated Files

This option gives the Component Testing wizard a list of source files to simulate or stub upon execution of the test.

A stub is a dummy software component designed to replace a component that the code under test relies on, but cannot use for practicality or availability reasons. A stub can simulate the response of the stubbed component.

Note Declaration and stub files (**.dcl** and **.stb**) for Component Testing for C++ do not appear in the test node. You can access and edit these files from the **Asset Browser** tab of the **Project Explorer**.

In Component Testing for C++, the use of stubs requires Source Code Insertion (SCI) instrumentation of the source code under test.

The Component Testing parser analyzes simulated files and produces an **.stb** stub file written in C++ Test Script Language. When manually creating a test node, use of a separate **.stb** stub file is optional. It is entirely possible to define stubs directly inside the **.otd** C++ Test Driver script.

Additional Files

Additional source files are source files that are required by the test script, but not actually tested. For example, with Component Testing for C++, Visual C++ resource files can be compiled inside a test node by specifying them as additional files.

Additional header files (**.h**) are not handled in the same way as additional body files (**.cc**, **.C**, or **.cpp**):

- **Body files:** With a body file, the Test Generation Wizard considers that the compiled file will be linked with your test program. This means that all *defined* variables and routines are considered as defined, and therefore not stubbed.
- **Header files:** With a header file (a file containing only *declarations*), the Test Generation Wizard considers that all the entities *declared* in the source file itself (not in included files) are defined. Typically, you would use additional header files if you only have a **.h** file under test and a matching object file (**.o** or **.obj**), but not the actual source file (**.cc**, **.C**, or **.cpp**).

You can toggle a source file from *under test* to *additional* by changing the **Instrumentation** property in the Properties Window dialog box.

Additional directories are directories that are declared to only contain additional source files.

Included Files

Included files are normal source files under test. However, instead of being compiled separately during the test, they are included and compiled with the C++ Test Driver script.

Header files are automatically considered as included files, even if they are not specified as such.

Source files under test should be specified as included when:

- The file contains the class definition of a class you want to test
- A function or a variable definition depends upon a type which is defined in the file under test itself
- You need access in your test script to a static variable or function, defined in the file under test

In most cases, you do not have to specify files to be included. The Component Testing wizard automatically generates a warning message in the Output Window, when it detects files that should be specified as included files. If this occurs, rerun the Component Testing wizard, and select the files to be included in the **Include source files** section of the Advanced Options dialog box.

Declaration Files

A declaration file (**.dcl**) ensures that the types, class, variables and functions needed by your test script will be available in your code.

Note Declaration and stub files (**.dcl** and **.stb**) for Component Testing for C++ do not appear in the test node. You can access and edit these files from the **Asset Browser** tab of the **Project Explorer**.

Using a separate **.dcl** file is optional, since it is merely included within the C++ Test Driver script. It is possible to declare types, classes, variables and functions directly within an C++ Test Driver script file.

Typically, **.dcl** files are created by the Component Testing Wizard and do not need to be edited by the user. If you do need to define your own declarations for a test, it is recommended that you do this within the Test Driver script.

Declaration files must be written in C++ Test Script Language and contain native code declarations. See the **Test RealTime Reference Manual** for

details about the language.




C++ Test Reports




Understanding Component Testing for C++ Reports

Test reports for Component Testing for C++ are displayed in Test RealTime's Report Viewer.

The test report is a hierarchical summary report of the execution of a test node. Parts of the report that have **Passed** are displayed in green. **Failed** tests are shown in red.

Report Explorer

The Report Explorer displays each element of a Test Verdict report with a Passed , Failed  or Undefined  symbol:

- Elements marked as Failed  are either a failed test, or an element that contains at least one failed test.
- An Undefined  marker means either that the test was not executed, or that the element contains a test that was not executed AND all executed tests were passed.
- Elements marked as Passed  are either passed tests or elements that contain only passed tests.

Test results are displayed in two parts:

- Test Classes, Test Suites and Test Cases of all the executed C++ Test scripts.
- Class results for the entire Test. Each class contains assertions (WRAP statement), invariants, states and transitions.

Report Header

Each Test Verdict report contains a report header with:

- The path and name of the **.xrd** report file.
- A general verdict for the test campaign: Passed or Failed.
- The number of test cases Passed and Failed. These statistics are calculated on the actual number of test elements (Test Case, Procedure, Stub and Classes) listed sections below.

Note The total number counts the actual test elements, not the number of times each element was executed. For instance, if a test case is run 5 times, of which 2 runs have failed, it will be counted as one *Failed* test case.

Test Script




Each script is displayed with a metrics table containing the number of Test Suite, Test Class, Test Case, Epilogue, Procedure, Prologue and Stub blocks encountered. In this section, statistics reflect the number of times an element occurs in a C++ Test script.

Test Results

For each Test Case, Procedure and Stub, this section presents a summary table of the test status. The table contains the number of times each verification was executed, failed and passed.

For instance, if a Test Case containing three **CHECK** statements is run twice, the reported number of executions will be six, the number of failed verifications will be two, and the number of passed verifications will be four.

The general status is calculated as follows:






Condition	Result	Status
A verification fails		Failed
A verification does not occur		Undefined
All verifications pass on each execution		Passed

Tested Classes

Class results are grouped at the end of the report and sorted in alphabetical order.

For each class the report shows the general status of assertions (**WRAP** statement), invariants, states and transitions.

The general status is computed as follows:

Condition	Result	Status
An assertion or invariant fails		Failed
An assertion or invariant does not occur		Undefined
All assertions or all invariants pass on each execution		Passed
A state is not reached		Not reached
A state has no exit transition		Not fired

When a class does not behave as expected, a table of violations is displayed. A violation is observed at the exit of a state and can be one of the following:

- Multiple: means that several states were reachable at the same time,
- Illegal: means that no state was reachable.

The displayed table gives the number of times a violation has occurred for each state. The status of this table is always Failed.

Understanding Component Testing for C++ UML Sequence Diagrams

During the execution of the test, Component Testing for C++ generates trace data this is used by the UML/SD Viewer. The Component Testing for C++ sequence diagram uses standard UML notation to represent both Contract-Check and Test Driver results.

- Class Contract-check sequence diagrams,
- Test Driver Sequence Diagrams.

Both types of results can appear simultaneously in the same sequence diagram. When using Runtime Tracing with Component Testing for C++, all results are generated in the same sequence diagram.

Illegal and Multiple Transitions

When dealing with state or transition diagrams, Component Testing for C++ adds a custom *observation state*, which is both the initial state and error state. All user-defined states can make a transition towards the *initial/error* state, and this state can transition towards all user-defined states.

At the beginning of test execution, the object is in the initial/error state.

During the test, the object is continuously tested to comply to the user-defined **STATES** and **TRANSITIONS**. There are three possible cases.

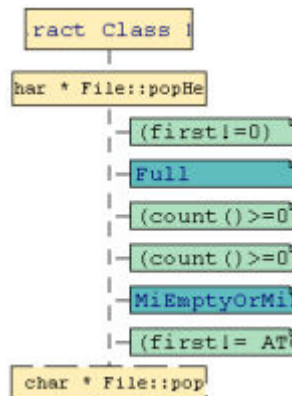
- The transition can be fired to a single state: the current state is set.
- The transition cannot be fired to any of the defined states: in this case, the state switches to the observation state and Component Testing for C++ generates an **ILLEGAL TRANSITION** note.
- The transitions can be fired to two or more states. In this case, the transition diagram is no longer unambiguous. The state is set to the observation state and Component Testing for C++ generates a

MULTIPLE TRANSITION.

When the state diagram is in the *initial/error* state, the transition is still continuously checked, however all user defined states can be potentially fired.

Contract-Check Sequence Diagrams

The following example shows how a typical class contract is represented by Component Testing for C++. C++ classes are represented as vertical lines, like object instances. The events related to the class - method entry and exit, assertion and state chart checks - are attached to the class lifeline.



Methods

For each class, methods are shown with method entry and exit actions:

- Method entry actions have a solid border,
- Method exit actions have a dotted border.

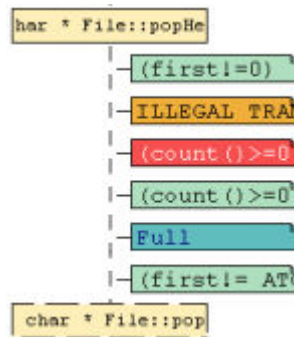
Contract-Checks

Pre and post-conditions, invariants and state verifications are displayed as Notes, attached to the class instance, and contained within the method.

You can click a note to highlight the corresponding OTC Contract-Check script line in the Text Editor window.

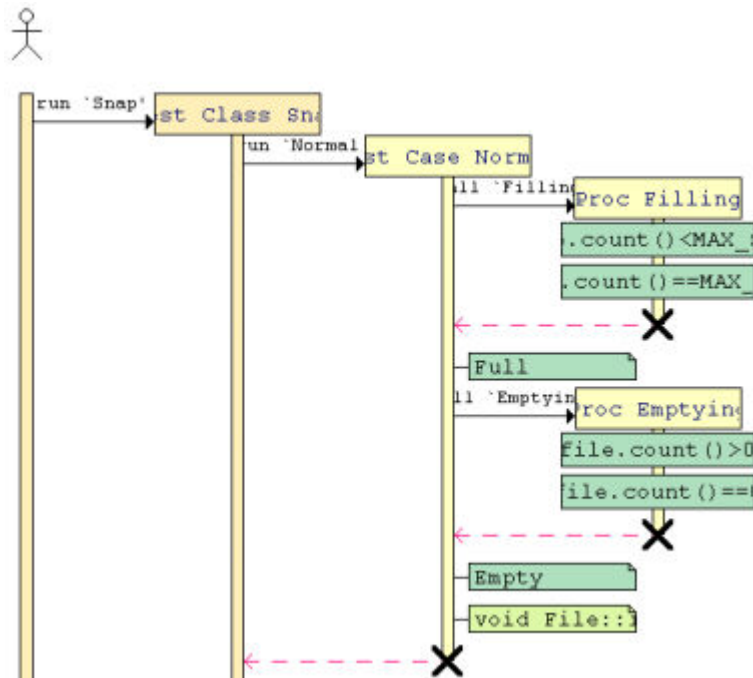
Illegal and Multiple Transitions

State or transition diagram errors are identified as ILLEGAL TRANSITION or MULTIPLE TRANSITION Notes as shown in the following figure:



Test Driver Sequence Diagrams

The following example illustrates typical results generated by a Test Driver script:



Instances

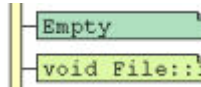
When using a Test Driver script, each of the following C++ Test Script Language keywords are represented as a distinct object instance:

- **TEST CLASS**
- **TEST SUITE**
- **TEST CASE**
- **STUB**
- **PROC**

You can click an instance to highlight the corresponding statement in the Text Editor window.

Checks

Test Driver checks are displayed as Notes attached to the instances, as shown below:



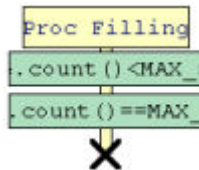
Checks appear in green when the result is *Passed*, and in red when *Failed*.

- **CHECK**
- **CHECK PROPERTY**
- **CHECK STUB**
- **CHECK METHOD**
- **CHECK EXCEPTION**

To distinguish checks that occur immediately from checks that apply to a stub, method or exception, the three latter use different shades of red and green.

You can click an instance to highlight the corresponding statement in the Text Editor window.

Pre and Post-conditions



The following pre and post-condition statements are green (Passed) or red (Failed) actions contained in **STUB** or **PROC** instances.

- **REQUIRE**

- **ENSURE**

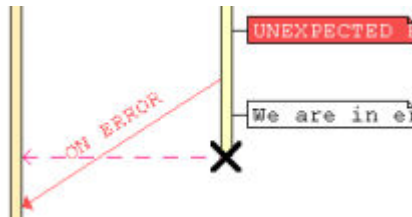
Exceptions

Component Testing for C++ generates **UNEXPECTED EXCEPTION** Notes whenever an unexpected exception is encountered. These notes will be followed by the **ON ERROR** condition.

Error Handling

Whenever a check and a pre- or post-condition generates an error, or an **UNEXPECTED EXCEPTION** occurs, the **ON ERROR** condition is displayed as shown in the following diagrams.

An **ON ERROR BYPASS** condition:



An **ON ERROR CONTINUE** condition:

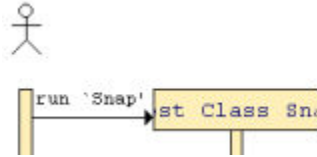


Comments and Prints

COMMENT and **PRINT** statements generate a white note, attached to the corresponding instance.

Messages

Messages can represent either a **RUN** or a **CALL** statement, or a native code stub call, as shown below:



Component Testing for Java

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.

Component Testing for Java uses the standard JUnit testing framework for test harness development. There are two ways of using the JUnit test harness:

- You use the Test RealTime GUI to set up your test campaign, write a JUnit test script, run your tests and view the results
- You import an existing JUnit test harness into the Test RealTime GUI.

The test driver, Target Deployment Port, stubs and dependency files make up a test harness. The test harness interacts with the source code under test and produces test results.

At the same time, your code-under-test can be instrumented with Test RealTime's Runtime Analysis features.

Java Testing Overview

Rational Test RealTime uses JUnit as a standard framework for Component Testing for Java.

The current documentation assumes that you have basic knowledge and understanding of the working principles of:

- JUnit
- Java 2 Platform, Standard Edition (J2SE)
- Java 2 Platform, Micro Edition (J2ME)

Component Testing for Java adapts JUnit to either the J2SE or J2ME framework via Target Deployment Technology.

JUnit Overview

JUnit is a regression testing framework written by Erich Gamma and Kent Beck. JUnit is Open Source Software, released under the IBM's Common Public License Version 0.5 and hosted on the SourceForge website

Please refer to the JUnit documentation for further information about JUnit. More information on JUnit can be found at the following locations:

<http://junit.sourceforge.net>

<http://www.junit.org>

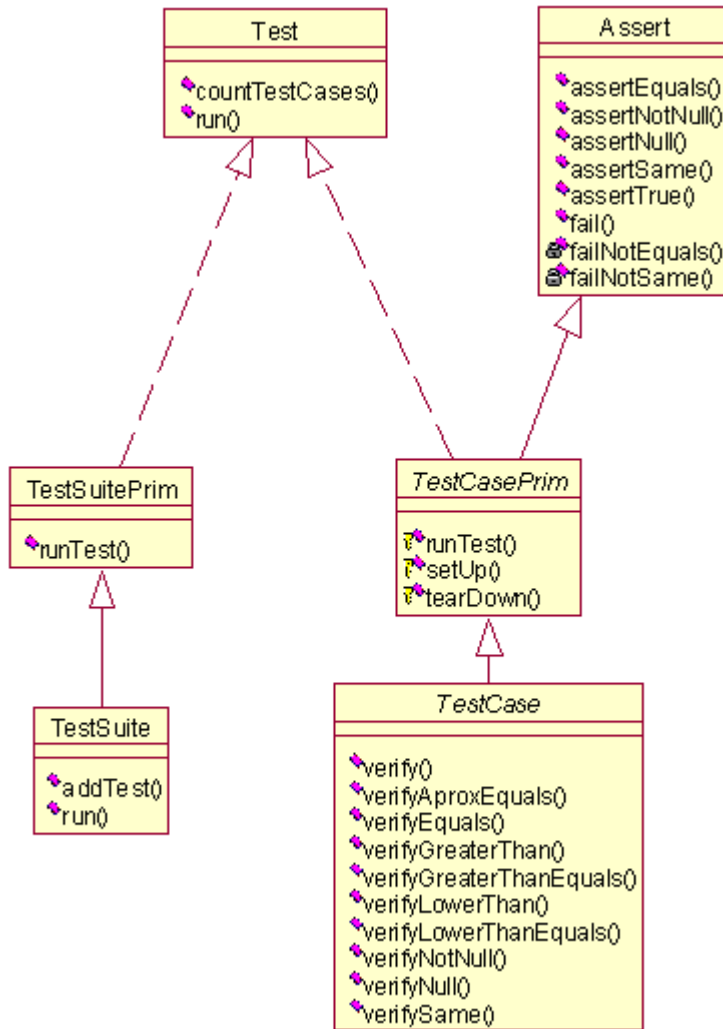
JUnit for J2ME

Basic JUnit was originally written for the J2SE framework. Rational Test RealTime brings offers an additional JUnit implementation for J2ME, referred to as JUnit for J2ME.

Test RealTime JUnit Extensions

Rational Test RealTime extends the JUnit *assert* primitives with a set of *verify* primitives.

The following UML model diagram demonstrates the basic structure of JUnit as well as how Test RealTime extends the JUnit model.



The main difference of the **verify** primitives is that failed **verify** tests do not stop the execution of the test program.

The complete list of extended **verify** primitives can be found in the **Component Testing for Java** section of the **Test RealTime Reference Manual**.

Java Test Nodes

The project structure of the Rational Test RealTime GUI uses *test nodes* to represent your Component Testing test harness.

Test nodes created for Component Testing for Java use the following structure

- **Java Test Node:** represents the Component Testing for Java test harness
- **TestDriver.java:** is the main test driver class
- **Test<Class>.java:** is the test class derived from the **TestCase** class
- **<Class>.java:** is the actual class under test.

In Component Testing for Java, all classes have the **excluded from build** tag except for the main **TestDriver.java** class.

Note JUnit test harnesses that were manually imported into Test RealTime and not created through the Component Testing wizard may not display the correct yellow icons. This is not an issue as long as all files are excluded from the build except for the main test driver class.

Java Test Harness

Component Testing for Java generates a full Java test harness based on JUnit-compliant classes for J2SE and J2ME framework.

The Java test harness can be used for single thread component testing, using the following main JUnit classes:

- ***TestSuite***: This class is a container for multiple test classes derived from ***TestCase***
- ***TestCase***: The basic class that is derived into a series of user-defined test classes
- ***TestResult***: This class returns the results of a given test class:
 - Unexpected errors: unwanted exceptions
 - Failed assertions: produced by the JUnit ***assert*** test primitives
 - Failed verifications: produced by the extended ***verify*** test primitives

The list of extended ***verify*** test primitives can be found in the Test RealTime Reference Manual.

Please refer to the JUnit documentation for further information about JUnit ***assert*** primitives as well as general JUnit documentation. This can be found at the official JUnit Web site:

<http://junit.sourceforge.net>

Test Harness Constraints

Component Testing for Java complies with most JUnit test cases. However, it introduces the two following constraints:

- User test classes must derive from the ***TestCase*** class, or from a ***TestSuite*** that contains one or several ***TestCase*** classes
- The test harness cannot be applied to multi-threaded Java components

You must be especially aware of these constraints when importing existing

test cases into Rational Test RealTime

Naming Conventions

Test class names should be prefixed with ***Test***, as in

```
Test<ClassUnderTest>
```

Where *<ClassUnderTest>* is the name of the class under test. This naming convention enables the test class to use test class primitives.

Test method names must be prefixed with ***test***, as in:

```
test<TestName>
```

Where *<TestName>* is the name of the test.

Test Class Primitives

The test class defines the primitives that create and test the objects under test. The test class primitives are:

- **Creation of the objects under test:** This primitive must create and initialize all the objects under test.

```
void setUp() throws Exception  
setUp:
```

- **End of test:** Use this primitive to insert any code that is required to end the test, such as to set any setUp created objects to null.

```
void tearDown() throws Exception  
tearDown:
```

- **Test Primitives:** The test class must also define as many **test<TestName>** methods as there are tests.

You can inject such a ***TestCase*** into a ***TestSuite***. This way, The ***TestSuite*** automatically creates as many ***TestCases*** as requires and executes a sequential run of all the tests.

Running a Test

To run a series of tests, you must incorporate a *main* inside a **TestCase** or **TestSuite** class, build the *main*, the **TestSuite** and **TestClass**, and execute the run.

In J2ME, these objects can be built in a midlet, which contains only **TestSuite** and **TestCase**, and launches the run on the **start app** primitive. If the test case was generated by Test RealTime, you must comment the main method that was automatically generated.

Example

To test that the sum of two Moneys with the same currency contains a value which is the sum of the values of the two Moneys, write:

```
public void testSimpleAdd() {
    Money m12EUR=new Money(12, "EUR");
    Money m14EUR=new Money(14, "EUR");
    Money expected= new Money(26, "EUR");
    Money result= m12EURdd(m14EUR);
    assertTrue(expected.equals(result));
}
```

Using the TestCase Class

To create a test case in the Component Testing for Java test harness, create a test class by deriving from the **TestCase** of the test harness. Only classes derived from **TestCase** can use the test harness services

1. Create a derived class of **TestCase**:
2. Create a constructor which accepts a String as a parameter and passes it to the superclass. This string must carry the name of the test class, so as to call the correct method.
3. Override the method **runTest()**. This is the method that controls creation and handling of the objects under test as well as the actual verification points: the **verify** and **assert** methods of the **TestCase** class.

Adding Test Primitives

To add a new test, use the **setup** and **teardown** methods to create and configure objects under test.

Such objects belong to the test class. The **setup** method allows you to configure the objects under test. The **teardown** frees the objects.

Running a Test Case

The most convenient way to invoke a test case is to use the constructor with the test name as an argument. For example:

```
TestCase TestStocksObject = new
TestStocks("testStocksValues");
TestCase TestStocksObject2 = new
TestStocks("testStocksAmount");
```

This way, the TestCase object automatically call the public method name that was passed as an argument with the run call.

To use this technique in J2ME, you must first create a runTest() method in the test class which will call the correct function.

Examples

The following series of examples shows how to test a simple class Stocks in the J2SE framework. First, derive the test class from TestCase to check the arithmetic methods:

```
package examples;
import junit.framework.*;
import examples.Stocks.*;
public class TestStocks extends TestCase {
public TestStokcs(String name) {
super(name);
}
public void testStocks1() {
Stock first = new Stock("Company","Dollar",100,1.25);
Stock second = new Stock("Company","Dollar",250,1.25);
Stock added = new Stock(first + second);
```

```

//Display a message in the report.
verifyLogMessage("Check equals for the count of stocks");
verifyEquals("verify equals added count",
added.amountstocks(),
(first.amountstocks()+second.amountstocks()));
}

```

An equivalent implementation for J2ME would be:

```

import j2meunit.framework.*;
import examples.Stocks.*;
public class TestStocks extends TestCase {
public TestStocks(String name) {
super(name);
}
protected void runTest() throws java.lang.Throwable {
if(getTestMethodName().equals("testStocksAmount"))
testStocksAmount ();
else if(getTestMethodName().equals("testStocksValues"))
testStocksValues();
}
}

```

This test class checks for a thrown exception:

```

public void testException3()
{
verifyLogMessage("Check true for RTE");
Throwable toverify= new Throwable("StocksError");
verify(toverify);
//This rate conversion will throw a StockError Exception.
Stock divided = new Stock(first.convertwithrate(0));
}

```

The following example demonstrates an object vector verification:

```

public void testAccounts()
{
Vector RefAccountStocks = new Vector();
RefAccountStocks .addElement( new
Stocks("Company","Dollar",100,1.25));
RefAccountStocks .addElement(new
Stocks("Company2","Dollar",100,2.68));
verifyLogMessage("Verify Equality for Accounts");
verify("verify equal vector", RefAccountStocks,
OtherAccountStocks );
}

```

Component Testing for Java allows you to check timing between events by using the **time** method of the **TestCase** class:

```

public void testTimerOnStocks()

```

```

{
    int idtimer1;
    idtimer1 = createTimer("first timer created");

    //then start the timers.
    timerStart(timer1,"Start 1");
    long val1;
    //Unit is ms.
    val2 = 100;
    verifyLogMessage("Timer report Transaction");
    timerReportEllapsedTime(timer1,"First report of time
before the action");
    Stock dynamicalAccount = first.extractfromWebSite(second);
    verifyEllapsedTime(timer1,val1,"ellapsed 1 with 100");
}

```

In J2SE, run the tests by calling the **run** method of the test class:

```

    TestResult result = TestStockObject.run() ;

    TestResult result = new TestResult();
    TestStockObject.run(result) ;

```

In J2ME, you run the test class by calling the run method of the object under test:

```

    TestResult result = TestStockObject.run() ;
    TestResult result = new TestResult();
    TestStockObject.run(result) ;

```

Using the TestResult Class

The TestResult object is used to produce dynamic test results during the execution of a TestCase or TestSuite. The TestResult class offers the same behavior in J2SE and J2ME.

TestResult provides the following methods:

verifyCount()

Returns the number of failed **verify** calls during test execution.

errorCount()

Returns the number of errors (unexpected exceptions) encountered during test execution.

failureCount()

Returns the number of failed **assert** calls during test execution.

Using the TestSuite Class

After writing several simple test cases, you will need to group the individual test classes derived from **TestCase** and run them together. This can be done with the **TestSuite** class.

There are three ways of constructing a **TestSuite**:

- By explicit calls to **TestCase**
- By test class (J2SE only)
- By creating the **suite()** method

A **TestSuite** can only contain objects derived from **TestCase** or a **TestSuite** that contains a **TestCase**.

Construction by Explicit calls to TestCase

You can add the explicit calls to the **TestSuite** instance by instance, as in the following example:

```
TestSuite suiteStocks = new TestSuite() ;
suiteStocks.addTest(new TestStocks("testStocksValues"));
suiteStocks.addTest(new TestStocks("testStocksAmount"));
```

You can also directly pass the test object. In this case, the **TestSuite** automatically builds all the test classes from the public method names:

```
TestSuite suiteStocks = new TestSuite() ;
suiteStocks.addTest(TestStocks.class);
```

Such a **TestSuite** can be contained in another **TestSuite**.

Construction by Test Class

```
TestSuite suiteAllTests = new TestSuite() ;  
suiteAllTests.AddTest (SuiteStocks) ;  
suiteAllTests.AddTest (OthersTests.class) ;
```

Creating a *suite()* Method

In J2ME, to be able to build a **TestSuite** from a test class, you cannot pass the class object as sole argument. To resolve this, an extra **suite()** method is added to the test class, which returns a valid **TestSuite**:

```
TestSuite suiteStocks = new TestSuite() ;  
suiteStocks.addTest (new TestStocks ().suite ()) ;
```

Running a TestSuite,

In J2SE, you run a test suite exactly as you would run a test class, either by producing a **TestResult** object, or by modifying the **TestResult** passed as a parameter, as in the following examples:

```
TestResult result = suiteAllTests.run(result) ;  
TestResult result = new TestResult () ;  
suiteAllTests.run(result) ;
```

In J2ME, in order to save memory, the **TestSuite** destroys the last **TestCase** instance after each run.

Simulated and Additional Classes

Component Testing for Java

When creating a Component Testing test node for Java, the Component Testing wizard offers the following options for specifying dependencies of the source code under test:

- Simulated files
- Additional files

Simulated Files

This option gives the Component Testing wizard a list of source files to simulate or stub upon execution of the test.

A stub is a dummy software component designed to replace a component that the code under test relies on, but cannot use for practicality or availability reasons. A stub can simulate the response of the stubbed component.

See Java Stubs for more information about JUnit stub handling.

Additional Files

Additional source files are source files that are required by the test script, but not actually tested.

You can toggle a source file from *under test* to *additional* by changing the **Instrumentation** property in the Properties Window dialog box.

Additional directories are directories that are declared to only contain additional source classes.

Java Stubs

Component Testing for Java supports the following verification methods for stubbed classes:

- Stub failure detection
- Stub sequence mechanism

Stub Failure

In the Component Testing for Java test harness, stubs can declare an error.

To check for the existence of a stub error, use the following global call type:

```
verify("Test single  
sequence", TestSynchroStub.areStubfail(this), true);
```

Stub Sequence Mechanism

The Component Testing for Java test harness provides a stub logging mechanism. The purpose of this mechanism is to check that calls to a stubbed object are achieved in a correct order.

To do this, two objects are provided:

- **TestSynchroStub:** checks that stub calls follow a given sequence.
- **StubInfo:** returns the method entry or exit information type:

Example

The purpose of this example is to check the stubbed class *StubbedOne* is called with the following sequence:

- Entry into the method *methodone*
- From this method, entry and exit of the method *m1* of the same class

This translates into the following test class:

```
StubSequence testof = new StubSequence(this);  
Class StubbedClass = new StubbedObject().getClass();  
testof.addEltToSequence( StubbedClass, "methodone", StubInfo.ENTER) ;  
testof.addEltToSequence( StubbedClass , "m1", StubInfo.ENTER) ;  
testof.addEltToSequence( StubbedClass , "m1", StubInfo.EXIT) ;  
verifyLogMessage("Check false for stub not support for the stubs");  
verify("Test single  
sequence", TestSynchroStub.isSeqRespected(testof), true);
```

Importing a JUnit Test Campaign

JUnit is becoming an industry standard in the field of testing Java software.

Rational Test RealTime can import your existing JUnit test campaigns. This

requires manually building a new Java test node that contains:

- The classes under test
- The test classes derived from *TestCase*
- All other test harness components

After this, you must ensure that only the main test driver class is passed on to the Java compiler. To do this, exclude all other classes from the build.

Test Harness Constraints

Component Testing for Java complies with most JUnit test cases. However, it introduces the two following constraints:

- User test classes must derive from the *TestCase* class, or from a *TestSuite* that contains one or several *TestCase* classes
- The test harness cannot be applied to multi-threaded Java components

You must be especially aware of these constraints when importing existing JUnit test classes into Rational Test RealTime.

To import an existing JUnit test harness:

1. In the **Project Explorer**, select the **Project View** and right-click the Project node.
2. From the pop-up menu, select **Add Child** and **Component Testing for Java**.
3. Enter the name of the new Java test node.
4. In the Project Explorer, right-click the Java test node.
5. From the pop-up menu, select **Add Child** and **Files**.
6. Locate and select the classes under test and the JUnit test classes.
7. Click **OK**.

8. Exclude from the build all Java classes, except the *main* test driver class.

J2ME Specifics

Component Testing for Java supports the Java 2 Platform Micro Edition (J2ME) through a specialized version of the JUnit testing framework.

This framework requires that you manually perform the two following additional steps:

1. Create a test suite class **Suite()** that transforms a test class into a J2ME test suite.
2. Create a **runTest()** primitive that transforms the name of the test case into a relevant call to the test function.

The objects under test must belong to the test class and must have been initialized in the **setUp** method.

The following code sample is a **runTest** selection method for J2ME, which switches the correct test method depending on the name of the test case:

```
protected void runTest() throws java.lang.Throwable {
    if (getTestMethodName().equals("testOne"))
        testOne();
    else if (getTestMethodName().equals("testTwo"))
        testTwo();
}
```

Building a Test Suite

The two following methods demonstrate how to build a test suite from a J2ME test case.

```
public Test suite() {
    return new TestSuite(new TestOne().getClass(), new String[]
        {"testOne"});
}
```

```
public static Test suite() {
    TestSuite suite = new TestSuite();
}
```

```
suite.addTest(new TestMine().suite());
suite.addTest(new TestMine2().suite());
return suite;
}
```

Integration of Objects Under Test

The objects under test must belong to the test class and must have been initialized in the **setUp** method.



Java Test Reports



Understanding Java Test Reports

Test reports for Component Testing for Java are displayed in Test RealTime's Report Viewer.

The test report is a hierarchical summary report of the execution of a test node. Parts of the report that have **Passed** are displayed in green. **Failed** tests are shown in red.

Report Explorer

The Report Explorer displays each element of a Test Verdict report with a *Passed*  or *Failed*  glyph:

- Elements marked as Failed  are either a failed test, or an element that contains at least one failed test.
- Elements marked as Passed  are either passed tests or elements that contain only passed tests.

Test results are displayed in two parts:

- TestClasses, TestSuites and derived test cases of all the executed JUnit scripts.
- Class results for the entire Test.

Report Header

Each Test Verdict report contains a report header with:

- The path and name of the **.xrd** report file.
- A general verdict for the test campaign: *Passed* or *Failed*.
- The number of test cases *Passed* and *Failed*. These statistics are calculated on the actual number of test elements (Test Case, Procedure, Stub and Classes) listed sections below.

Note The total number counts the actual test elements, not the number of times each element was executed. For instance, if a test case is run 5 times, of which 2 runs have failed, it will be counted as one *Failed* test case.

Test Script

Each script is displayed with a metrics table containing the number of **TestSuite**, **TestClass** and derived test case encountered. In this section, statistics reflect the number of times an element occurs in a JUnit script.

Test Results

For each test case, this section presents a summary table of the test status. The table contains the number of times each verification was executed, failed and passed.

For instance, if a Test Case containing three ***assert*** functions is run twice, the reported number of executions will be six, the number of failed verifications will be two, and the number of passed verifications will be four.

The general status is calculated as follows:

Condition	Result	Status
-----------	--------	--------

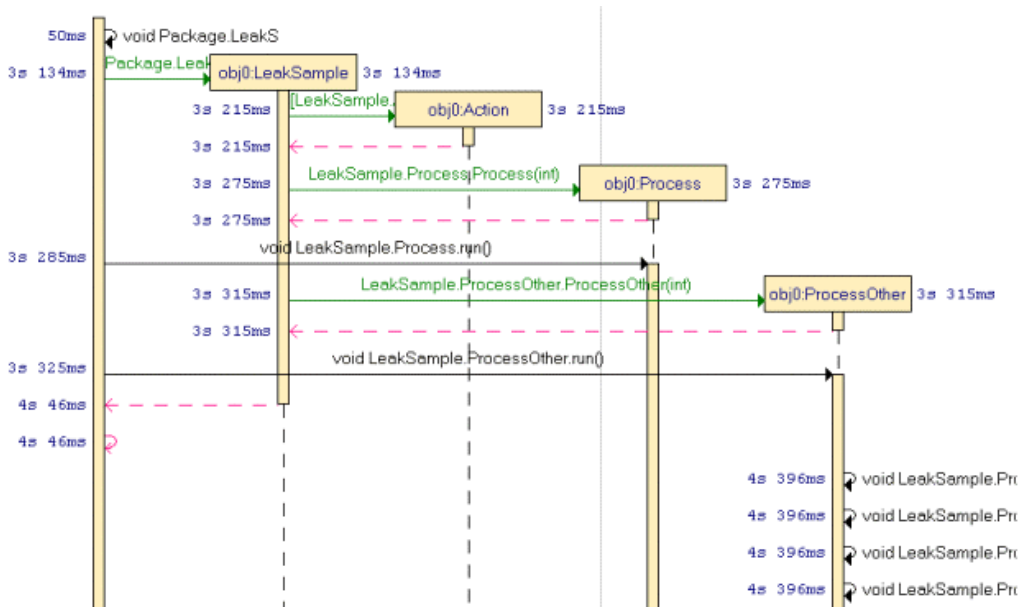
A verification fails	❌	Failed
All verifications pass on each execution	✅	Passed

Understanding Java Component Testing UML Sequence Diagrams

During the execution of the test, Component Testing for Java generates trace data this is used by the UML/SD Viewer. The sequence diagram uses standard UML notation to represent JUnit test results.

When using Runtime Tracing with Component Testing for Java, all results are generated in the same sequence diagram.

The following example illustrates typical results generated by a JUnit test script:



Instances

Each of the following classes are represented as a distinct object instance:

- TestSuite
- Derived test case classes

You can click an instance to highlight the corresponding statement in the Text Editor window.

Checks

JUnit ***assert*** and ***verify*** primitives are displayed as Passed ("✓") or Failed ("✗") glyphs attached to the instances.

You can click any of these glyphs to highlight the corresponding statement in the Text Editor window.

Exceptions

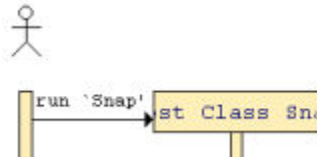
Component Testing for Java generates **UNEXPECTED EXCEPTION** Notes whenever an unexpected exception is encountered.

Comments

Calls to ***verifyLogMessage*** generate a white note, attached to the corresponding instance.

Messages

Messages can represent either a run or a call statement as shown below:



System Testing for C

Test RealTime's System Testing feature is the first commercial automated feature dedicated to testing message-based applications. Until now most of the projects developing real-time, embedded or distributed systems spent a fair amount of resources building dedicated test beds. Project managers can now save time and money by avoiding this costly, non-core-business activity.

System Testing helps you solve complex testing issues related to system interaction, concurrency, and time and fault tolerance by addressing the functional, robustness, load, performance and regression testing phases from small, single threads or tasks up to very large, distributed systems.

With Test RealTime's System Testing feature, test engineers can easily design, code and execute virtual testers that represent unavailable portions of the system under test - SUT - and its environment.

System Testing is recommended for testing:

- Telecommunication and networking equipment using standard protocols
- Aerospace equipment using standard or proprietary operating systems and a communication bus (ARINC, 1553, etc.)
- Automotive Electronic Control Units (ECUs) based on OSEK operating system, or appliance systems such as Driver Information Systems
- Distributed applications based on message-oriented middleware

- (MOM) such as MQseries, Tuxedo or an in-house MOM
- Applications developed using Rational Rose RealTime

System Testing Overview

Circular Trace Buffer

The *circular trace buffer* memorizes System Testing for C traces and flushes them to the **.rio** output file when the Virtual Tester ends or at a specified point in the **.pts** test script.

To activate the circular trace buffer option or to set the size of the buffer, see Test Script Compiler Settings.

How the Circular Buffer Works

During execution of the test node, System Testing accumulates traces in the buffer. When the buffer fills up, new traces replace old ones, as shown in the following diagram, without flushing to file.

Contents of the Buffer

By default, the buffer stores all traces.

Use the **TRACE_OFF** instruction in your **.pts** System Testing for C test script to trace only scenario begins and ends, environment blocks, procedure blocks, PRINT instructions, and failed instructions.

Use the **TRACE_ON** instruction to resume default behavior.

See the **Rational Test RealTime Reference Manual** for detailed information on **.pts** test script instruction.

Flushing the Buffer on the Disk

By default, the buffer is flushed to a file when the Virtual Tester ends.

You may flush the buffer at any point in the **.pts** test script by using the **FLUSH_TRACE** instruction.

You cannot call the **FLUSH_TRACE** instruction, either directly or indirectly, from a **CALLBACK** or **PROCSEND** block.

See the **Rational Test RealTime Reference Manual** for detailed information on **.pts** test script instruction.

Note The **TRACE_ON**, **TRACE_OFF** and **FLUSH_TRACE** instructions only apply when the Circular Trace Buffer option is selected.

System Testing Supervisor

Test RealTime System Testing manages the simultaneous execution of Virtual Testers distributed over a network. When using System Testing feature of Test RealTime, the machine running Test RealTime runs a Supervisor process, whose job 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 a deployment script, generated by the Virtual Tester Configuration and Virtual Tester Deployment dialog boxes, 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.

Agents and Virtual Testers

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.

System Testing generates Virtual Testers from a test script according to the declared instances. The System Testing Supervisor, which runs in the Test RealTime host machine, is in charge of deploying and controlling remote Virtual Testers.

Note A System Testing Agent 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 Test Script Compiler provides several ways to generate the Virtual Testers.

System Testing Agents

Installing System Testing Agents

When using Virtual Testers on remote target hosts, a daemon must be running on the target to act as an interface between the virtual tester and the System Testing Supervisor. This daemon is known as the System Testing Agent.

Note Always make sure that the version of the System Testing Agent matches the version of Test RealTime. If you have upgraded from a

previous version of Test RealTime, you must also update all System Testing Agents on remote machines.

The installation directory of System Testing includes the following necessary agent files:

- **atsagtd.bin**: the agent executable binary for UNIX
- **atsagtd.exe**: the agent executable binary for Windows
- **atsagtd**: the agent launcher for UNIX when using *inetd*
- **atsagtd.sh**: a UNIX shell script that starts **atsagtd.bin**

Installing the Agent

There are two methods for installing the System Testing Agent:

- **Manual launch**
- **Inetd daemon installation**

To install a System Testing Agent for manual execution:

This procedure does not require system administrator access, but launching of the agent is not fully automated.

1. Copy **atsagtd.bin** or **atsagtd.exe** to a directory on the target machine.
2. On the target machine, set the **ATS_DIR** environment variable to the directory containing the agent binaries.
3. Add that same agent directory to your **PATH** environment variable.

Note You can add these commands to the user configuration file: **login**, **.cshrc** or **.profile**.

4. On UNIX systems, create an agent access file **.atsagtd** file in your home directory. On Windows create an **atsagtd.ini** file in the agent installation directory. See System Testing Agent Access Files.

5. Move the agent access file to your chosen base directory, such as the directory where the Virtual Testers will be launched.
6. Launch the agent as a background task, with the port number as a parameter. By default, this number is 10000.

```
atsagtd.bin <port number>&  
atsagtd <port number>
```

To install a System Testing Agent with *inetd*:

This procedure is for UNIX only. Launching agents on target machines is automatic with *inetd*.

With this method, the *inetd* daemon runs the **atsagtd.sh** shell script that initializes environment variables on the target machine and launches the System Testing Agent.

1. Copy **atsagtd.sh** and **atsagtd.bin** to a directory on the target machine.
2. On the target machine, set the **ATS_DIR** environment variable to the directory containing the agent binaries.
3. Add that same agent directory to your **PATH** environment variable.

Note You can add these commands to the user configuration file: **login**, **.cshrc** or **.profile**.

4. Log on as root on the target machine.
5. Add the following line to the **/etc/services** file:

```
atsagtd <port number>/tcp
```

The agent waits for a connection to *<port number>*. By default, System Testing uses port 10000.

Note If NIS is installed on the target machine, you may have to update the NIS server. You can check this by typing **ypcat services** on the target host.

6. Add the following line to the **/etc/inetd.conf** file:

```
atsagtd stream tcp nowait <username> <atsagtd path>
```

`<atsagtd path>`

where `<username>` is the name of the user that will run the agent on the target machine and `<atsagtd path>` is the full path name of the System Testing Agent executable file **atsagtd**.

7. To reconfigure the `inetd` daemon, use one of the following methods:
 - Type the command `/etc/inetd -c` on the target host.
 - Send the **SIGHUP** signal to the running `inetd` process.
 - Reboot the target machine.
8. In some cases, you might need to update the file **atsagtd.sh** shell script to add some environment variables to the target machine.
9. Return to your user account and create an agent access file **.atsagtd** file in your home directory. See System Testing Agent Access Files.

Troubleshooting the agent

To check the installation, type the following command on the host running Test RealTime:

```
telnet <target machine> <port number>
```

where `<port number>` is the port number you specified during the installation procedure. By default, System Testing uses port 10000. After the connection succeeds, press **Enter** to close the connection.

If the connection fails, try the following steps to troubleshoot the problem:

- Check the target hostname and port.
- Check the Agent Access File.
- Check the target hostname and port in the `atsagtd.sh` shell script.
- Check the `/etc/services` and `/etc/inetd.conf` files on the target machine.
- If you are using NIS services on your network, check the NIS configuration.

System Testing Agent Access Files

The **.atsagtd** (UNIX) or **atsagtd.ini** (Windows) agent access file is an editable configuration file that secures access to System Testing Agents and contains a list of machines and users authorized to execute agents on that machine, with the following syntax:

```
<Test RealTime hostname> <username>
```

A plus sign + can be used as a wildcard to provide access to all users or all workstations.

The minus sign - suppresses access to a particular user.

You can add comments to the agent access file by starting a line with the # character:

Example

```
# This is a sample .atsagtd or atsagtd.ini file.
# The following line allows access from user jdoe on a
machine named workstation
workstation jdoe

# The following line allows access from all users of
workstation
workstation +

# The following allows access from jdoe on any host
+ jdoe

# The following allows access to all users except anonymous
from the machine workstation
workstation +
workstation -anonymous
```

Configuring Virtual Testers

The Virtual Tester Configuration dialog box allows you to create and configure a set of Virtual Testers that can be deployed for System Testing.

To open the Virtual Test Configuration dialog box:

1. In the **Project Explorer**, right-click a **.pts** test script.
2. From the pop-up menu, select **Virtual Tester Configuration**.

Note The Virtual Tester Configuration box is also included as part of the System Testing Wizard when you are setting up a new activity.

Virtual Tester List

Use the Virtual Tester List to create a **New** Virtual Tester, **Remove** or **Copy** an existing one.

Select a Virtual Tester in the Virtual Tester List to apply any changes in the property tabs on the right.

General Tab

This tab specifies an instance and target deployment to be assigned to the selected Virtual Tester.

- **VT Name:** This is the name of the Virtual Tester currently selected in the **Virtual Tester List**.
- **Implemented INSTANCE:** Use this box to assign an instance, defined in the **.pts** test script, to the selected virtual tester. This information is used for Virtual Tester deployment. Select **Default** to specify the instance during deployment.
- **Target:** This specifies the Target Deployment Port compilation parameters for the selected Virtual Tester.
- **Configure Settings:** This button opens the Configuration Settings dialog for the selected Virtual Tester node.

Scenario Tab

Use this tab to select one or several scenarios as defined in the **.pts** test script. During execution, the Virtual Tester plays the selected scenarios.

Family Tab

Use this tab to select one or several families as defined in the **.pts** test script. During execution, the Virtual Tester plays the selected families.

Debugging Virtual Testers

In some cases, you may want to observe how your system under test reacts when an error occurs and the consequences of this error on the whole process, without stopping the Virtual Tester.

By default, when an error occurs in a block, the execution of the block is interrupted. To prevent interruption, use the virtual tester debug mode.

You can statically activate the debug mode by compiling the generated Virtual Tester with the **ATL_SYSTEMTEST_DEBUG** variable, as in the following example:

```
cc -c -I$ATLTGT/lib/ -DATL_SYSTEMTEST_DEBUG <source.c>
```

where **\$ATLTGT** is the current TDP directory.

Deploying Virtual Testers

The Virtual Tester Deployment Table allows to deploy previously created Virtual Testers.

To open the Virtual Tester Deployment Table

1. In the **Project Explorer**, right-click a System Testing node.
2. From the pop-up menu, select **Deployment Configuration**.
3. Select **Advanced Options** and click **Rendezvous**.

Note The Virtual Tester Deployment Table is also included in the System Testing Wizard when you are setting up a new activity.

Virtual Tester Deployment Table

Use the **Add**, **Remove** or **Copy** buttons to modify the list. Each line represents one or several executions of a Virtual Tester assigned to an instance, target host, and other parameters.

- **Number of Occurrences:** Specifies the number of simultaneous executions of the current line.
- **Virtual Tester Name:** Specifies one of the previously created Virtual Testers.
- **Instance:** Specifies the instances assigned to this Virtual Tester. If an instance was specifically assigned in the Virtual Tester Configuration box, this cannot be changed. Select **<all>** only if no INSTANCE is defined in the test script.
- **Network Node:** This defines the target host on which the current line is to be deployed. You can enter a machine name or an IP address.

Advanced Options

Click the **Advanced Options** button to add the following columns to the Virtual Tester Deployment Table, and to add the **Rendezvous...** button.

- **Agent TCP/IP Port:** This specifies the port used by the System Testing Agents to communicate with Test RealTime. By default, System Testing uses port 10000.
- **Delay:** This allows you to set a delay between the execution of each line of the table.
- **First Occurrence ID:** This specifies the unique occurrence ID identifier for the first Virtual Tester executed on this line. The occurrence ID is automatically incremented for each number of instances of the current

line. See *Communication Between Virtual Testers* for more information.

- **Start Routine:** This specifies the name of the function containing the Virtual Tester, for use in multi-threaded or RTOS environments, if the starting procedure is not **main()**.

Click the **Rendezvous Configuration** button to set up any rendezvous members.

File System Limitations

Deployment of the Virtual Testers results in the creation of an **.spv** deployment script. This script contains file system commands, such as **CHDIR**. If you are deploying the test to a target platform that does not support a file system, you must edit the **.spv** script manually.

Editing the Deployment Script

The System Testing Supervisor actually runs a script, which is automatically generated by configuring Virtual Testers and deploying Virtual Testers.

In some cases, you will need to manually edit the script. To do this, you first have to generate an **.spv** deployment script in your workspace.

To generate a deployment script

1. In the **Project Explorer**, right-click a System Testing node.
2. From the pop-up menu, select **Generate Deployment Script**.
3. Enter a name for the generated script.

If you decide to manually maintain a deployment script, you must ensure that any pathnames and other parameters remain up to date with the rest of the System Testing node.

For information on the **.spv** script command language, please refer to the

Test RealTime Reference Manual.

Optimizing Execution Traces

Each Virtual Tester generates a trace file during its execution. This trace file is used to generate the System Testing Report.

You may want to adapt the volume of traces generated at execution time. For example, each Virtual Tester saves its execution traces in an internal buffer that you can configure.

To optimize execution trace output, use the Execution Traces area in the Test Script Compiler Settings dialog box.

- By default, System Testing generates a normal trace file.
- Select **Time stamp only** to generate traces for each scenario begin and end, all events, and for error cases. This option also generates traces for each **WAITTIL** and **PRINT** instruction. Use this option for load and performance testing, if you expect a large quantity of execution traces and you want to store all timing data.
- Select **Block start/end only** to generate traces for each scenario beginning and end, all events, and for all error cases.
- Select **Error only** to generate traces only if an error is detected during execution of the application. This report will be incomplete, but the report will show failed instructions as well as a number of instructions that preceded the error. This number depends on the Virtual Tester's trace buffer size. Use this option for endurance testing, if you expect a large quantity execution traces.

In addition to the above, you can select the **Circular trace** option for strong real-time constraints when you need full control over the flush of traces to disk. If you want to still store a large amount of trace data, specify a large buffer.

Setting Up Rendezvous Members

When you have used Rendezvous points in your **.pts** test script, it is necessary to indicate the number of members that the supervisor must expect at each rendezvous.

The **Rendezvous Members** dialog box is an advanced option of the Virtual Tester Configuration.

To specify the number of members for each rendezvous:

1. In the **Project Explorer**, right-click a System Testing node.
2. From the pop-up menu, select **Deployment Configuration**.
3. Select **Advanced Options** and click **Rendezvous**.
4. For each rendezvous encountered in the **.pts** test script, select a number of rendezvous members.

Select **AutoGenerate** to automatically compute the number of members in each Rendezvous. In some cases, such as when rendezvous are placed in an exception, this option cannot provide correct information to the supervisor.

5. Click **OK**.

System Testing in a Multi-Threaded or RTOS Environment

When Virtual Testers must be executed as a threaded part of a UNIX or Windows process, or on RealTime Operating Systems (RTOS) you must take several constraints into account:

- The Virtual Tester should be generated as a function and not a main program.
- You must consider the configuration of the Virtual Testers' execution.

There are memory management constraints:

- There is no dynamic memory allocation.
- Stacks are small.
- Virtual Testers share global data.
- Configuration of Virtual Tester execution.

Virtual Tester as a Thread or Task

When using a flat-memory RTOS model, the Virtual Testers can run as a process thread or as a task in order to avoid conflicts with the application under test's global variables.

Moreover, the Target Deployment Port is fully reentrant. Therefore, you can run multiple instances of a Virtual Tester in the same process. The system runs each process as a different process thread.

In this case, the Test Script Compiler generates the virtual tester source code without a *main()* function, but with a user function.

To configure System Testing to run in multi-threaded mode, select the **Not shared** option in Test Script Compiler Settings.

Multiple Instances of a Same Virtual Tester

Multiple instances of a same Virtual Tester can run simultaneously on a same target. In this case, you need to protect the Virtual Tester threads in the same process against access to global variables.

The **Not Shared** setting in Test Script Compiler Settings allows you to specify global variables in the test script that should remain unshared by separate Virtual Tester threads. When selected, multiple instances of a Virtual Tester can all run in the same process.

You can share some global static variables in order to reuse data among different Virtual Testers by using the **SHARE** command in the **.pts** test

script. See the Rational Test RealTime Reference Manual for information about the System Testing Language.

Virtual Tester Thread Starter Program

In a multi-thread environment, the only way to start the Virtual Tester threads is to write a program, specifying:

The name of the execution trace file

The name of the instance to be started

To do this, use the **ATL_T_ARG** structure, defined in the **ats.h** header file of the Target Deployment Port.

Example

```
#include <stdio.h>
#include <sched.h>
#include <pthread.h>
#include <errno.h>
#include "TP.h"
extern ATL_T_THREAD_RETURN *start(ATL_PT_ARG);
int main(int argc, char *argv[])
{
    pthread_t thrTester_1,thr_Tester_2;
    pthread_attr_t pthread_attr_default;
    ATL_T_ARG arg_Tester_1, arg_Tester_2;
    int status;
    arg_Tester_1.atl_riofilename = "Tester_1.rio";
    arg_Tester_1.atl_filters = "";
    arg_Tester_1.atl_instance = "Tester_1";
    arg_Tester_1.atl_occid = 0;
    arg_Tester_2.atl_riofilename = "Tester_2.rio";
    arg_Tester_2.atl_filters = "";
    arg_Tester_2.atl_instance = "Tester_2";
    arg_Tester_2.atl_occid = 0;
    pthread_attr_init(&pthread_attr_default);
    /* Start Thread Tester 1 */

    pthread_create(&thrTester_1,&pthread_attr_default,start,&arg
_Tester_1);
    /* Start Thread Tester 2 */
```

```

pthread_create(&thrTester_2, &pthread_attr_default, start, &arg
_Tester_2);
/* Both Testers are running */
/* Wait for the end of Thread Tester 1 */
pthread_join(thrTester_1, (void *)&status);
/* Wait for the end of Thread Tester 2 */
pthread_join(thrTester_2, (void *)&status);
return(0);
}

```

System Testing for C Test Scripts

Basic Structure

The overall structure of a C and Ada test script must follow these rules:

- A test script always starts with the **HEADER** keyword.
- A test script is composed of one or several scenarios.

The basic structuring statements are:

- **HEADER:** Specifies the name of the test script, the version of the tested system, and the version of the test script. This information will be included in the test report.
- **SCENARIO:** Indicates the beginning of a **SCENARIO** block. A **SCENARIO** block ends with an **END SCENARIO** statement. A **SCENARIO** block can be iterated multiple times using to the **LOOP** keyword.
- **FAMILY:** Qualifies the scenario and all its sub-scenarios. The **FAMILY** attribute is optional. A list of qualifiers can be given such as: **FAMILY nominal, structural**.

Each scenario can be split into sub-scenarios.

Example

```

HEADER "Registering", "1.0", "1.0"
SCENARIO basic_registration

```

```

FAMILY nominal
-- The body of my basic_registration test

END SCENARIO
SCENARIO extended_registration
FAMILY robustness
SCENARIO reg_priv_area
-- The body of my reg_priv_area test
END SCENARIO -- reg_priv_area
SCENARIO reg_pub_area LOOP 10
-- The body of my reg_pub_area test
END SCENARIO -- reg_priv_area
END SCENARIO

```

Include Statements

To avoid writing large test scripts, you can split test scripts into several files and link them using the **INCLUDE** statement.

This instruction consists of the keyword **INCLUDE** followed by the name of the file to include, in quotation marks (" ").

INCLUDE instructions can appear in high- and intermediate-level scenarios, but not in the lowest-level scenarios.

You can specify both absolute or relative filenames. There are no default filename extensions for included files. You must specify them explicitly.

Example

```

HEADER "Socket validation", "1.0", "beta"
INCLUDE "../initialization"

SCENARIO first
END SCENARIO

SCENARIO second
  INCLUDE "scenario_3.pts"
  SCENARIO level2
    FAMILY nominal, structural
    . . .
  END SCENARIO
END SCENARIO

```

Procedures

You can also use procedures to build more compact test scripts. The following are characteristics of procedures:

- They must be defined before they are used in scenarios.
- They do not return any parameters.

A procedure begins with the keyword **PROC** and ends in the sequence **END PROC**. For example:

```
HEADER "Socket Validation", "1.0", "beta"
PROC function ()
...
END PROC
SCENARIO first
...
CALL function ()
...
END SCENARIO
SCENARIO second
SCENARIO level2
FAMILY nominal, structural
...
END SCENARIO
END SCENARIO
```

A procedure can call sub-procedures as long as these sub-procedures are located above the current procedure.

Procedure blocks can take parameters. When defining a procedure, you must also specify the input/output parameters.

Each parameter is described as a type followed by the name of the variable.

The declaration syntax requires, for each argument, a type identifier and a variable identifier. If you want to use complex data types, you must use either a macro or a C or C++ type declaration.

Example

In the following example, the argument to procedure **function1** is a character string of 35 bytes. The arguments to procedure **function2** are an integer and a pointer to a character.

```
HEADER "Socket Validation", "1.0", "beta"
#typedef char string[35];
#define ptr_car char *
PROC function1 (string a)
...
END PROC
PROC function2 (int a, ptr_car b)
...
END PROC
SCENARIO first
...
CALL function1 ( "foo" )
...
END SCENARIO
```

Flow Control

Several execution flow instructions let you develop algorithms with multiple branches.

System Testing **.pts** test script flow control instructions include:

- Conditions
- Iterations
- Multiple Conditions

Conditions

The **IF** statement comprises the keywords **IF**, **THEN**, **ELSE**, and **END**. It lets you define branches and follows these rules:

- The test following the keyword **IF** must be a Boolean expression in C or C++.
- IF instructions can be located in scenarios, procedures, or environment

blocks.

- The **ELSE** branch is optional.

The sequence **IF** (test) **THEN** must appear on a single line. The keywords **ELSE** and **END IF** must each appear separately on their own lines.

Example

```
HEADER "Instruction IF", "1.0", "1.0"
#int IdConnection;
SCENARIO Main
  COMMENT connection
  CALL socket(AF_UNIX, SOCK_STREAM, 0)@@IdConnection
  IF (IdConnection == -1) THEN
    EXIT
  END IF
END SCENARIO
```

Iterations

The **WHILE** instruction comprises the keywords **WHILE** and **END**. It lets you define loops and follows these rules:

- The test following the keyword **WHILE** must be a C Boolean expression.
- The **WHILE** instructions can be located in scenarios, procedures, or environment blocks.

The sequence **WHILE** (test) and the keyword **END WHILE** must each appear separately on their own lines.

Example

```
HEADER "Instruction WHILE", "", ""
#int count = 0;
#appl_id_t id;
#message_t message;
SCENARIO One
  FAMILY nominal
  CALL mbx_init(&id) @ err_ok
  VAR id.applname, INIT="JUPITER"
  CALL mbx_register(&id) @ err_ok
```

```

    VAR message, INIT={
& type=>DATA,
& applname=>"SATURN",
& userdata=>"hello world!"}
    WHILE (count<10)
        CALL mbx_send_message(&id,&message) @ err_ok
        VAR count, INIT=count+1
    END WHILE
    CALL mbx_unregister(&id) @ err_ok
    CALL mbx_end(&id) @ err_ok
END SCENARIO

```

Multiple Conditions

The multiple-condition statement **CASE** comprises the keywords **CASE**, **WHEN**, **END**, **OTHERS** and the arrow symbol =>.

CASE instructions follow these rules:

- The test following the keyword **CASE** must be a C or C++ Boolean expression. The keyword **WHEN** must be followed by an integer constant.
- The keyword **OTHERS** indicates the default branch for the **CASE** instruction. This branch is optional.
- **CASE** instructions can be located in scenarios, procedures, or environment blocks.

Example

```

HEADER "Instruction CASE", "", ""
...
MESSAGE message_t: response
SCENARIO One
...
    CALL mbx_send_message(&id,&message) @ err_ok
    DEF_MESSAGE response, EV={}
    WAITTIL (MATCHING(response), WTIME == 10)
    -- Checking the just received event type
    CASE (response.type)
        WHEN ACK =>
            CALL mbx_send_message(&id,&message) @ err_ok
        WHEN DATA =>
            CALL mbx_send_message(&id,&ack) @ err_ok

```

```

        WHEN NEG_ACK =>
            CALL mbx_send_message(&id,&error) @ err_ok
        WHEN OTHERS => ERROR
    END CASE
END SCENARIO

```

Native C

CALL Instruction

The CALL instruction lets you call functions or methods in a test script and to check return values of functions or methods.

For the following example, you must pre-declare the **param1**, **param2**, **param4**, and **return_param** variables in the test script, using native language.

```

CALL function ( )
-- indicates that the return parameter is neither checked
nor stored in a variable.
CALL function ( ) @ "abc"
-- indicates that the return parameter to the function must
be compared with the string "abc", but its value is not
stored in a variable.
CALL function ( ) @@return_param
-- indicates that the return parameter is not checked, but
is stored in the variable return_param.
CALL function ( ) @ 25 @return_param
-- indicates that the return parameter is checked against 25
and is stored in the variable return_param.

```

Using Native Language

In some cases, it can be necessary to include portions of C native code inside a **.pts** test script for one the following reasons:

- To define native variables to control the flow of a scenario
- To insert native code into a scenario

To use native C declarations in the test script, start the declarations with a **#** character:

```

#int i;
#char *foo;

```

Declarations must be placed outside of System Testing Language blocks or at the beginning of scenarios and procedures.

To use native C code in the test script, start instructions with a # character:

Start native code with the @ symbol.

```
@for(i=0; i++; i<100) func(i);  
@foo(a, &b, c);
```

You can add native code either inside or outside of C and Ada Test Script Language blocks.

Instances

Instance Declaration

The **DECLARE_INSTANCE** instruction lets you declare the set of the instances included in the test script.

Note Each instance behavior will be translated into different Virtual Testers executed within a process or a thread.

The **DECLARE_INSTANCE** instruction must be located before the top-level scenario.

The instance declaration can be done by one or several **DECLARE_INSTANCE** instructions. They must appear in the test script in such a way that no **INSTANCE** block containing global declarations uses an instance that has not been previously declared.

Example

```
HEADER "Multi-server / Multi-client example", "1.0", ""  
DECLARE_INSTANCE server1, server2  
...  
DECLARE_INSTANCE client1, client2, client3  
...  
SCENARIO Principal  
...
```

Instance Synchronization

The **RENDEZVOUS** statement, provides a way to synchronize Virtual Testers to each instance.

When a scenario is executed, the **RENDEZVOUS** instruction stops the execution until all Virtual Testers sharing this synchronization point (the identifier) have reached this statement.

When all Virtual Testers have met the rendezvous, the scenario resumes.

```
SCENARIO first_scenario
FAMILY nominal
  -- Synchronization point shared by both Instances
  RENDEZVOUS sync01
  INSTANCE JUPITER:
  RENDEZVOUS sync02
  . . .
  END INSTANCE
  INSTANCE SATURN:
  RENDEZVOUS sync02
  . . .
  END INSTANCE
END SCENARIO
```

Synchronization can be shared with other parts of the test bench such as in-house Virtual Testers, specific feature , and so on. This can be done easily by linking these pieces with the current Target Deployment Port.

Then, to define a synchronization point, you must make a call to the following function:

```
atl_rdv("sync01");
```

This synchronization point matches the following instruction used in a test script:

```
RENDEZVOUS sync01
```

Example

The following test script is based on the example developed in the Event Management section. The script provides an example of the usefulness of

instances for describing several applications in a same test script.

```

HEADER "SystemTest Instance-including Scenario Example",
"1.0", ""
DECLARE_INSTANCE JUPITER, SATURN
COMMTYPE appl_comm IS appl_id_t
MESSAGE message_t: message, data, my_ack, neg_ack
CHANNEL appl_comm: appl_ch
#appl_id_t id;
#int errcode;
PROCSEND message_t: msg ON appl_comm: id
CALL mbx_send_message( &id, &msg ) @ err_ok
END PROCSEND
CALLBACK message_t: msg ON appl_comm: id
    CALL mbx_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
SCENARIO first_scenario
FAMILY nominal
    COMMENT Initialize, register, send data
    COMMENT wait acknowledgement, unregister and release
    CALL mbx_init(&id) @ err_ok @ errcode
    ADD_ID(appl_ch,id)
    INSTANCE JUPITER:
    VAR id.applname, INIT="JUPITER"
END INSTANCE
    INSTANCE SATURN:
    VAR id.applname, INIT="SATURN"
    END INSTANCE
    CALL mbx_register(&id) @ err_ok @ errcode
    COMMENT Synchronization of both instances
    RENDEZVOUS start_RDV
    INSTANCE JUPITER:
    VAR message, INIT={type=>DATA,num=>id.s_id,
&      applname=>"SATURN",
&      userdata=>"Hello Saturn!"}
    SEND( message , appl_ch )
    DEF_MESSAGE my_ack, EV={type=>ACK}
    WAITTIL (MATCHING(my_ack), WTIME==300)
    DEF_MESSAGE data, EV={type=>DATA}
    WAITTIL (MATCHING(data), WTIME==1000)
    END INSTANCE
    INSTANCE SATURN:
    DEF_MESSAGE data, EV={type=>DATA}

```

```

    WAITTIL (MATCHING(data), WTIME==1000)
    VAR message, INIT={type=>DATA,num=>id.s_id,
&      applname=>"JUPITER",
&      userdata=>"Fine, Jupiter!"}
    SEND( message , appl_ch )
    DEF MESSAGE my_ack, EV={type=>ACK}
    WAITTIL (MATCHING(my_ack), WTIME==300)
    END INSTANCE
    CALL mbx_unregister(&id) @ err_ok @ errcode
    CLEAR_ID(appl_ch)
    CALL mbx_end(&id) @ err_ok @ errcode
    COMMENT Termination Synchronization
    RENDEZVOUS term_RDV
END SCENARIO

```

The scenario describes the behavior of two applications (**JUPITER** and **SATURN**) exchanging messages by using a communications stack.

Some needed resources are allocated and a connection is established with the communication stack (**mbx_init**). This connection is made known by the Virtual Tester with the **ADD_ID** instruction. Note that this is a common part to both instances.

Then, the two applications register (**mbx_register**) onto the stack by giving their application name (**JUPITER** or **SATURN**). These operations are specific to each instance, which is why these operations are done in two separate instance blocks.

The application **JUPITER** sends the message "Hello Saturn!" to the **SATURN** application (through the communication stack) which is supposed to have set itself in a message waiting state (**WAITTIL (MATCHING(data), ...)**).

Once the message has been sent, **JUPITER** waits for an acknowledgment from the communication stack (**WAITTIL(my_ack),...**). Then, it waits for the response of **SATURN** (**WAITTIL (MATCHING(data),...)**) which answers by the message "Fine, Jupiter!" (**SEND(message , appl_ch)**). These operations are specific to each instance.

Finally, the applications unregister themselves and free the allocated

resources in the last part, which is common to both instances.

Instances

In a distributed environment, you can merge the description of several entities, Virtual Testers, in a unique test script. This is possible through the concept of interaction instances, as defined in UML.

Hence, you create Virtual Testers, all based on a same test script, with distinct behaviors such as a client and a server or both.

The use of instances in a test script must be split into two parts, as follows:

- The declaration of the instances used in test script
- The description of the instances by specific blocks containing declarations or instructions.

Environments

When creating a test script, you typically write several test scenarios. These scenarios are likely to require the same resources to be deployed and then freed. You can avoid writing a series of scenarios containing similar code by factorizing elements of the scenario.

To resolve these problems and leverage your test script writing, you can define environments introduced by the keywords **INITIALIZATION**, **TERMINATION**, and **EXCEPTION**.

This section describes

- Error Handling
- Exception Environment (Error Recovery Block)
- Initialization Environment
- Termination Environment

Error Handling

The ERROR Statement

The **ERROR** instruction lets you interrupt execution of a scenario where an error occurs and continue on to the next scenario at the same level.

ERROR instructions follow these rules:

- **ERROR** instructions can be located in scenarios, in procedures, or in environment blocks.
- If an **ERROR** instruction is encountered in an **INITIALIZATION** block, the Virtual Tester exits with an error from the set of scenarios at the same level.

Note In debug mode, the behavior of **ERROR** instructions is different (see Debugging Virtual Testers).

The following is an example of an **ERROR** instruction:

```
HEADER "Instruction ERROR", "1.0", "1.0"
#int IdConnection;
SCENARIO Main
  COMMENT connection
  CALL socket(AF_UNIX, SOCK_STREAM, 0)@@IdConnection
  IF (IdConnection == -1) THEN
    ERROR
  END IF
END SCENARIO
```

The EXIT Statement

The **EXIT** instruction lets you interrupt execution of a Virtual Tester. Subsequent scenarios are not executed.

EXIT instructions follow these rules:

- **EXIT** instructions can be located in scenarios, procedures, or environment blocks.

- If an **EXIT** instruction is encountered, the **EXCEPTION** blocks are not executed.

The following is an example of an **EXIT** instruction:

```

HEADER "Instruction EXIT", "1.0", "1.0"
#int IdConnection;
SCENARIO Main
    COMMENT connection
    CALL socket(AF_UNIX, SOCK_STREAM, 0)@@IdConnection
    IF (IdConnection == -1) THEN
        EXIT
    END IF
END SCENARIO

```

Exception Environment (Error Recovery Block)

A test script is composed of a hierarchy of scenarios. An exception environment can be defined at a given scenario level.

When an error occurs in a scenario all exception blocks at the same level or above are executed sequentially.

The syntax for exception environments can take two different forms, as follows:

- **A block:** This begins with the keyword **EXCEPTION** and ends with the sequence **END EXCEPTION**. A termination block can contain any instruction.
- **A procedure call:** This begins with the keyword **EXCEPTION** followed by the name of the procedure and, where appropriate, its arguments.

Example

In the following example, the highest level of the test script is made up of two scenarios called first and second. The exception environment that precedes them is executed once if scenario premier finished with an error, and once if scenario second finishes with an error.

```

HEADER "Validation", "01a", "01a"

```

```

PROC Unload_mem()
...
END PROC
EXCEPTION Unload_mem()
SCENARIO first
...
END SCENARIO
SCENARIO second
EXCEPTION
...
END EXCEPTION
SCENARIO level2_1
FAMILY nominal, structural
...
END SCENARIO
SCENARIO level2_2
FAMILY nominal, structural
...
END SCENARIO
END SCENARIO

```

Scenario second is made up of two sub-scenarios, level2_1 and level2_2. The second exception environment is executed after incorrect execution of scenarios level2_1 and level2_2. The highest-level exception environment is not re-executed if scenarios level2_1 and level2_2 finish with an error.

Only one exception environment can appear at a given scenario level.

An exception environment can appear among scenarios at the same level. It does not have to be placed before a set of scenarios at the same level.

In a test report, the execution of an exception environment is shown even if you decided not to trace the execution.

Initialization Environment

A test script is composed of scenarios in a tree structure. An initialization environment can be defined at a given scenario level.

This initialization environment is executed before each scenario at the same level.

The syntax for initialization environments can take two different forms, as follows:

- **A block:** This begins with the keyword **INITIALIZATION** and ends with the sequence **END INITIALIZATION**. An initialization block can contain any instruction.
- **A procedure call:** This begins with the keyword **INITIALIZATION** followed by the name of the procedure and, where appropriate, its arguments.

Example

In the following example, the highest level of the test script is made up of two scenarios called *first* and *second*. The initialization environment that precedes them is executed twice: once before scenario *first* is executed and once before scenario *second* is executed.

```
HEADER "Validation", "01a", "01a"
PROC Load_mem()
...
END PROC
INITIALIZATION Load_mem()
SCENARIO first
...
END SCENARIO
SCENARIO second
INITIALIZATION
END INITIALIZATION
SCENARIO level2_1
FAMILY nominal, structural
...
END SCENARIO
SCENARIO level2_2
FAMILY nominal, structural
...
END SCENARIO
END SCENARIO
```

Scenario *second* is made up of two sub-scenarios, *level2_1* and *level2_2*. The second initialization environment is executed before scenarios *level2_1* and *level2_2* are executed. The highest-level initialization environment is not re-

executed between scenarios *level2_1* and *level2_2*.

Only one initialization environment can appear at a given scenario level.

An initialization environment can appear among scenarios at the same level. The initialization environment does not have to be placed before a set of scenarios at the same level.

In a test report, the execution of an initialization environment is shown beginning with the word **INITIALIZATION** and ending with the words **END INITIALIZATION**.

Termination Environment

A test script is composed of scenarios in a tree structure. A termination environment can be defined at a given scenario level.

This termination environment is executed at the end of every scenario at the same level, provided that each scenario finished without any errors.

The syntax for termination environments can take two different forms, as follows:

- **A block:** This begins with the keyword **TERMINATION** and ends with the sequence **END TERMINATION**. A termination block can contain any instruction.
- **A procedure call:** This begins with the keyword **TERMINATION** followed by the name of the procedure and, where appropriate, its arguments.

Example

In the previous example, the highest level of the test script is made up of two scenarios called first and second. The termination environment that precedes them is executed twice:

- once after scenario first is executed correctly
- once after scenario second is executed correctly

```

HEADER "Validation", "01a", "01a"
PROC Unload_mem()
...
END PROC
TERMINATION Unload_mem()
SCENARIO first
...
END SCENARIO
SCENARIO second
TERMINATION
...
END TERMINATION
SCENARIO level2_1
FAMILY nominal, structural
...
END SCENARIO
SCENARIO level2_2
FAMILY nominal, structural
...
END SCENARIO
END SCENARIO

```

Scenario *second* is made up of two sub-scenarios, *level2_1* and *level2_2*. The second termination environment is executed after the correct execution of scenarios *level2_1* and *level2_2*. The highest-level termination environment is not re-executed between scenarios *level2_1* and *level2_2*.

Only one termination environment can appear at a given scenario level.

A termination environment can appear among scenarios at the same level. The termination environment does not have to be placed before a set of scenarios at the same level.

In a test report, the execution of a termination environment is shown beginning with the word **TERMINATION** and ending with the words **END TERMINATION**.

Time Management

In some cases, you will need information about execution time within a test script.

The following instructions provide a way to dump timing data, define a timer, clear a timer, get the value of a timer, and temporarily suspend test script execution:

- **TIME** Instruction
- **TIMER** Instruction
- **RESET** Instruction
- **PRINT** Instruction
- **PAUSE** Instruction

TIME Instruction

The **TIME** instruction returns the current value of a timer. You must use a C expression or scripting instruction (**IF**, **PRINT**, and so on).

Before using **TIME**, you must declare the timer with the **TIMER** instruction.

Example

```
HEADER "Socket validation", "1.0", "beta"
TIMER globalTime
PROC first
TIMER firstProc
...
PRINT globalTimeValue, TIME (globalTime)
END PROC
SCENARIO second
SCENARIO level2
TIMER level2Scn
...
PRINT level2ScnValue, TIME (level2Scn)
END SCENARIO
END SCENARIO
```


TIMER Instruction

The **TIMER** instruction declares a timer in the test script.

You may declare a timer in any test script block: global, initialization, termination, exception, procedure, or scenario.

The timer lasts as long as the block in which the timer is defined. This means that a timer defined in the global block can be used until the end of the test script.

You may define multiple timers in the same test script. The timer starts immediately after its declaration.

The unit of the timer unit is defined during execution of the application, with the **WAITTIL** and **WTIME** instructions.

Example

```
HEADER "Socket validation", "1.0", "beta"
TIMER globalTime
PROC first
TIMER firstProc
...
END PROC
SCENARIO second
SCENARIO level2
TIMER level2Scn
...
END SCENARIO
END SCENARIO
```

RESET Instruction

The **RESET** instruction lets you reset a timer to zero.

The timer restarts immediately when the **RESET** statement is encountered.

A timer must be declared before using **RESET**.

Example

```
HEADER "Socket validation", "1.0", "beta"  
TIMER globalTime  
PROC first  
TIMER firstProc  
RESET globalTime  
...  
END PROC  
SCENARIO second  
SCENARIO level2  
TIMER level2Scn  
...  
RESET level2Scn  
END SCENARIO  
END SCENARIO
```

PRINT Instruction

You can print the result of an expression in a performance report by using the **PRINT** statement. The **PRINT** instruction prints an identifier before the expression.

Example

```
HEADER "Socket validation", "1.0", "beta"  
#long globalTime = 45;  
SCENARIO first  
PRINT timeValue, globalTime  
END SCENARIO  
SCENARIO second  
SCENARIO level2  
PRINT time2Value, globalTime*10+5  
...  
END SCENARIO  
END SCENARIO
```

PAUSE Instruction

The **PAUSE** instruction lets you temporarily stop test script execution for a given period.

The unit of the **PAUSE** instruction is defined during execution of the application, with the **WAITTIL** and **WTIME** instructions.

Example

```
HEADER "Socket validation", "1.0", "beta"
#long time = 20;
PROC first
PAUSE 10
...
END PROC
SCENARIO second
SCENARIO level2
PAUSE time*10
...
END SCENARIO
END SCENARIO
```

Event Management

Event management helps you describe communication between the Virtual Tester and the system under test.

Many different means of communication allow your systems to talk with each other. At the software application level, a communication type is identified by a set of services provided by specific functions.

For example, a UNIX system provides several means of communication between processes, such as *named pipes*, *message queues*, *BSD sockets*, or *streams*. You address each communication type with a specific function.

Furthermore, each communication type has its own data type to identify the application you are sending messages to. This type is often an *integer* (message queues, BSD sockets, ...), but sometimes a *structure* type.

Data exchanged this way must be interpreted by all communicating applications. For this reason, each type of exchanged data must be well identified and well known. By providing the type of exchanged data to the Virtual Tester, it will be able to automatically print and check the incoming messages.

- Basic Declarations
- Sending Messages

- Receiving Messages
- Messages and Data Management
- Communication Between Virtual Testers

Basic Declarations

COMMTYPE Instruction

For each communication type, there is a specific data type that identifies the application you are sending messages to. In a test script, the **COMMTYPE** instruction is used to identify clearly this data type, and then, the communication type.

The data type has to be defined by a C *typedef* or a C++ object.

On UNIX systems, the data type for the BSD sockets is an integer. The **COMMTYPE** instruction is used as follows:

```
#typedef int bsd_socket_id_t;
COMMTYPE ux_bsd_socket IS bsd_socket_id_t
```

The stack defines the data type **appl_id_t**. Therefore, the following figure defines a new communication type called **appl_comm**:

```
COMMTYPE appl_comm IS appl_id_t
MESSAGE Instruction
```

The **MESSAGE** instruction identifies the type of the data exchanged between applications. It also defines a set of reference messages.

The type of the messages exchanged between applications using our stack is **message_t**.

The following instruction also declares three reference messages:

```
MESSAGE message_t: ack, neg_ack, data
CHANNEL Instruction
```

The **CHANNEL** instruction is used to declare a communication channel on

a specific communication type. Thanks to channels of communication, the user can easily manage a large number of opened connections.

```
CHANNEL appl_comm: appl_channel_1, appl_channel_2
ADD_ID Instruction
```

A communication channel is a logical medium of communication that multiplexes several opened connections of the same type between the Virtual Tester and applications under test. When opening a new connection, it has to be linked to a communication channel, so that the Virtual Tester knows about this new connection.

```
CALL mbx_init( &id ) @ err_ok @ errcode
ADD_ID (appl_channel, id)
```

the function call to **mbx_init** opens a connection between the Virtual Tester and the stack. This connection is identified by the value of `id` after the call. The **ADD_ID** instruction add this new connection to the channel **appl_channel**.

Sending Messages

PROCSEND Instruction

Event management provides a mechanism to send messages. This mechanism needs the definition of a message sending procedure or **PROCSEND** for each couple communication type, message type.

The **PROCSEND** instruction is then called automatically to sends a message to the stack.

In the following example, **msg** is a **message_t** typed input formal parameter specifying the message to send. The input formal parameter `stack` is used to know where to send a message on the communication type **appl_comm**.

```
PROCSEND message_t: msg ON appl_comm: id
CALL mbx_send_message ( &id, &msg ) @ err_ok
END PROCSEND
```

The sending is done by the API function call to **mbx_send_message**. The

return code is treated to decide whether the message was correctly sent. Another value than **err_ok** means that an error occurred during the sending.

VAR Instruction

The instruction **VAR** allows you to initialize messages declared using **MESSAGE** instructions. This message may also be initialized by any other C or C++ function or method:

```
VAR ack, INIT= { type => ACK }
VAR data, INIT= {
& type => DATA,
& applname => "SATURN",
& userdata => "hello world !" }
```

To learn all the nuts and bolts of the **DEF_MESSAGE** Instruction, see the Messages and Data Management chapter.

SEND Instruction

This instruction allows you to invoke a message sending on one communication channel .

It has two arguments:

- the message to send,
- the communication channel where the message should be sent.

The send instruction is as follows:

```
SEND ( message , appl_ch )
```

In the previous figure, the **SEND** instruction allows the test program to send a message on a known connection (see the **ADD_ID** instruction). If an error occurs during the sending of the message, the **SEND** exits with an error. The scenario execution is then interrupted.

Example

The following test script describes a simple use of our stack. First of all, some resources are allocated and a connection is established with the communication stack (**mbx_init**). This connection is made known by the Virtual Tester with the **ADD_ID** instruction. Then, the Virtual Tester registers (**mbx_register**) onto the stack by giving its application name (**JUPITER**). The Virtual Tester sends a message to an application under test (**SATURN**). Finally, the Virtual Testers unregisters itself (**mbx_unregister**) and frees the allocated resources (**mbx_end**)

```
HEADER "SystemTest 1st example: sending a message","1.0",""
COMMTYPE appl_comm IS appl_id_t
MESSAGE message_t: message, ack, data, neg_ack
CHANNEL appl_comm: appl_ch
#appl_id_t id;
#int errcode;
PROSEND message_t: msg ON appl_comm: id
    CALL mbx_send_message ( &id, &msg) @ err_ok
END PROSEND
SCENARIO first_scenario
FAMILY nominal
    COMMENT Initialize, register, send data
    COMMENT wait acknowledgement, unregister and release
    CALL mbx_init(&id) @ err_ok @ errcode
    ADD_ID(appl_ch,id)
    VAR id.applname, INIT="JUPITER"
    CALL mbx_register(&id) @ err_ok @ errcode
    VAR message, INIT={
& type=>DATA,
& applname=>"SATURN",
& userdata=>"hello Saturn!"}
    SEND ( message, appl_ch )
    CALL mbx_unregister(&id) @ err_ok @ errcode
    CLEAR_ID(appl_ch)
    CALL mbx_end(&id) @ err_ok @ errcode
END SCENARIO
```

Receiving Messages

CALLBACK Instruction

The event management provides an asynchronous mechanism to receive messages. This mechanism needs the definition of a callback for each couple

communication type, message type.

A callback should do a non-blocking read for a specific message type on a specific communication type.

The **MESSAGE_DATE** instruction lets you mark the right moment of the reception of messages. The **NO_MESSAGE** instruction exits from the callback and indicates that no message has been read.

The callback to receive messages from our stack is as follows:

```
CALLBACK message_t: msg ON appl_comm: id
  CALL mbx_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
```

In this example, **msg** is an output formal parameter of the callback. Its type is **message_t**. The input formal parameter **id** is used to know where to read a message on the communication type **appl_comm**.

The reading is done by the function call to **mbx_get_message**. The return code is stored into the variable **errcode**. The value **err_empty** for the return code means that no message has been read. Another value than **err_ok** or **err_empty** means that an error occurred during the reading. The **NO_MESSAGE** and **ERROR** instructions make the callback to return.

DEF_MESSAGE Instruction

The **DEF_MESSAGE** instruction defines the values of a reference message declared with the **MESSAGE** instruction. A reference message is a message expected by the virtual tester from an application under test.

```
DEF_MESSAGE ack, EV= { type => ACK }
DEF_MESSAGE data, EV= {
```



```
& type => DATA,  
& applname => "SATURN",  
& userdata => "hello world !" }
```

To learn all the nuts and bolts of the **DEF_MESSAGE** Instruction, see the Messages and Data Management chapter.

WAITTIL Instruction

The **WAITTIL** instruction allows waiting for events or conditions. **WAITTIL** is made of two Boolean expressions: an expected condition, and a failure condition. The instruction blocks until one of the two expressions becomes true.

In the following example, the **WAITTIL** instruction receives all the messages sent to the Virtual Tester on a known connection. As soon as a received message matches the reference message **ack**, the **WAITTIL** exits normally. Otherwise, if any message matching the reference message **ack** is received during 300 units of time, the **WAITTIL** exits with an error (the time unit is configurable in the Target Deployment Port according to the execution target). The scenario execution is interrupted.

```
WAITTIL ( MATCHING(ack), WTIME == 300)
```

In the example given above, the status of the reference event variable **ack** is tested using the function **MATCHING()** which identifies if the last incoming event corresponds to the content of the variable **ack**. **WTIME** is a reserved keyword valuated with the time expired since the beginning of the **WAITTIL** instruction.

The **WAITTIL** Boolean conditions are described using C or C++ conditions including operators to manipulate events:

- **MATCHING:** does the last event match the specified reference event?
- **MATCHED:** did the Virtual Tester receive an event matching the specified event?
- **NOMATCHING:** is the last event different from the specified

reference event?

- **NOMATCHED:** did the Virtual Tester receive an event different from the specified event?

The different combinations of these operators allow an easy an extensive definition of event sequences:

```
-- I expect evt1 on channel1 before my_timeout is reached
WAITTIL (MATCHING(evt1, channel1), WTIME>my_timeout)
-- I expect evt1 then evt2 on one channel before my_timeout
is reached
WAITTIL (MATCHED(evt1)&& MATCHING(evt2), WTIME>my_timeout)
-- I expect to receive nothing during my_time
WAITTIL (WTIME>my_time, MATCHING(empty_evt))
-- I expect evtA or evtB before my_timeout is reached
WAITTIL (MATCHING(evtA) || MATCHING(evtB), WTIME>my_timeout)
*
```

After the **WAITTIL** instruction, the value of these operators is available until the next call to **WAITTIL**.

Example: Sending and Receiving Messages

The following test script describes a simple use of our stack. First of all, some resources are allocated and a connection is established with the communication stack (**mbx_init**). This connection is made known by the Virtual Tester with the **ADD_ID** instruction. Then, the Virtual Tester registers (**mbx_register**) onto the stack giving its application name (**JUPITER**).

The Virtual Tester sends a message to an application under test (**SATURN**), and waits for the acknowledgment sent back by the stack with the **WAITTIL** instructions. Finally, the Virtual Tester unregisters (**mbx_unregister**) and frees the allocated resources (**mbx_end**).

```
HEADER "SystemTest 1st example: sending & receiving a
message", "1.0", ""
COMMTYPE appl_comm IS appl_id_t
MESSAGE message_t: message, ack, data, neg_ack
CHANNEL appl_comm: appl_ch
#appl_id_t id;
```

```

#int errcode;
PROCSEND message_t: msg ON appl_comm: id
CALL mbx_send_message ( &id, &msg) @ err_ok
END PROCSEND
CALLBACK message_t: msg ON appl_comm: id
CALL mbx_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
SCENARIO first_scenario
FAMILY nominal
COMMENT Initialize, register, send data
COMMENT wait acknowledgement, unregister and release
CALL mbx_init(&id) @ err_ok @ errcode
ADD_ID(appl_ch,id)
VAR id.applname, INIT="JUPITER"
CALL mbx_register(&id) @ err_ok @ errcode
VAR message, INIT={
& type=>DATA,
& applname=>"SATURN",
& userdata=>"hello Saturn!"}
SEND ( message, appl_ch )
COMMENT Negative acknowledgment expected
COMMENT (Saturn is not running !)
DEF MESSAGE ack, EV={type=>ACK}
WAITTIL (MATCHING(ack), WTIME==10)
CALL mbx_unregister(&id) @ err_ok @ errcode
CLEAR_ID(appl_ch)
CALL mbx_end(&id) @ err_ok @ errcode
END SCENARIO

```

Messages and Data Management

The instruction VAR allows you to initialize and check the contents of simple or complex variables.

The process of initializing or checking variables is performed independently by the following two sub-instructions:

```
VAR <variable> , INIT = <init_expr>
```

or

```
VAR <variable> , EV = <expect_expr>
```

This instruction allows you to initialize and check the contents of structured variables, such as messages.

The field <variable> represents a variable or part of a structured variable.

<init_expr> and <expect_expr> let you describe the contents of structured variables using a simple syntax.

To describe a sequence of fields at the same level in a structured variable, you enclose the sequence in braces '}' or brackets '[]' and separate the fields with a comma ','.

You can reference members of a structured variable in the following ways:

- Reference by name
- Reference by position

You cannot however mix both methods.

The System Testing report does not show **VAR** instructions relating to initializations. Only **VAR** instructions relating to content checks on variables or messages are recorded in the test report.

The **DEF_MESSAGE** instruction allows you to define reference messages using the **DEF_MESSAGE** instruction, using exactly the same syntax. The following examples are presented using the **VAR** instruction, but are also applicable to **DEF_MESSAGE**.

The report does not show **DEF_MESSAGE** instruction as they appear in the test script, but only when they are used within a **WAITTIL** instruction.

Reference by Name

You can describe the contents of a structure by naming each field in the

structure. This is very useful if you do not know the order of the fields in the declaration of the structure.

When referencing by name, a parameter is described by the name of the field in the structure followed by the arrow symbol (\Rightarrow) and the initialization or checking expression.

```
#typedef struct
# {
#   int Integer;
#   char String [ 15 ];
#   float Real;
# } block;
# block variable;
VAR variable, INIT={Real=>2.0, Integer=>26, String=>"foo"}
```

You can omit the specification of structure elements by name if you know the order of the fields within the structure. For the block type defined above, you can write the following **VAR** statement:

```
VAR variable, INIT={ 26, "foo", 2.0 }
```

Reference by Position

You can describe the contents of an array by giving the position of elements within the array.

When referencing by position, define a parameter by giving the position of the field in the array followed by the arrow symbol (\Rightarrow) and the initialization or checking expression.

Note that numbering begins at zero.

```
#int array[5];
VAR array, EV=[4=>5, 1=>12, 2=>-18, 5=>15-26, 3=>0, 0=>123]
```

You can use ranges of positions when referencing by position. These ranges are specified by two bounds separated by the symbol double full-stop (\dots).

```
#typedef int matrix[3][150];
VAR matrix, EV= [
& 2=>[0..99=>1, 100..149=>2],
& 0=>[99..0=>2, 100..149=>1],
```

```
& 1=>[0..80=>-1, 81..149=>0]]
```

Note that the bounds of an interval can be reversed.

When referencing by position, you must reference an entire sequence at a given level.

Partial Initialization and Checks

With a **VAR** instruction, you can partially initialize and check a structured variable.

```
#float array[10];  
VAR array, INIT=[5..7=>2.1]
```

The array elements **5**, **6** and **7** are initialized to **2.1**. Other elements are not initialized.

Multi-dimension Initialization and Checks

With a **VAR** instruction, you can initialize and check multi-dimension variables with judicious use of bracket '[' and brace '}' separators.

The separators delimit the description of a structured variable to a given dimension. The absence of separators at a given level indicates that the initialization or checking value is valid for all the sub-dimensions of the variable.

In the following example:

- **Ex. 1:** The set of 300 integer values of the matrix variable are initialized to zero.
- **Ex. 2:** The 100 integer values contained in **matrix[0]** are initialized to **1**, the 100 values of **matrix[1]** are initialized to **2**, and the 100 values of **matrix[2]** are initialized to **3**.
- **Ex. 3:** Only the **matrix[0][0]** is initialized to zero.

- **Ex. 4:** Only the first 100 values of **matrix[0]** are initialized to zero.

```
#int matrix[3][100];
-- -Ex. 1- Global initialization
VAR matrix, INIT=0
-- -Ex. 2- Global initialization of lines
VAR matrix, INIT=[1,2,3]
-- -Ex. 3- Initialization of only one element
VAR matrix, INIT=[0]
-- -Ex. 4- Initialization of only one line
VAR matrix, INIT=[0]
```

The following example provides a set of **VAR** instructions that are semantically identical:

```
#int matrix[3][3];
VAR matrix, EV=0
VAR matrix, EV=[0,0,0]
VAR matrix, EV=[[0,0,0],[0,0,0],[0,0,0]]
```

In the three **VAR** instructions above, all the matrix elements are checked against zero.

Array Indices

With a **VAR** instruction, you can initialize and check array elements according to their index at a given level.

The index is specified by a capital I followed by the level number. Levels begin at **1**. You can use **I1**, **I2**, **I3**, etc. as implicit variables.

```
#int matrix[3][100];
VAR matrix, EV=I1*I2
```

Each element of the above matrix is checked against the product of variables **I1** and **I2**, which indicate, respectively, a range from 0 to 2 and a range from 0 to 99. The above matrix is checked against the 3 by 100 multiplication table.

Reference by Default

You can reference the remaining set of fields in an array, structure, or object

in a **VAR** instruction. To do this, use the keyword **OTHERS**, followed by the arrow symbol =>, and an expression in C or C++.

Note: To use **OTHERS**, the remaining fields must be the same type and must be compatible with the expression following **OTHERS**.

```
#typedef struct {
# char String[25];
# int Value;
# int Value2;
# int Array[30];
#} block;
# block variable;
VAR variable, INIT=[
& String=>"chaine",
& Array=>[0..10=>0, OTHERS=>1] ,
& OTHERS=>2]
```

In the previous example, OTHERS has two functions:

- When initializing the array, the values indexed from 11 to 29 begin at 1.
- When initializing the structure, the value and value2 fields begin at 2.

Checking Pointers

With a **VAR** instruction, you may use **NIL** and **NONIL**, to check for null and non-null pointers.

```
#typedef struct {
# int a;
# float b;
#} block, *PT_block;
#PT_block addr[10];
VAR addr, EV=[0..5=>NIL, OTHERS=>NONIL]
```

In the above example, the pointers indexed from 0 to 5 of the addr array are compared with the null address. The test of the pointers indexed from 6 to 9 is correct if these pointers are different from the null address.

Checking Ranges

You may use ranges of acceptable values instead of immediate values. To

do this, use the following syntax:

```
VAR <variable>, EV=[Min..Max]
DEF_MESSAGE <variable>, EV=[Min..Max]
```

The following example demonstrates this syntax:

```
#typedef struct {
# int a;
# float b;
#} block, *PT_block;
#PT_block addr[10];
VAR addr, EV=[0..5=>{a=>[0..100]}], OTHERS=>NONIL]
```

In the previous example, the elements indexed from 0 to 5 of the **addr** array are checked with the following constraint:

a should be greater than 0 and lower than 100.

The test of the pointers indexed from 6 to 9 is correct if these pointers are different from null address

Character Strings

When you use the **VAR** instruction for character strings, you may alter it. In C, a character string can also be an array. This flexibility is retained in the **VAR** instruction.

In the following example, the first variable **String** initializes as in C (null-terminated). The second **String** initializes as an array of characters (not null-terminated).

```
#char String[15];
VAR String, INIT="abcdef"
VAR String, INIT=['a', 'b', 'c', 'd', 'e', 'f']
```

Note You must define the **VAR** instruction either as a character string or an array of characters.

Communication Between Virtual Testers

Virtual Testers can communicate between themselves with simple messages

by using the **INTERSEND** and **INTERRECV** statements. These messages can be either an *integer* or a *text string*.

For information about the **INTERSEND** and **INTERRECV** statements, please refer to the C and Ada Test Script Language section in the **Rational Test RealTime Reference Manual**.

Identifier

For message delivery purposes, each Virtual Testers carries a unique *identifier*. The virtual tester identifier is constructed with the following rules:

- If the Virtual Tester is run as an instance named *<instance>*:
<instance>_<occid>
- If the Virtual Tester is running in multi-threaded mode, with its entry point in *<function>*:
<function_name>_<occid>
- In any other case, the identifier uses the **.rio** file name:
<filename>.rio_<occid>

By default the occurrence identification number *<occid>* for each Virtual Tester is 0, but you can set different *<occid>* values in the Virtual Tester Deployment dialog box.

There must never be two Virtual Testers at the same time with the same identifier. If an **INTERSEND** message cannot be delivered because of an ambiguous identifier, the System Testing supervisor returns an error message.

Understanding System Testing for C Reports

Test reports for System Testing are displayed in Test RealTime's Report Viewer.

The test report is a hierarchical summary report of the execution of a test

node. Parts of the report that have *Passed* are displayed in green. *Failed* tests are shown in red.

Report Explorer

The Report Explorer displays each element of a test report with a *Passed* ✓, *Failed* ✗ symbol.

- Elements marked as *Failed* ✗ are either a failed test, or an element that contains at least one failed test.
- Elements marked as *Passed* ✓ are either passed tests or elements that contain only passed tests.

Test results are displayed for each instance, following the structure of the **.pts** test script.

Report Header

Each test report contains a report header with:

- The version of Test RealTime used to generate the test as well as the date of the test report generation
- The path and name of the project files used to generate the test
- The total number of test cases *Passed* and *Failed*. These statistics are calculated on the actual number of test elements listed in the sections below
- Virtual Tester information.

Main Report Sections

For each Virtual Tester execution, the report lists the details of test script execution, with time stamps

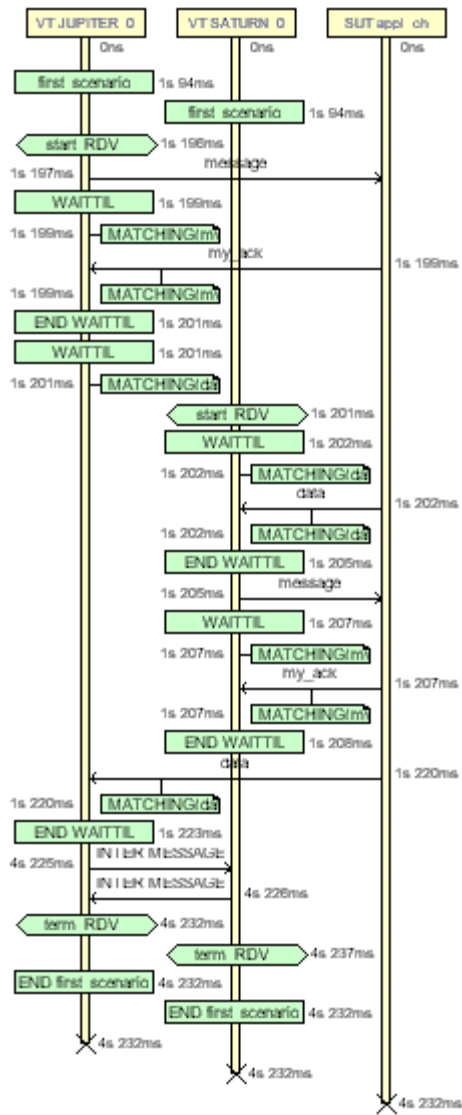
and test result tables.

- **Messages:** The report displays fields and values for each field
- **Tests Results:** For each message, the report compares initial values, expected values and obtained values

Understanding System Testing UML Sequence Diagrams

During the execution of the test, System Testing generates trace data this is used by the UML/SD Viewer. The System Testing sequence diagram uses standard UML notation to represent both System Testing results.

This is an example of a typical System Testing UML sequence diagram.



You can modify the appearance of UML sequence diagrams by changing the UML/SD Viewer Preferences.

When using System Testing with Runtime Tracing or other Test RealTime

features that generate UML sequence diagrams, all results are merged in the same sequence diagram.

You can click any element of the UML sequence diagram to open the System Testing reports at the corresponding line. Click again in the test report, and you will locate the line in the **.pts** test script.

Virtual Testers and System Under Test

The system under test (SUT) and the Virtual Testers (VT) are represented as vertical instances. Messages sent and received by the Virtual Tester are represented along the Virtual Tester lifeline.

Messages

Messages are sent and received between Virtual Tester and system instances.

Rendezvous

RENDEZVOUS statements are displayed as Synchronizations in the Virtual Tester lifeline.

Test Script Events and Errors

Test script events and errors are represented as UML actions. Only significant instructions, such as **INITIALIZATION**, **WAITTIL** blocks and test errors are represented.

By default, errors appear in red. Other events are green.

WAITTIL blocks are displayed with their start and end events. Matching conditions are represented as notes. Use the mouse cursor tool-tip to get more information about the matching conditions.

On-the-Fly Tracing

If you are using the On-the-Fly option, only the following information can be displayed in real-time during the execution of the application:

- Virtual Tester and system under test
- Messages
- Rendezvous
- Test script blocks

Advanced System Testing for C

Trace Probes

The Probe feature of Test RealTime allows you to manually add special probe C macros at specific points in the source code under test, in order to trace messages.

Adding trace probes to the application produces a binary which is functionally identical to the original, but which generates extra message tracing results 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, including message content and a time stamp. Probe trace results can **then be processed and displayed** as .tdf dynamic trace files in the UML/SD Viewer.

The use of C macros offers extreme flexibility. For example, when delivering the final application, you can leave the macros in the final source and simply provide an empty definition.

Using Probe Macros

Before adding probe macros to your source code, add the following **#include** statement to each source file that is to contain 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.

The following probe macros are available:

```
atl_dump_trace()  
atl_end_trace()  
atl_recv_trace()  
atl_select_trace()  
atl_send_trace()  
atl_start_trace()  
atl_format_trace()
```

Please refer to the **Probe Macros** section in the **Test RealTime Reference Manual** for a complete definition of each probe macro.

To activate the trace probe feature:

- 1 In the **Project Browser**, select the application or test node on which you want to use the feature.
- 2 Click **Settings** and open the **Probe Control Settings** box.
- 3 Set **Probe Enable** to **Yes**, select the correct output mode in **Probe Settings** and click **OK**.
- 4 Edit the source code under test to add the trace probe macros, including the **#include** line.
- 5 Set up your trace probes within your application source files.

To read the trace probe output:

- 1 From the **File** menu, select **Open** and **File**.

- 2 In the file selector, select **Trace Files (*.tsf, *.tdf)** and select the **.tsf** and **.tdf** files produced after the execution of the application under test.
- 3 Click **OK**.

Generated Test Script

When a probed application is executed, System Testing for C produces a **.pts** test script based on probe activity.

You can edit and reuse this script in further tests to replay the exact same data exchanges in a System Testing for C test node.

Custom Probe Output

By default, the message traces are written to the **.rio** System Testing output file for C. However, the Probe capability can send traces to a temporary buffer.

- **DEFAULT:** In this mode, the message traces are written directly to the **.rio** System Testing output file for C
- **FIFO:** Select this to direct traces to a temporary *first-in first-out* memory buffer before writing to the **.rio** file
- **FILE:** Select this to direct traces to a temporary file before writing to the **.rio** file
- **USER:** Uses a method, described in a user-defined **probecst.c** file that must be added to the application node
- **IGNORE:** Use this setting to ignore trace macros.

In custom mode, when **FIFO**, **FILE** or **USER** are selected in the **Probe** area of the Test Script Compiler Settings, the traces must be flushed to the **.rio** file with an **atl_dump_trace** macro placed in the source code.

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 need to modify this file to adapt the probe mechanism to your application and platform.

On-the-Fly Tracing

The System Testing for C on-the-fly tracing capability allows you to monitor the Virtual Testers during the test execution in a UML sequence diagram. Information provided by dynamic tracking includes:

- Beginning and end of scenarios
- Rendezvous
- Sent and received messages
- Inter-tester messages (only received messages)
- Beginning and end of termination, initialization and exception blocks
- End of Testers

On-the-fly tracing output is displayed in the UML/SD Viewer in real-time. You can click any item in the sequence diagram to instantly highlight the corresponding test script line in the Text Editor window.

To activate System Testing dynamic tracking, you must:

- select **Display using on-the-fly mode** in the System Testing Report Generator Settings for the System Testing test node
- select **Enable** for On-the-Fly Tracing in the System Testing Target Deployment Port Settings for the test node or for each particular Virtual Tester node.
- ensure that the **Authorize connections** option is selected in the General Preferences.

Graphical User Interface

4

The graphical user interface (GUI) provides an integrated test environment designed to act as a single, unified work space for all automated testing and runtime analysis activities.

This section describes the features and capabilities included within the GUI that are designed to make your testing effort a lot more manageable.

GUI Philosophy

In addition to acting as an interface with your usual development tools, the GUI provides navigation facilities, allowing natural hypertext linkage between test and analysis reports, UML sequence diagrams and source code. For example:

- You can click any element of a test report to highlight the corresponding test script line in the embedded text editor.
- You can click any element of a runtime analysis report to highlight and edit the corresponding item in your application source code
- You can click a filename in the output window to open the file in the Text Editor

In addition, the GUI provides easy-to-use Activity Wizards to guide you through the creation of your project components.

Discovering the GUI

When you launch the Graphical User Interface (GUI), you are first greeted with the Start Page and a series of windows. Click the elements below to learn how to use them:

- The Start Page is a convenient starting point when you launch the GUI
- The Project Explorer is where you create, develop and execute your project nodes
- The Properties Window provides information about node properties
- The Output Window displays the output of command line tools and compilers
- The Standard Toolbars provide quick and convenient access to the most commonly used features
- The Report Explorer allows you to navigate through analysis reports

GUI Components and Tools

In addition to these main windows, the product GUI provides a comprehensive set of tools and components that make it an efficient and customizable development environment.

- The Text Editor is a full-featured editor for source code
- The Tools menu is a convenient way of integrating any command-line tool into the GUI
- The Test Process Monitor provides ongoing activity statistics and metrics
- The Report Viewer displays runtime analysis reports
- The UML/SD Viewer displays UML sequence diagrams provided by Runtime Tracing feature.

Start Page

When you launch the graphical user interface, the first element that appears is the Test RealTime Start Page.

The Start Page is the central location of the application. From here, you can create a new project, start a new activity and navigate through existing project reports.

The Start Page contains the following sections:

- **Get Started:** this section lists your recent projects as well as a series of example projects provided with the product.
- **Activities:** this section displays a series of new activities. Click a new activity to launch the corresponding activity wizard.

Note A project must be open before selecting a new activity.

Output Window

The Output Window displays messages issued by product components or custom features.

The first tab, labelled **Build**, is the standard output for messages and errors. Other tabs are specific to the built-in features of the product or any user defined tool that you may have added.

To switch from one console window to another, click the corresponding tab. When any of the Output Window tabs receives a message, that tab is automatically activated.

When a console message contains a filename, double-click the line to open the file in the Text Editor. Similarly when a test report appears in the Output Window, double-click the line to view the report.

Output Window Actions

Right-click the Output Window to bring up a pop-up menu with the following options:

- **Edit Selected File:** Opens the editor with the currently selected filename.
- **Copy:** Copies the selection to the clipboard.
- **Clear Window:** Clears the contents of the Output Window.

To hide or show the Output Window:

From the View menu, select Other Windows and Output Window.

Project Explorer

The Project Explorer allows you to navigate, construct and execute the components of your project. The Project Explorer organizes your workspace from two viewpoints:

- **Project Browser:** This tab displays your project as a tree view, as it is to be executed.
- **Asset Browser:** Source code and test script components are displayed on an object or elementary level.

To change views, select the corresponding tab in the lower section of the **Project Explorer** window.

Project Browser

The **Project Browser** displays the following hierarchy of nodes:

- **Project:** the Project Explorer's root node.
- **Test groups:** provide a way to group and organize test nodes into one or more test campaigns

- **Test nodes:** in Test RealTime only, these contain test scripts and source files:
 - **Test Scripts:** for Component Testing or System Testing
 - **Source files:** for code-under-test as well as additional source files
 - Any other test related files
- **Application nodes:** represent your application, to which you can apply SCI instrumentation for Memory Profiling, Performance Profiling, Code Coverage and Runtime Tracing.
- **External Command nodes:** these allow you to add shell command lines at any point in the Test Campaign.

After execution of a test or application node, double-click the node to open all associated available reports.

When you run a **Build** command in the **Project Browser**, the product parses and executes each node from the inside-out and from top to bottom. This means that the contents of a parent node are executed in sequence before the actual parent node.

Asset Browser

The Asset Browser displays all the files contained in your project. The product parses the files and displays individual components of your source files and test scripts, such as classes, methods, procedures, functions, units and packages.

Use the Asset Browser to easily navigate through your source files and test scripts.

In Asset Browser, you can select the type of Asset Browser in the **Sort**

Method box at the top of the **Project Explorer** window. Each view type can be more or less relevant depending on the programming language used:

- **By Files:** This view displays a classic source file and dependency structure
- **By Objects:** Primarily for C++ and Java, this view type presents objects and methods independently from the file structure
- **By Packages:** This is mostly relevant for Java and displays packages and components

Double-click a node in the Asset Browser to open the source file or test script in the text editor at the corresponding line.

To switch Project Explorer views:

- Click the **Project Browser** or **Asset Browser** tab.

To hide or show the Project Explorer:

1. Right-click an empty area within the toolbar.
2. Select or clear the **Project Window** menu item.

or from the **View** menu, select **Other Windows** and **Project Window**.

Properties Window

The **Properties Window** box contains information about the node selected in the Project Explorer. It also allows you to modify this information.

Project Browser

Depending on the node selected, any of the following relevant information may be displayed:

- **Name:** is the name carried by the node in the Project Explorer.
- **Exclude from Build:** excludes the node from the Build process. When

this option is selected a cross is displayed next to the node in the **Project Explorer**.

- **Execute in background:** enables the build and execution of more than one test or application node at the same time.
- **Relative path:** indicates the relative path of the file.
- **Full path:** indicates the entire path of the file.
- **Source type:** You can select either **Integrated** or **Tested**.

Asset Browser

Select the type of Object View in the **Sort Method** box at the top of the **Project Explorer** window: **By Object**, **By Files**, or **By Packages**. Depending on the sort method selected, and the type of object or file, any of the following relevant information may be displayed:

- **Name:** is the name carried of the file, object or package.
- **Filters (for folders):** is the file extension filter for files in that folder. See *Creating a Source File Folder*.
- **Name:** is the name carried of the file or package.
- **Relative path:** indicates the relative path of the file.
- **Full path:** indicates the entire path of the file.

To open the Properties window:

1. In the **Project Explorer**, right-click a node.
2. Select **Properties...** in the pop-up menu.

To hide or show the Properties window:

1. Right-click an empty area within the toolbar.
2. Select or clear the *<object>* **Property** menu item.

or from the **View** menu, select **Other Windows** and *<object>* **Property**.

Report Explorer

The **Report Explorer** allows you to navigate through all text and graphical reports, including:

- Test reports generated by Test RealTime
- Memory Profiling, Performance Profiling and Code Coverage reports
- UML Sequence Diagram reports from the Runtime Tracing feature
- Metrics produced by the **Metrics Viewer**

The actual appearance of the Report Explorer contents depends on the nature of the report that is currently displayed, but generally the Report Explorer offers a dynamic hierarchical view of the items encountered in the report.

Click an item in the Report Explorer to locate and select it in the **Report Viewer** or **UML/SD Viewer** window.

To hide or show the Report Explorer:

1. Right-click an empty area within the toolbar.
2. Select or clear the **Report Explorer** menu item.

Standard Toolbars

The toolbars provide shortcut buttons for the most common tasks.

The following toolbars are available

- Main toolbar
- View toolbar
- Build toolbar
- Status bar

Main Toolbar

The main toolbar is available at all times:

- The **New File** button creates a new blank text file in the Text Editor.
- The **Open** button allows you to load any project, source file, test script, or report file supported by the product.
- The **Save File** button saves the contents of the current window.
- The **Save All** button saves the current workspace as well as all open files.
- The **Cut**, **Copy** and **Paste** buttons provide the standard clipboard functionality.
- The **Undo** and **Redo** buttons allow you undo or redo the last command.
- The **Find** button allows you to locate a text string in the active Text Editor or report window.

View Toolbar

The View toolbar provides shortcut buttons for the Text Editor and report viewers.

- The **Choose zoom Level** box and the **Zoom In** and **Zoom Out** buttons are classic Zoom controls.
- The **Reload** button refreshes the current report in a report viewer. This is useful when a new report has been generated.
- The **Reset Observation Traces** button clears cumulative reports such as those from Code Coverage, Memory Profiling or Performance Profiling.

Build Toolbar

The build toolbar provides shortcut buttons to build and run the test.

- The **Configuration** box allows you to select the target configuration on which the test will be based.
- The **Build** button launches the build and executes the node selected in the Project Explorer. You can configure the Build Options for the workspace by selecting the **Options** button.
- The **Stop** button stops the build or execution.
- The **Clean Parent Node** button removes files created by previous tests.
- The **Execute Node** button executes the node selected in the Project Explorer.

Status Bar

The Status bar is located at the bottom of the main GUI window. It includes a **Build Clock** which displays execution time, and the **Green LED** which flashes when work is in progress.

To hide or show a toolbar:

1. Right-click an empty area within the toolbar.
2. Select and clear those toolbars you want to display or hide.

or from the **View** menu, select **Toolbars** and the toolbar(s) you want to display or hide.

Using the GUI Components

Report Viewer

The Report Viewer allows you to view Test or Runtime Analysis reports from Component Testing, System Testing and any of the Runtime Analysis

features

Most reports are produced as XML-based **.xrd** files, which are generated during the execution of the test or application node.

To navigate through the report:

- You can use the Report Explorer to navigate through the report. Click an element in the **Report Explorer** to go to the corresponding line in the **Report Viewer**.
- You can also jump directly to the next or previous Failed test in the report by using the **Next Failed Test** or **Previous Failed Test** buttons.

To filter out passed tests:

You can choose to only display the Failed tests in the report.

- From the **Report Viewer** menu, select **Failed Tests Only** or click the **Failed Tests Only** button in the Report Viewer toolbar.
- To switch back to a complete view of the report, from the **Report Viewer** menu, select **All Tests** or click the **All Tests** button in the Report Viewer toolbar.

To hide or show report nodes:

The Report Viewer can hide or show some types of elements of the test, such as Test Cases, Services or Scenarios.

- From the **Report Viewer** menu, select the elements that you want to hide or show.

Understanding Test and Runtime Analysis Reports

The product generates Test and Runtime Analysis reports for each test or runtime analysis feature.

Runtime Analysis Reports

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

Test Verdict Reports

- Component Testing for C and Ada
- Component Testing for C++
- System Testing for C

Setting a Zoom Level

UML sequence diagrams and other reports can be viewed with different zoom levels.

To set the zoom level:

You can directly change the zoom level in the View Toolbar by using the **Zoom In** and **Zoom Out** buttons or by selecting one of the pre-defined or custom levels from the **Choose Zoom Level** box.

Report Viewer Toolbar

The Report toolbar eases report navigation with the **Report Viewer**.

Report Viewer commands are available when a **Report Viewer** window is open:

- The **Previous Failed Test** and **Next Failed Test** buttons allow you to quickly navigate through the Failed items.
- The **Failed Tests Only** or **All Tests** button toggles between the two display modes.

Report Viewer Style Preferences

The **Preferences** dialog box allows you to change the appearance of your Test and Runtime Analysis reports.

To choose Report Editor colors and attributes

1. Select the **Report Viewer** node:
 - **Background color:** This allows you to choose a background color for the **Report Viewer** window.
2. Expand the **Report Viewer** node, and select **Syntax Color:**
 - **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
 - **Font:** This allows you to change the font type and size for the selected style.
 - **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
 - **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.
3. Click **OK** to apply your changes.

Text Editor

The product GUI provides its own Text Editor for editing and browsing script files and source code.

The Text Editor is a fully-featured text editor with the following capabilities:

- Syntax Coloring
- Find and Replace functions
- Go to line or column

The main advantage of the Text Editor included in the GUI is its tight integration with the rest of the test environment. You can click items within the **Project Explorer**, **Output Window**, or any Test and Runtime Analysis Report to immediately highlight and edit the corresponding line of code in the Editor.

Creating a Text File

To create a new text file:

1. Click the **New Text File** toolbar button,
2. From the **Editor** menu, use the **Syntax Color** submenu to select the language.

or

1. From the **File** menu, select **New...** and then open the **Text File** option
2. From the **Editor** menu, use the **Syntax Color** submenu to select the language.

Opening a Text File

The Text Editor is tightly integrated with the Test RealTime GUI. Because of the links between the various views of the GUI, there are many ways of opening a text file. The most common ones are described here.

Using the Open command:

1. From the **File** menu, select **Open...** or click the **Open** button from the standard toolbar.
2. Use the file selector to select the file type and to locate the file.
3. Select the file you want to open.
4. Click **OK**.

Using the File Explorer:

1. Select a file in the Project Explorer. If there are recognized components

in the file, a '+' symbol appears next to it.

2. Click the '+' symbol to expand the list of references in the file.
3. Double-click a reference to open the **Text Editor** at the corresponding line.

Tip: You can navigate through the source file by double-clicking other reference points in the **Project Explorer**.

Using a Test or Report Viewer:

1. With the **Report Viewer** open, locate an element inside the report.
2. Double-click the item to open the **Text Editor** at the corresponding line.

Finding Text in the Text Editor

To locate a particular text string within the **Text Editor**, use the **Find** command.

Search options:

The **Search** box allows you to select the search mode:

- **All** searches for the first occurrence from the beginning of the file.
- **Selected** searches through selected text only.
- **Forward** and **Backward** specify the direction of the search, starting at the current cursor position.

Match case restricts search criteria to the exact same case.

Match whole word only restricts the search to complete words.

Use regular expression allows you to specify UNIX-like regular expressions as search criteria.

To find a text string in the Text Editor:

1. From the **Edit** menu, select **Find...**
2. The editor **Find and Replace** dialog appears with the **Find** tab selected.
3. Type the text that you want to find in the **Find what:** section. A history of previously searched words is available by clicking the **Find List** button.
4. Change search options if required.
5. Click **Find**.

Replacing Text in the Text Editor

To replace a text string with another string, you use the **Find and Replace** command.

To replace a text string:

1. From the **Edit** menu, select **Replace...**
2. The editor **Find and Replace** dialog appears with the **Replace** tab selected.
3. Type the text that you want to change in the **Find what** box. A history of previously searched words is available by clicking the **Find List** button.
4. Type the text that you want to replace it with in the **Replace with** box. A history of previously replaced words is available by clicking the **Replace List** button.
5. Change search options (see below) if required.
6. Click **Replace** to replace the first occurrence of the searched text, or **Replace All** to replace all occurrences.

Search options:

The **Search** box allows you to select the search mode:

- **All** searches for the first occurrence from the beginning of the file.
- **Selected** searches through selected text only.
- **Forward** and **Backward** specify the direction of the search, starting at the current cursor position.

Match case restricts search criteria to the exact same case.

Match whole word only restricts the search to complete words.

Use regular expression allows you to specify UNIX-like regular expressions as search criteria.

Locating a Line and Column in the Text Editor

The **Go To** command allows you to move the cursor to a specified line and column within the Text Editor.

To use the Go To feature:

1. From the **Edit** menu, select **Go To...**
2. The Text Editor's **Find and Replace** dialog appears with the **Go To** tab selected.
3. Enter the number of the line or column or both.
4. Click **Go** to close the dialog box and to move the cursor to the specified position.

Text Editor Syntax Coloring

The Text Editor provides automatic syntax coloring for C, C++, and Ada source code as well for the C and Ada, C++ test script languages, and System Testing Script Language. The Text Editor automatically detects the language based on the filename extension.

However, if the filename does not have a standard extension, you must select the language from the **Syntax Color** submenu.

To manually set the syntax coloring mode:

1. From the **Editor** menu, select the desired language through the **Syntax Color** submenu.

Note To change the colors used by the Text Editor, see Text Editor Preferences.

Text Editor Preferences

The **Preferences** dialog box allows you to change the appearance of the source code and scripts in the Text Editor.

To choose Editor report colors and attributes:

1. Select the **Editor** node.
 - **Font:** This allows you to change the general font type and size for Editor. This parameter is overridden for defined styles by the **Style** font setting. This parameter can be overridden for defined styles by the **Style** font settings.
 - **Global Colors:** This is where you select background colors for text categorized as **Normal**, **Information** or **Error** as well as the general background color. Click a color to open a standard color palette.
 - **Autodetect parenthesis and bracket mismatch** - When this option is selected, the **Error** color is used when the Editor detects a missing bracket "]" or parenthesis "(".
 - **Tabulation length:** This specifies the tabulation length, which is equivalent to a number of inserted spaces.
2. Expand the **Editor Viewer** node, and select **Styles:**
 - **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
 - **Font:** This allows you to change the font type and size for the selected style.

- **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
 - **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.
3. Click **OK** to apply your changes.

Tools Menu

The **Tools** menu is a user-configurable menu that allows you to access personal tools from the graphical user interface (GUI). You can customize the Tools menu to meet your own requirements.

Custom tools can be applied to a selection of nodes in the Project Explorer. Selected nodes can be sent as a parameter to a user-defined tool application. A series of macro variables is available to pass parameters on to your tool's command line.

See the section **GUI Macro Variables** in the **Reference Manual** for detailed information about using the macro command language.

Using the Tools Menu

To use a user-defined tool:

1. Select an icon from the **Project Explorer** pane.
2. Click the **Tools** menu and select the tool you want to use.

To add a new tool to the Tools menu:

1. From the **Tools** menu, select **Toolbox...**
2. To create an entirely new tool, click **Add...** If you want to copy from an existing tool, select the existing tool, click **Copy** and click **Edit...**
3. Edit the tool in the **Tool Edit** box.
4. Click **OK** and **Close**.

To edit a user-defined tool:

1. From the **Tools** menu, select **Toolbox...**
2. Select the tool that you want to modify and click **Edit...**
3. Edit the tool in the **Tool Edit** box.
4. Click **OK** and **Close**.

To remove a tool from the Tools menu:

1. From the **Tools** menu, select **Toolbox...**
2. Select an existing tool from the tool list.
3. Click **Remove** and **Close**.

Tool Configuration

The **Tool Configuration** dialog allows you to configure a new or existing tool.

In the **Tools** menu, each tool appears as a submenu item, or **Name**, with one or several associated actions or **Captions**.

Identification

In this tab, you describe how the tool will appear in the **Tools** menu.

- Enter the **Name** of the tool submenu as it will appear in the Tools menu and a **Comment** that is displayed in the lower section of the Toolbox dialog box.
- Select **Change Management System** if the tool is used to send and retrieve from a change management system. When **Change Management System** is selected, **Check In** and **Check Out** actions are automatically added to the Action tab (see below) and a **Change Management System** toolbar is activated.
- Clear the **Add to Tools menu** checkbox if you do not want the tool to

be added to the Tools menu.

- Select **Send messages to custom tab in the Output Window** if you want to view the tool's text output in the **Output Window**.
- Use the **Icon** button to attach a custom icon to the tool that will appear in the **Tools** menu. Icons must be either **.xpm** or **.png** graphic files and have a size of 22x22 pixels.

Actions

This tab allows you to describe one or several actions for the tool.

- The **Actions** list displays the list of actions associated with the tool. If **Change Management System** is selected on the **Identification** tab, **Check In** and **Check Out** tool commands will listed here. These cannot be renamed or removed.
- **Menu text** is the name of the action that will appear in the **Tools** submenu.
- **Command** is a shell command line that will be executed when the tool action is selected from **Tools** menu. Command lines can include toolbox macro variables and functions.

Click **OK** to validate any changes made to the Tool Edit dialog box.

To add a new action:

- Enter a **Caption** and a **Command**, then click **Add**.

To remove an action from the list:

- Select an action in the **Actions** list and click **Remove**.

To modify an action:

- Select an action, make any changes in the **Caption** or **Command** lines, and click **Modify**.

Test Process Monitor

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

Each generated metric is stored in its own file and consists of one or more fields.

The Test Process Monitor works by gathering the statistical data from these files and then generating a graphical chart based on each field.

The preexistence of a file is required before running the Test Process Monitor. Files are created either by running a runtime analysis feature that generates test process data, or by creating and updating your own file.

Note Currently only the Code Coverage feature provides data for the Test Process Monitor. You can, however, build your own files with the **tpmadd** command-line feature. See the **Reference Manual** for further information.

Changing Curve Properties

The **Curve Properties** menu allows you to change the way a particular graph is displayed.

To change the curve color:

1. Right-click a curve.
2. From the pop-up menu, select **Change Curve Color**.
3. Use the **Color Palette** to select a new color, and click **OK**.

To hide a curve:

1. Right-click a curve.
2. From the pop-up menu, select **Hide Curve**.

To set a maximum value:

Changing the maximum displayed value for a curve actually changes the

scale at which it is displayed. For instance, when a curve only reaches 100, there is no point in displaying it at on a scale of 1000, unless you want to compare it with another curve that uses that scale.

1. Right-click a curve.
2. From the pop-up menu, select **Set Max Value**.
3. Enter the scale value, and click **OK**.

Note Setting a maximum value lower than the actual maximum value of a curve can result in erratic results.

To display a scale:

For any curve, you can display a scale on the right or left-hand side of the graph. When you display a new scale, it replaces any previously displayed one.

1. Right-click a curve.
2. From the pop-up menu, select **Right Scale** or **Left Scale**.

Custom Curves

In some cases, you may want to remove certain figures from a chart to make it more relevant. The custom curves capability allows you to alter the chart by selecting the records that you want to include.

Note Using the custom curves capability does not impact the actual database. If you remove a record from the chart by using the custom curves function, the actual record remains in the database and may impact other figures.

Custom curves create a new metric, using the name of the base metric, with a Custom prefix.

To create a custom curve:

1. Make sure a user is selected in the **Report Explorer** pane. If not, select a

user.

2. From the **Project** menu, select **Test Process Monitor** and **Custom Curves**.
3. In the **Custom Curves** dialog box, select a metric and the start and end date of your chart.
4. The record list displays all the records contained in the database of that metric. Select the records that you want to use for your custom curve. Clear the records that you do not want to use.
5. Click **OK**. A new metric is created.

To change a custom curve:

1. From the Project menu, select Test Process Monitor and Custom Curves.
2. In the **Custom Curves** dialog box, select the Custom metric that you want to modify.
3. Select the records that you want to use for your custom curve. Clear the records that you do not want to use.
4. Click **OK**.

Event Markers

Use event markers to identify milestones or special events within your Test Process Monitor chart. An event marker is identified by the date of the event and a marker label.

Event markers appear as bold vertical lines in a Test Process Monitor chart.

To create an event marker:

1. Right-click the location where you want to put the chart
2. From the pop-up menu, select **Event Properties** and **New Event**.
3. Enter the date of the event, and a marker label, and click **OK**.

To remove an event marker:

1. Right-click the event marker that you want to hide.
2. From the pop-up menu, select **Delete Event**.

To hide a specific event marker:

Hiding a marker does not remove it. You can still make the marker reappear.

1. Right-click the event marker that you want to hide.
2. From the pop-up menu, select **Hide Event**.

To hide or show all event markers:

1. In the **Test Process Monitor** toolbar, click the **Events** button to hide all event markers.
2. Click again to show all hidden event markers.

Setting the Time Scale

The Scale capability defines the period that you want to view in the **Test Process Monitor** window. This option allows you to select an annual, monthly or daily view, as well as a user-definable time period.

To set the time scale:

1. Select a user in the **Report Explorer** pane.
2. From the **Project** menu, select **Test Process Monitor, Scale** and the desired time scale.
3. If you chose **Customize**, enter the start and end date of the period that you want to monitor, and click **OK**.

Test Process Monitor Toolbar

The Test Process Monitor (TPM) toolbar is useful for navigating through TPM charts.

These buttons are available when a TPM window is open:

- The **Clear** button removes all curves from the chart.
- The **Hide Event** button hides the displayed event markers.
- The **Floating Schedule** button toggles the automatic location of new curves.

To hide or show a toolbar:

1. Right-click an empty area within the toolbar.
2. Select and clear those toolbars you want to display or hide.

Adding a Metric

Metrics generated Code Coverage or other tools are directly available through the Test Process Monitor. Each metric file contains one or several fields.

To open a metric database a metric chart:

1. From the **Project** menu, select **Test Process Monitor** and either **Project** or **Current Workspace**. **Current Workspace** applies to the user of the current workspace. **Project** applies to all workspace users in the project.
2. If a new metric database is detected, you need to provide a name for the metric, as well as a label for each field of the database.
3. In the **Report Explorer**, select a user.
4. From the **Project** menu, select **Test Process Monitor**, the metric and the field that you want to display.

You can add as many curves as you want to the chart.

To hide a curve:

1. Right-click a curve.
2. From the pop-up menu, select **Hide Curve**.

UML/SD Viewer

The UML/SD Viewer renders sequence diagram reports as specified by the UML standard.

UML sequence diagram can be produced directly via the execution of the SCI-instruction application when using the Runtime Tracing feature.

The UML/SD Viewer can also display UML sequence diagram results for Component and System Testing features.

Navigating through UML Sequence Diagram

There are several ways of moving around the UML sequence diagrams displayed by the UML/SD Viewer:

- **Navigation Panel:** Click and drag the **Navigation** button in the lower right corner of the **UML/SD Viewer** window to scroll through a miniature navigation pane representing the entire UML sequence diagram.
- **Free scroll:** Press the **Control** key and the left mouse button simultaneously. This displays a compass icon, allowing you to scroll the UML sequence diagram in all direction by the moving the mouse.
- **Report Explorer:** The Report Explorer is automatically activated when the UML/SD Viewer is activated. The Report Explorer offers a hierarchical view of instances. Click an item in the **Report Explorer** to locate and select the corresponding UML representation in the main **UML/SD Viewer** window.

Time Stamping

The UML/SD Viewer displays time stamping information on the left of the UML sequence diagram. Time stamps are based on the execution time of the application on the target.

You can change the display format of time stamp information in the UML/SD Viewer Preferences.

The following time format codes are available:

- **%n** - nanoseconds
- **%u** - microseconds
- **%m** - milliseconds
- **%s** - seconds
- **%M** - minutes
- **%H** - hours

These codes are replaced by the actual number. For example, if the time elapsed is 12ms, then the format **%mms** would result in the printed value **12ms**. If the number 0 follows the % symbol but precedes the format code, then 0 values are printed to the viewer - otherwise, 0 values are not printed. For example, if the time elapsed is 10ns, and the selected format code is **%0mms %nns**, then the time stamp would read **0ms 10ns** .

Note: To change the format code you must press the Enter key immediately after selecting/entering the new code. Simply pressing the **OK** button on the **Preferences** window will not update the time stamp format code.

Coverage Bar

In C, C++ and Java, the coverage bar provides an estimation of code coverage.

Note The coverage bar is unrelated to the Code Coverage feature. For detailed code coverage reports, use the dedicated Code Coverage feature.

When using the Runtime Tracing feature, the UML/SD Viewer can display an extra column on the left of the UML/SD Viewer window to indicate code coverage simultaneously with UML sequence diagram messages.

The UML/SD Viewer code coverage bar is merely an indication of the ratio of *encountered* versus *declared* function or method entries and potential exceptions since the beginning of the sequence diagram.

If new declarations occur during the execution the graph is recalculated, therefore the coverage bar always displays a increasing coverage rate.

To activate or disable coverage tracing with a Java application:

1. Before building the node-under-analysis, open the Memory Profiling settings box.
2. Set **Coverage Tracing** to **Yes** or **No** to respectively activate or disable code coverage tracing for the selected node.
3. Click **OK** to override the default settings of the node

To hide the coverage bar:

1. Right-click inside the **UML/SD Viewer** window.
2. From the pop-up menu, select **Hide Coverage**.

To show the coverage bar:

1. Right-click inside the **UML/SD Viewer** window.
2. From the pop-up menu, select **Show Coverage**.

Memory Usage Bar

When using the Runtime Tracing feature on a Java application, the UML/SD Viewer can display an extra bar on the left of the UML/SD Viewer window to indicate total memory usage for each sequence diagram message event.

The memory usage bar indicates how much memory has been allocated by the application and is still in use or not garbage collected.

In parallel to the UML sequence diagram, the graph bar represents the allocated memory against the highest amount of memory allocated during the execution of the application.

This ratio is calculated by subtracting the amount of free memory from the total amount of memory used by the application. The total amount of memory is subject to change during the execution and therefore the graph is recalculated whenever the largest amount of allocated memory increases.

A tooltip displays the actual memory usage in bytes.

To activate or disable coverage tracing with a Java application:

1. Before building the node-under-analysis, open the **Memory Profiling** settings box.
2. Set **Coverage Tracing** to **Yes** or **No** to respectively activate or disable coverage tracing for the selected node.
3. Click **OK** to override the default settings of the node

To hide the memory usage bar:

1. Right-click inside the **UML/SD Viewer** window.
2. From the pop-up menu, select **Hide Memory Usage**.

To show the memory usage bar:

1. Right-click inside the **UML/SD Viewer** window.
2. From the pop-up menu, select **Show Memory Usage**.

Thread Bar

When using the Runtime Tracing feature on C, C++ and Java code, the UML/SD Viewer can display an extra column on the left of its window to indicate the active thread during each UML sequence diagram event.

Each thread is displayed as a different colored zone. A tooltip displays the name of the thread.

Thread List

click the thread bar to open the thread list. The thread list window displays a list of all threads that are created during execution of the application.

You can change the sort order by clicking the column titles.

You can jump to the portion of source code that creates a thread by clicking a thread name.

To hide the thread bar:

1. Right-click inside the **UML/SD Viewer** window.
2. From the pop-up menu, select **Hide Thread Bar**.

To show the thread bar:

1. Right-click inside the **UML/SD Viewer** window.
2. From the pop-up menu, select **Show Thread Bar**.

Applying Filters

Filters are used to detect particular events within a test trace. You use the Viewer's **Filter List** dialog box to specify how events are to be detected and filtered.

To access the Filter List:

- From the **UML/SD Viewer** menu, select **Filters** or click the **Filter** button in the UML/SD Viewer toolbar.

To create a new filter:

1. Click the **New** button
2. Create the new filter with the Event Editor.

To modify an existing filter:

1. Select the filter that you want to change.
2. Click the **Edit** button.

3. Modify the filter with the Event Editor.

To import one or several filters:

The import facility is useful if you want to re-use filters created in another Project.

1. Click the **Import** button.
2. Locate and select the **.tft** file(s) that you want to import.
3. Click **OK**.

To export a filter event:

The export facility allows you to transfer filters.

1. Select the filter that you want to export.
2. Click the **Export** button.
3. Select the location and name of the exported **.tft** file.
4. Click **OK**.

Sequence Diagram Triggers

Sequence Diagram triggers allow you to predefine automatic start and stop parameters for the UML/SD Viewer. The trigger capability is useful if you only want to trace a specific portion of an instrumented application.

Triggers can be inactive, time-dependent, or event-dependent.

To access the Trigger dialog box:

- From the **UML/SD Viewer** menu, select **Triggers** or click the **Trigger** button in the UML/SD Viewer toolbar.

Start and End of Runtime Tracing:

The Runtime Tracing start is defined on the **Start** tab:

- **At the beginning:** Runtime Tracing starts when the application starts.

- **On time:** Runtime Tracing starts after a specified number of microseconds.
- **On event:** Runtime Tracing starts when a specified event is detected. One or several events must be specified with the Event Editor.

The Runtime Tracing end is defined on the **Stop** tab:

- **Never:** Runtime Tracing ends when the application exits.
- **On time:** Runtime Tracing ends after a specified number of seconds.
- **On event:** Runtime Tracing ends when a specified event is detected. One or several events must be specified with the Event Editor.

To create a new trigger event:

1. Click the **New** button
2. Create the new trigger event with the **Event Editor**.

To modify an existing trigger event:

1. Select the trigger event that you want to change.
2. Click the **Edit** button.
3. Modify the trigger event with the **Event Editor**.

To import one or several trigger events:

The import facility is useful if you want to reuse trigger events created in another Project.

1. Click the **Import** button.
2. Locate and select the file(s) that you want to import.
3. Click **OK**.

To export a trigger event:

The export facility allows you to transfer trigger events.

1. Select the trigger event that you want to export.

2. Click the **Export** button.
3. Select the location and name of the exported **.tft** file.
4. Click **OK**.

Editing Trigger or Filter Events

Use the Event Editor to create or modify event triggers or filters for UML sequence diagrams:

- **Filters:** Specified events are hidden or shown in the UML sequence diagram.
- **Start triggers:** The UML/SD Viewer starts displaying the sequence diagram when a specified event is encountered. If no event matches the output of the application, the diagram will appear blank.
- **Stop triggers:** The UML/SD Viewer stops displaying the sequence diagram when a specified event is encountered.

Events can be related to messages, instances, notes, synchronizations, actions or loops.

To define an event or filter:

1. Specify a name for the event.
2. Select the type of UML element you want to define for the event and select **Activate**. Several types of elements can be activated for a single filter or trigger event.
3. Click **More** or **Fewer** to add or remove line to the event criteria.
4. From the drop-down criteria box, select a criteria for the filter, and an argument.

Arguments must reflect an exact match for the criteria. Pay particular attention when referring to labels that appear in the sequence diagram since they may be truncated.

You can use wildcards (*) or regular expressions by selecting the corresponding option.

5. Click the **Aa** button to enable or disable case sensitivity in the criteria.
6. You can add or remove a criteria by clicking the **More** or **Fewer** buttons.
7. Click **Ok**.

Message Criteria

- **Name:** Specifies a message name as the filter criteria.
- **Internal message:** Considers all messages other than constructor calls coming from any internal source, as opposed to those messages coming from the World instance.
- **From Instance:** Considers all messages other than constructor calls prior to the first message sent from the specified object
- **To Instance:** Considers out all messages other than constructor calls if any message is sent to the specified object
- **From World:** Considers all messages received from the World instance
- **To World:** Considers all messages sent to the World instance

Instance Criteria

- **Name:** Specifies an instance name as the filter criteria
- **Instance child of:** Specifies a child instance of the specified class.

Note Criteria

- **All:** Considers all notes
- **Name:** Specifies a note name
- **All message notes:** Considers any note attached to a message

- **All instance notes:** Considers any note attached to an instance
- **Instance child of:** Specifies a note attached to an instance of the specified class
- **Note on message named:** Considers a note attached to a specified message
- **With style named:** Considers a note with the specified style attributes

Synchronization Criteria

- **All:** Considers all synchronization events
- **Name:** Specifies a synchronization name

Action Criteria

- **All:** Considers all actions
- **Name:** Specifies an action name
- **From Instance:** Considers an action performed by the specified object
- **From World:** Considers all actions performed by the World instance
- **Instance child of:** Specifies an action performed by an instance of the specified class
- **With style named:** Considers an action with the specified style attributes

Loop Criteria

- **All:** Considers all loops
- **Name:** Specifies a loop name

Boolean Operators

- **All Except** expresses a NOT operation on the criteria

- **Match All** performs an AND operation on the series of criteria
- **Match Any** performs an OR operation on the series of criteria

Finding Text in a UML Sequence Diagram

The UML/SD Viewer has an extensive search facility that allows users to locate specific UML sequence diagram elements by searching for a text string.

To search for a text string inside the UML/SD Viewer:

1. Click inside a **UML/SD Viewer** window to activate it.
2. Select the **Edit -> Find...** menu item. The **Find** dialog box opens.
3. Type your search criteria in the **Find** dialog box.
4. Click the **Find Next** button.
5. If a string corresponding to the search criteria is found in the UML/SD Viewer, the string is highlighted and the following message is displayed: **Runtime Tracing has finished searching the document.**
6. Click **OK**.

Search options

- **Forward** and **Backward** specifies the direction of the search.
- The **Search into** option allows you to specify type of object in which you expect to find the search string.
- The **Find** dialog box accepts either UNIX regular expressions or DOS-like wildcards ('?' or '*'). Select either **wildcard** or **reg. exp.** in the *Find* dialog box to select the corresponding mode.

Step-by-Step mode

When tracing large applications, it may be useful to slow down the display of the UML sequence diagram. You can do this by using the Step-by-Step

mode.

To activate Step-by-Step mode:

- From the UML/SD Viewer menu, select Display Mode and Step-by-Step.

To select the type of graphical element to skip over:

1. In the UML/SD Viewer toolbar, click the **Step** button.
2. Select the graphical elements that will stop the **Step** command. Clear the elements that are to be ignored.

To step to the next selected element:

- Click the **Step** button in the UML/SD Viewer toolbar.

To skip to the end of execution:

- Click the **Continue** button in the UML/SD Viewer toolbar. This will immediately display the rest of the UML sequence diagram.

To restart the Step-by-Step display:

- Click the **Restart** button in the UML/SD toolbar.

To de-activate Step-by-Step mode

- From the **UML/SD Viewer** menu, select **Display Mode** and **All**.

UML/SD Viewer Toolbar

The UML/SD Viewer toolbar provides shortcut buttons to commands related to viewing graphical test reports and UML sequence diagrams.




UML/SD Viewer commands are only available when a UML sequence diagram is open.

- The **Filter** button allows you to define a sequence diagram filter.
- The **Trigger** button sets sequence diagram triggers.

The following buttons are only available when using the Step-by-Step mode.

- The **Step** button moves the UML/SD Viewer to the next selected event.
- The **Select** button allows you to select the type of event to trace.
- The **Continue** button draws everything to the end of the trace diagram.
- The **Restart** button restarts Step-by Step mode.
- The **Pause** button pauses the On-the-Fly display mode. The application continues to run.

The TDF file selector is only available when using the Split TDF File feature.

- Click the  button to select a **.tdf** dynamic trace file from the list.
- Click the  and  buttons to select the previous or next file in the list.

To hide or show a toolbar:

1. Right-click an empty area within the toolbar.
2. Select and clear those toolbars you want to display or hide.
3. Click **OK**.

UML/SD Viewer Preferences

The **Preferences** dialog box allows you to change the appearance of the UML Sequence Diagram reports.

To choose UML sequence diagram preferences:

1. Select the **UML/SD Viewer** node:
 - **Background:** This allows you to choose a background color for the UML sequence diagram.
 - **Panel:** This allows you to choose a background color for panels in the UML sequence diagram.
 - **Panel Background:** This allows you to choose a background color for selected panels.
 - **Coverage Bar:** This allows you to choose a background color for the

coverage bar.

- **Memory Usage:** This allows you to choose a background color for the memory usage bar.
 - **Print Page header:** Select this option to print a page header.
 - **Print Page footer:** Select this option to print a page footer.
 - **Display Page Breaks:** When this option is selected, the UML/SD Viewer displays horizontal and vertical dash lines representing the page size for printing.
 - **Show tooltip in UML/SD Viewer:** Use this option to hide or show the information tooltip in the UML/SD Viewer.
 - **Time Stamp Format:** Use the editable box to select the format in which time stamps are displayed in the UML/SD Viewer. See Time Stamping.
2. Expand the **UML/SD Viewer** node, and select **Styles** or **Styles System Test**:
 - **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.
 - **Font:** This allows you to change the font type and size for the selected style.
 - **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.
 - **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.
 3. Click **OK** to apply your changes.

Configurations and Settings

Configurations and Settings

Two major concepts of Test RealTime are Configurations and Configuration Settings:

- A Configuration is an instance of a Target Deployment Port (TDP) as used in your project.
- Configuration Settings are the particular properties assigned to each node in your project for a given Configuration.

A Configuration is not the actual Target Deployment Port. Configurations are derived from the Target Deployment Port that you select when the project is created, and contain a series of Settings for each individual node of your project.

This provides extreme flexibility when you are using multiple platforms or development environments. For example:

- You can create a Configuration for each programming language or compiler involved in your project.
- If you are developing for an embedded platform, you can have one Configuration for native development on your Unix or Windows development platform and another Configuration for running and testing the same code on the target platform.
- You can set up several Configurations based on the same TDP, but with different libraries or compilers.
- If you are using multiple programming languages in your project, you can even override the TDP on one or several nodes of a project.

The Configuration Settings allow you to customize test and runtime analysis configuration parameters for each node or group of your project, as

well as for each Configuration. You reach the **Configuration Settings** for each node by right-clicking any node in the Project Explorer window and selecting **Settings**.

The left-hand section of the **Configuration Settings** window allows you to select the settings families related to the node, as well as the Configuration itself, to which changes will be made. The right-hand pane lists the individual setting properties.

The right-hand section contains the various settings available for the selected node.

Propagation Behavior of Configuration Settings

The Project Explorer displays a hierarchical view of the nodes that constitute your project.

Settings for each node are inherited by child nodes from parent nodes. For instance, Settings of a project node will be cascaded down to all nodes in that project.

Child settings can be set to *override* parent settings. In this case, the overridden settings will, in turn, be cascaded down to lower nodes in the hierarchy. Overridden settings are displayed in bold.

Settings are changed only for a particular Configuration. If you want your changes to a node to be made throughout all Configurations, be sure to select **All Configurations** in the Configuration box.

To change the settings for a node:

1. In the Project Explorer, click the Open Settings... button.
2. Use the **Configuration** box to change the Configuration for which the changes will be made.
3. In the left pane, select the settings family that you want to edit.

4. In the right pane, select and change the setting properties that you want to override.
5. When you have finished, click **OK** to validate the changes.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Configuration Settings Structure

The Configuration Settings provides access to the following settings families:

- General
- Build
- Runtime Analysis
- Component Testing

The actual settings available for each node depend on the type of node and the language of the selected Configuration.

General Settings

Runtime Analysis

The Runtime Analysis setting family covers Configuration Settings for Memory Profiling, Performance Profiling, Code Coverage and Runtime Tracing.

Automated Testing Settings

This setting family covers Configuration Settings for Component Testing

and System Testing features (for Rational Test RealTime only).

General Settings

The General settings are part Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Host Configuration

The Host Configuration area lets you override any information about the machine on which the Target Deployment Port is to be compiled.

- **Hostname:** The hostname of the machine. By default this is the **local host**.
- **Address:** The IP address of the host. For the local host, use **127.0.0.1**.
- **System Testing Agent TCP/IP Port:** The port number used by System Testing Agents. The default is **10000**.
- **Socket Uploader Port:** The default value is *7777*.
- **Target Deployment Port:** This allows you to change the Target Deployment Port for the selected nodes. Child nodes will use the default Configuration Settings from this Target Deployment Port, such as compilation flags.

Directories

- **Build:** Specify an optional working directory for the Target Deployment Port. This is where the generated test program will be executed on the target host.

- **Temporary:** Enter the location for any temporary files created during the Build process
- **Report:** Specify the directory where test results are created.
- **Java Main Class (for Java only):** Specifies the name of the main class for Java programs.

Target Deployment Port

The Target Deployment Port (TDP) Settings allow you to override the TDP used for a particular node in the current Configuration. By default, the TDP used is that of the current Configuration.

- **Directory:** Specifies the TDP directory
- **Name:** Displays the name of the TDP.
- **ini File:** Indicates the default **.ini** file in the TDP directory.
- **Language:** Sets the current language of the TDP.

To edit the General settings for a node:

1. In the Project Explorer, click the Open Settings... button.
2. Select a node in the **Project Explorer** pane.
3. In the Configuration Settings list, expand **General**.
4. Select Host Configuration, Directories or Target Deployment Port.
5. When you have finished, click **OK** to validate the changes.

Build Settings

The **Compiler** settings are part of the **Build** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden

fields are displayed in bold.

Compiler Settings

- **Preprocessor options:** Specific compilation flags to be sent to the Test Compiler.
- **Compiler flags:** Extra flags to be sent to the compiler.
- **Preprocessor macro definitions:** Specify any macro definition that are to be sent to both the compiler preprocessor (if used) and the Test Compilers. Several generation conditions must be separated by a comma ',' with no space, as in the following example:

`WIN32,DEBUG=1`

- **Directories for Include Files:** Click the ... button to create or modify a list of directories for included files when the include statement is encountered in source code and test scripts. In the directory selection box, use the **Up** and **Down** buttons to indicate the order in which the directories are searched.
- **User Link File for Ada (for Ada only):** When using the Ada Instrumentor, you must provide a link file. See Ada Link Files for more information.
- **Boot Class Path (for Java only):** Click the ... button to create or modify the Boot Class Path parameter for the JVM.
- **Class Path (for Java only):** Click the ... button to create or modify the Class Path parameter for the JVM.

Linker Settings

This area contains parameters to be sent to the linker during the build of the current node.

- **Link Flags:** Flags to be sent to the linker.
- **Additional objects or libraries:** A list of object libraries to be linked to

the generated executable.

- **Directories for Libraries:** Click the ... button to create or modify a list of directories for library link files. In the directory selection box, use the **Up** and **Down** buttons to indicate the order in which the directories are searched.

Linker Settings for Testing Activities

These linker settings apply to Component Testing and System Testing nodes when using Test RealTime.

- **Test Driver Filename:** The name of the generated test driver binary. By default, Test RealTime uses the name of the test or application node.
- **Main application procedure (for Ada only):** Ada requires an entry point in the source code. For other languages, leave this blank.
- **Build jar file (for Java only):** Specifies whether to build an optional **.jar** file.
- **Jar file name (for Java only):** If **Build jar file** is set to **Yes**, enter the name of the **.jar** file.
- **Manifest file (for Java only):** Specifies the name of an optional manifest file.
- **Jar other directories (for Java only):** Enter the location to generate the **.jar** file. By default this is the source code directory.

Target Deployment Port Settings

This area relates to the parameters of the Target Deployment Port on which is based the Configuration:

- **Measure time used by:** Selects between a real-time **Operating system** clock or a **Process or task** clock for time measurement, if both options are available in the current Target Deployment Port. Otherwise, this

setting is ignored.

- **Maximum on-target buffer size:** This sets the size of the I/O buffer. A smaller I/O buffer can save memory when resources are limited. A larger buffer improves performance.
The default setting for the I/O buffer is 1024 bytes.
- **Multi-threads:** This box, when selected, protects Target Deployment Port global variables against concurrent access when you are working in a multi-threaded environment such as Posix, Solaris or Windows. This can cause an increase in size of the Target Port as-well-as an impact on performance, therefore select this option only when necessary.
- **Maximum number of threads:** When the multi-thread option is enabled, this setting sets the maximum number threads that can be run at the same time by the application.
- **Run Garbage Collector at exit (for Java only):** This setting runs the JVM garbage collection when the application terminates.

To edit the Build settings for a node:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select a node in the Project **Explorer** pane.
3. In the Configuration Settings list, expand **Build**.
4. Select **Compiler**, **Linker** or **Target Deployment Port**.
5. When you have finished, click **OK** to validate the changes.

External Command Settings

The External Command settings are part of the Configuration Settings dialog box, which allows you to configure settings for each node.

Use the External Command setting to set a command line for External Command nodes. An External Command is a command line that can be

included at any point in your workspace. External Commands can contain GUI macro variables, making them context-sensitive. See the **GUI Macro Variables** chapter in the **Reference Manual**.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

To edit the External Command settings for one or several nodes:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select a node in the **Project Explorer** pane.
3. Select the **External Command** node and enter a **Command** line.
4. When you have finished, click **OK** to validate the changes.

Probe Control Settings

The Probe Control settings are part of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

- **Probe Enable:** This setting enables or disables the Trace Probe feature as implemented with System Testing for C. See Trace Probes.
- **Probe Settings:** These settings allow you to select the Probe macro mode.
- **Messaging API:** Use the Add and Remove buttons to create a list of Messaging API files. These are source files that define the structure declarations required by Virtual Testers.

To edit the Probe Control settings for a node:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select a node in the **Project Explorer** pane.
3. In the **Configuration Settings** list, select **Probe Control**.
4. When you have finished, click **OK** to validate the changes.

Runtime Analysis Settings

General Runtime Analysis Settings

The General Runtime Analysis settings are part of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Snapshot Settings

In some case, such as with applications that never terminate or when working with timing or memory-sensitive targets, you might need to dump traces at specific points in your code.

- **On Function Entry:** Allows you to specify a list of function names, from your source code, that will dump traces at the beginning of the function.
- **On Function Return:** Allows you to specify a list of function names, from your source code, that will dump traces at the end of the function.
- **On Function Call:** Allows you to specify a list of function names, from your source code, that will dump traces before the function is called.

For each tab, click the ... button to open the function name selection box. Use the **Add** and **Remove** buttons to create a list of function names.

See [Generating SCI Dumps](#) for more information.

Selective Instrumentation

By default, runtime analysis features instrument all components of source code under analysis.

The Selective Instrumentation settings allow you to more finely define which units (classes and functions) you want to instrument and trace.

- **Units excluded from instrumentation:** Click the ... button to access a list of units (classes and functions) that can be excluded from the instrumentation process. Click a unit to select or clear a unit. Use the **Select File** and **Clear File** buttons to select and clear all units from a source file.
- **Files excluded from instrumentation:** Click the ... button and use the **Add** and **Remove** buttons to select the files to be excluded.
- **Instrument inline methods:** Extends instrumentation to inline methods.
- **Instrument included methods or functions:** Extends instrumentation to included methods or functions.
- **Directories excluded from instrumentation:** Click the ... button and use the **Add**, **Remove** buttons to select the files to be excluded.

Static File Storage

Depending on the runtime analysis feature, the product generates **.tsf** or **.fdc** temporary static data files during source code instrumentation of the application under analysis.

- **Code Coverage Static File Storage (.fdc):** These settings apply to Code Coverage **.fdc** static trace files:
 - **Build directory:** Select this option to use the current directory for all generated files.
 - **Other directory:** Select this option to define a specific directory.
 - **Source directory:** Select this option to use the same directory as the source under analysis.
 - **Use single temporary file (.fdc):** By default, Code Coverage produces one **.fdc** file for each instrumented source file. Select this option to use a single **.fdc** file for all instrumented source files, and specify its location.
- **FDC Directory:** When using the **Use single temporary file (.fdc)** option in the previous setting, specify a location for the **.fdc** file.
- **Memory Profiling, Performance Profiling, and Runtime Tracing Storage:** This setting applies to Memory Profiling, Performance Profiling and Runtime Tracing **.tsf** static trace files.
 - **Build directory:** Select this option to use the current directory for all generated files.
 - **Other directory:** Select this option to define a specific directory.
 - **Source directory:** Select this option to use the same directory as the source under analysis.
 - **Use single temporary file (.tsf):** By default, Memory Profiling, Performance Profiling and Runtime Tracing produces one **.tsf** file for each instrumented source file. Select this option to use a

single **.tsf** file for all instrumented source files, and specify its location.

- **TSF Directory:** When using the **Use single temporary file (.tsf)** option in the previous setting, specify a location for the **.tsf** file.

Miscellaneous Options

- **Label Instrumented Files:** Select this option to add an identification header to files generated by the Instrumentor, including the command line used to generate the file, the version of the product, date and operating system information.
- **Full template instantiation:** By default unused methods are ignored by the Instrumentor. Set this option to **Yes** to analyze all template methods, even if they are not used.
- **Additional Instrumentor Options:** This setting allows you to add command line options for the Instrumentor. Normally, this line should be left blank.

To edit the General Runtime Analysis settings for a node:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select a node in the Project **Explorer** pane.
3. In the Configuration Settings list, expand **Runtime Analysis** and **General**.
4. Select **Snapshot, Selective Instrumentation, Static File Storage** or **Miscellaneous**.
5. When you have finished, click **OK** to validate the changes.

Memory Profiling Settings

The **Memory Profiling Instrumentation Control** and **Memory Profiling Misc. Options** settings are part of the **Runtime Analysis** node of the Configuration Settings dialog box, which allows you to configure settings

for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Instrumentation Control

- **File in use (FIU):** When the application exits, this option reports any files left open.
- **Memory in use (MIU):** When the application exits, this option reports allocated memory that is still referenced.
- **Signal (SIG):** This option indicates the signal number received by the application forcing it to exit.
- **Freeing Freed Memory (FFM) and Late Detect Free Memory Write (FMWL):** Select **Display Message** to activate detection of these errors.
- **Free queue length (blocks)** specifies the number of memory blocks that are kept free.
- **Free queue size (Kbytes)** specifies the total buffer size for *free queue* blocks. See Freeing Freed Memory (FFM) and Late Detect Free Memory Write (FMWL).
- **Display Detect Array Bounds Write (ABWL):** Select **Yes** to activate detection of this error.
- **Red zone length (bytes)** specifies the number of bytes added by Memory Profiling around the memory range for bounds detection.
- **Number of functions:** specifies the maximum number of functions reported from the end of the CPU call stack. The default value is 6.

Misc. Options

- **Trace File Name (.tpf):** This box allows you to specify a filename for the generated .tpf trace file.
- **Global variables to exclude from observation (for Java only):** This box specifies a list of 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. Use the **Add** and **Remove** buttons to add and remove global variables.

JVMPI

- **Object hashtable size:** Specifies the size of hashtables for objects where <size> must be 64, 256, 1024 or 4096 values.
- **Class hashtable size:** Specifies the size of hashtables for classes where <size> must be 64, 256, 1024 or 4096 values.
- **Take a Snapshot:** You can select one of the following options:
 - **On method entry or return or dump snapshot button:** Uses a specified method to perform snapshot or the GUI snapshot button as specified in the Enable dump Snapshot button setting.
 - **After each Garbage Collection:** Takes a snapshot each time the JVM garbage collector runs.
- **Enable dump snapshot button and Delay Snapshot until next Garbage Collection:** Specify the trigger method.
- **Host name used by dump Snapshot button:** Use this option to specify a hostname for the JVMPI Agent to communicate with the GUI.
- **Port Number used by dump Snapshot button:** Use this option to specify a port number for the JVMPI Agent to communicate with the GUI.
- **TPF file name (.tpf):** Specifies the name of the Memory Profiling trace dump file produced by the JVMPI Agent.

- **TSF file name (.tsf):** Specifies the name of the static trace dump file.
- **Display only listed methods:** Use the Add and Remove buttons to add and remove methods to be listed by the Java Memory Profiling report.
- **Collect referenced objects:** Sets the filter to be used with the Java Memory Profiling Report.
- **Display only listed packages:** Use this setting to filter out of the report the packages that do not match the specified full package name (package and class).
- **Display only listed classes:** Use this setting to filter out of the report the classes that do not match the specified full classes.
- **Display call stack for listed methods:** Use this setting to list the methods for which the call stack is to be displayed in the Java Memory Profiling report.

To edit the Memory Profiling settings for one or several nodes:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select one or several nodes in the Configuration pane.
3. Select the **Runtime Analysis** node and the **Memory Profiling** node.
4. Select either **Instrumentation Control**, **Misc. Options** or **JVMPI**.
5. When you have finished, click **OK** to validate the changes.

Performance Profiling Settings

The **Performance Profiling** settings are part of the **Runtime Analysis** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Trace File Name (.tqf): This box allows you to specify a filename for the generated **.tqf** trace file for Performance Profiling.

To edit the Performance Profiling settings for one or several nodes:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select one or several nodes in the Configuration pane.
3. Select the **Runtime Analysis** node and the **Performance Profiling** node.
4. When you have finished, click **OK** to validate the changes.

Code Coverage Settings

The Code Coverage Instrumentation Control settings are part of the **Runtime Analysis** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Instrumentation Control Settings

You can use the Coverage Type settings to declare various types of coverage.

- **Coverage Level Functions or Methods:** select between function **Entries**, **With exits**, or **None**. See the Function or Method Code Coverage Ada, C, and C++ for more information.
- **Coverage Level Calls:** select **Yes** or **No** to toggle call code coverage for Ada and C.
- **Coverage Level Blocks:** select the desired block code coverage method. See the Block Code Coverage for Ada, C, and C++ for details.

- **Coverage Level Conditions:** select condition code coverage for Ada, C.

Please refer to Selecting Coverage Types for details on using each coverage type with each language.

You can combine, enable, or disable any of these coverage types before running the application node. All coverage types selected for instrumentation can be filtered out in the Code Coverage Viewer.

- **Mode:** This setting specifies the Instrumentation Modes to be used by Code Coverage.
- **Default (Optimized for Code Size and Speed):** This setting uses one byte per branch to indicate branch coverage.
- **Compact (Optimized for Memory):** This setting uses one bit per branch. This method saves target memory but uses more CPU time.
- **Report Hit Count:** This adds information about the number of times each branch was executed. This method uses one integer per branch.
- **Prefix (for Ada only):** Add a new prefix to Ada packages if the default Code Coverage prefix (**atc_**) generates conflicts.
- **Suffix (for Ada only):** Specifies how Code Coverage names the instrumented Ada packages:
 - Select **Standard** to use the your package name as a suffix
 - Select **Short** to reduce the size of the generated package name for compilers that have a package name length limit.

Selective Code Coverage Instrumentation

- **C/C++ Ternary coverage:** For C and C++, when this option is selected, Code Coverage is extended to ternary expressions as statement blocks.
- **Ada specification:** For Ada, selecting this option extends instrumentation to Ada package specifications. Specifications can

contain calls and conditions. In this case, the specification file must be included in the application node.

- **Functions to Exclude from Calls Code Coverage:** Specifies a list of functions to be excluded from the call coverage instrumentation type, such as **printf** or **fopen**. Use the **Add**, **Remove** buttons to tell Code Coverage the functions to be excluded.

Miscellaneous Options

- **Trace File Name (.tio):** this allows you to specify a path and filename for the **.tio** dynamic coverage trace file.
- **Compute Deprecated Metrics:**
- **User comment:** This adds a comment to the Code Coverage Report. This can be useful for identifying reports produced under different Configurations. To view the comment, click the a magnifying glass symbol that is displayed at the top of your source code in the Code Coverage Viewer.

To change the Code Coverage Instrumentation Control setting for an application or test node.

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select one or several nodes in the Configuration pane.
3. Select the **Runtime Observation** node, and the **Coverage** node.
4. Select **Instrumentation Control**.
5. When you have finished, click **OK** to validate the changes.

Runtime Tracing Control Settings

The Runtime Tracing Control settings are part of the **Runtime Analysis** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Instrumentation Control

- **Trace File Name (.tdf):** This allows you to force a filename and path for the dynamic **.tdf** file. By default, the **.tdf** carries the name of the application node.
- **Functions called within a return expression are sequenced:** For C 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.
- **Collapse unnamed classes and structures:** For C++ only. With this option, unnamed *structs* and *unions* are not instrumented.
- **Display class template instantiation in a note:** For C++ only. With this option, the UML/SD Viewer will not display notes for template instances for each template class instance.

Trace Control

- **Split Trace File Enable:** See Splitting trace files for more information on this setting.
- **Maximum Size (Kbytes):**
- **File name prefix:**
- **Automatic Loop Detection Enable:** Loop detection simplifies UML sequence diagrams by summarizing repeating traces into a loop symbol. Loops are an extension to the UML sequence diagram standard and are not supported by UML.
- **Options (Reserved for future use):**

- **Display largest call stack length:** When selected, the Target Deployment Port records the highest level attained by the call stack during the trace. This information is displayed at the end of the UML Sequence Diagram in the UML/SD Viewer as **Maximum Calling Level Reached**.

Target Deployment Port Settings

These settings allow you to set compilation flags that define how the Runtime Tracing feature interacts with the Target Deployment Port. These are general settings for the Target Deployment Port.

- **Disable on-the-fly mode:** When selected, this setting stops on-the-fly updating of the dynamic **.tdf** file. This option is primarily for Target Deployment Ports that use **printf** output.
- **Trace Buffer Enable** and **Partial Trace Flush Enable:** Please see Trace Item Buffer and Partial Trace Flush for more information about these settings.
- **Maximum number of recorded Trace elements before buffer flush**
- **When receiving user signal: No Action, Flush Call Stack, Trace On/Off**
- **Record and display Time Stamp:** This setting adds time stamp information to each element in the UML sequence diagram generated by Runtime Tracing.
- Record and display Heap Size:
- Record and display Thread Info:

To edit the Runtime Tracing Control settings for one or several nodes:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select one or several nodes in the Configuration pane.
3. Select the **Runtime Analysis** node and the **Runtime Tracing** node.

4. Select **Runtime Tracing Control**.
5. When you have finished, click **OK** to validate the changes.

Automated Testing Settings

Component Testing Settings for C and Ada

The Component Testing settings are part of the **Component Testing for C and Ada** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Display

- **Display variables:** lets you select the level of detail of the Component Testing output report
- **Initial and expected value display:** the way in which the values assigned to each variable are displayed in the report. See Initial and Expected Values.
- **Array and structure display:** indicates the way in which Component Testing processes variable array and structure statements. See Array and Structure Display for more information.

Additional Options

- **Continue test build despite warnings:** Select this option to ignore warning during the test compilation phase.
- **Call breakpoint function on test failure:** Select this option to call a breakpoint function whenever a test failure occurs in a **.ptu** Test Driver

script. To use this feature, you must set a breakpoint on the function **priv_check_failed ()**, located in the `<target_deployment_port>/lib/priv.c` file. You can use this option for debugging purposes.

- **Simulation:** This setting determines the conditional generation of code in the test program when using **SIMUL** blocks in the **.ptu** test script.
- **Display diff of last two test runs:** This setting activate the comparison option. See Comparing Reports.
- **Additional test compilation options enabled:**
- **Additional test compilation options:** Sends extra command line options to the Component Testing Test Compiler. Please refer to the **Test RealTime Reference Manual** for further information about addressing the Test Compiler in command-line mode.
- **Additional report generation options:** Sends extra command line options to the Component Testing Report Generator. Please refer to the **Test RealTime Reference Manual** for further information about addressing the Report Generator in command-line mode.

To edit the Component Testing for C and Ada settings for a node:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select a node in the Project **Explorer** pane.
3. In the Configuration Settings list, expand **Component Testing for C and Ada**.
4. Select **Display, Additional Options**, or **Comparison**.
5. When you have finished, click **OK** to validate the changes.

Component Testing for C++ Settings

The Component Testing settings for C++ are part of the Configuration Settings dialog box, which allows you to configure settings for each node in your workspace.

The Component Testing for C++ node settings lets you to customize the parameters for the Component Testing for C++ feature of Test RealTime.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Files

This area specifies the path and filenames for the intermediate files generated by the Component Testing for C++ feature during the test execution.

- **Test report file name (.xrd):** contains the location and name of the **.xrd** report file generated by Component Testing for C++
- **Generated test driver source file name:** contains the location and name of a **.cpp** source file generated from the C++ Test scripts by Component Testing for C++
- **Contract check file name:** contains the path and file name of a temporary **.oti** file created during source code instrumentation by Component Testing for C++

General Options

This area contains general information for the Component Testing for C++ feature.

- **Maximum test compilation errors displayed:** Specifies the maximum number of error messages that can be displayed by the C++ Test Script Compiler. The default value is 30.
- **Add #line directive into instrumented source file:** This option allows use of *#line* statements in the source code generated by Component Testing for C++. Disable this option in environments where the

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

Testing Options

These options are used for the C++ Test Driver Script.

- **Call breakpoint function on CHECK failure:** Select this option to call a breakpoint function whenever a check failure occurs in an **.otd** Test Driver script. To use this feature, you must set a breakpoint on the function `priv_check_failed ()`, located in the `<target_deployment_port>/lib/priv.c`, file. You can use this option for debugging purposes.
- **Report only failed CHECKs:** Select this option to hide passed tests in the UML Sequence Diagram generated by Component Testing for C++. Only failed checks are displayed. This option also reduces the size of the intermediate trace file.
- **Instances stack size:** This value defines the maximum level for C++ Test Driver Script calls that you expect to reach when running an **.otd** Test Driver script. The C++ Test Driver Script calling stack includes **RUN**, **CALL** and **STUBs**. The default value is 256 and should be large enough for most cases. When using recursive stubs, you may need to increase this value.
- **Display all PRINT arguments in a single note:** Select this option to display only one UML note for all arguments of a **PRINT** statement in the Component Testing for C++ UML Sequence Diagram. This option requires use of a **PRINT** buffer, which uses memory on the target machine. Disable this option if memory on the target is an issue. In this case, Component Testing for C++ generates one UML note for each argument of each **PRINT** statement.
- **PRINT buffer size (bytes):** With the previous option selected, this option defines the size, in bytes, of the buffer devoted to the **PRINT** instructions during the execution. This buffer should be large enough

to handle the complete result of a **PRINT** instruction. You may have to increase this value if your **PRINT** statements contain many arguments, or if arguments are long strings.

Contract Check Options

These options are used by the C++ Contract-Check Script.

- **Call breakpoint function on assertion failure:** Select this option to call a breakpoint function whenever an assertion failure occurs in an **.otc** Contract Check script. To use this feature, you must set a breakpoint on the function `priv_check_failure ()`, located in the `<target_deployment_port>/lib/priv.c`, file. You can use this option, for example, to debug your application when an assertion fails.
- **Report only failed assertions:** Select this option to hide passed assertions in the UML Sequence Diagram generated by Component Testing for C++. Only failed assertions are displayed. This option also reduces the size of the intermediate trace file.
- **Trace unchanged states:** Select this option to report states in UML Sequence Diagram generated by Component Testing for C++ each time states are evaluated. If the option is disabled, states are reported in UML Sequence Diagram only when they change. This affects both trace size and UML Sequence Diagram display size, but has no impact on execution time.
- **Check 'const' methods:** Usually C++ *const* methods are not checked for state changes because they cannot modify a field of the *this* object. Instead, *const* methods are only evaluated once for invariants. In some cases, however, the *this* object may change even if the method is qualified with *const* (by assembler code, or by calling another method that casts the *this* parameter to a non-const type). There may also be pointer fields to objects which logically belong to the object, but the C++ Test Script Compiler will not enforce that these pointed sub-objects are not modified. Select this option only if your code contains such

code implementations.

- **Reentrant object support:** Select this option if your application is multi-threaded and objects are shared by several threads. This ensures atomicity for state evaluation. This option has no effect if multi-thread support is not activated in the Target Deployment Compilation Settings.
- **Enforce 'const' assertions:** When this option is selected, the compiler requires that invariant and state expressions are constant. Disable this option if you do not use the *const* qualifier on methods that are actually constant.

To edit the Component Testing for C++ settings for a node:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select a node in the Project **Explorer** pane.
3. In the Configuration Settings list, expand **Component Testing for C++**.
4. Select **Files, General, Testing** or **Contract Check**.
5. When you have finished, click **OK** to validate the changes.

Component Testing for Java Settings

The Component Testing settings for Java are part of the Configuration Settings dialog box, which allows you to configure settings for each node in your workspace.

The Component Testing for Java node settings lets you to customize the parameters for the Component Testing for Java feature of Test RealTime.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Mode Settings

This area contains parameters to be sent to the linker during the build of the current node.

- **Check Stub:** If selected, Component Testing for Java checks simulated classes. Use **No** to save memory on the target platform.
- **Display Stub:** If the **Check Stub** setting is set to **Yes**, this setting specifies whether to display simulated class information in the Component Testing sequence diagram.

To edit the Component Testing for Java settings for a node:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select a node in the Project **Explorer** pane.
3. In the Configuration Settings list, expand **Component Testing for Java**.
4. Select Files, General, Testing or Contract Check.
5. When you have finished, click **OK** to validate the changes.

System Testing for C Settings

The Test Script Compiler settings are part of the **System Testing** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Test Compiler Settings

- **Virtual Tester Memory Allocation Method:** Allows you to allocate memory to the Virtual Tester for internal data storage.

- **Static** - use global static variables for internal data storage. This allows the Virtual Tester to run on systems that do not support dynamic memory allocation or that have limited execution stacks.
- **Stack** - store internal data in a local variable of the *main()* function. Necessary memory is then allocated on the execution stack.
- **Heap** - allocate memory through a Target Deployment Port dynamic allocation function, which is configurable.
- **Enable Generate Virtual Testers as a Thread (instead of as a process):** Use this setting to on multi-thread platforms.
- **Entry function name:** Specifies the name of a main entry function.
- **Do not share user-defined Virtual Tester global variables:** When using multiple Virtual Tester threads, this setting allows you to specify global variables in the test script that should remain unshared by separate Virtual Tester threads. When selected, multiple instances of a virtual tester can all run in the same process.
- **Trace Buffer Optimization:** See Optimizing Execution Traces.
 - Select **Time stamp only** to generate a normal trace file.
 - Select **Block start/end only** to generate traces for each scenario beginning and end, all events, and for error cases.
 - Select **Errors only** to generate traces only if an error is detected during execution of the application.
- **Circular buffer:** Select this option to activate the Circular Trace Buffer.
- **Trace Buffer size (Kbytes):** This box specifies the size - in kilobytes - of the circular trace buffer. The default setting is 10Kb.

Report Generator Settings

- **Enable Initial and Expected Value Display:** the way in which the values assigned to each variable are displayed in the report. See Initial and Expected Values.
- **Sort by time stamp:** By default, the report is sorted by test script structure blocks. Select this option to force the report to follow a fully chronological order.
- **Display using on-the-fly mode:** Select this option to monitor Virtual Testers in a UML sequence diagram during execution of the test. See On the Fly Tracing.

Target Deployment Port Settings

- **Item bufferized (No = Text bufferized):** This option performs internal compression of trace data. Select this for hard real-time constraints. If you select NO, no compression of trace data is performed.
- **Line buffer size (bytes):** Specifies the size of the line buffer for both options.
- **Enable On-the-fly Trace:** This option enables on-the-fly tracing at Target Deployment Port level.
- **On-the-fly trace buffer size (bytes):** This specifies the size of the trace buffer for on-the-fly tracing. By default the buffer size is 4096 bytes.
- **Timeout for INTERRECV (seconds):** This specifies the timeout associated to inter-tester communications.

To edit the System Testing settings for a node:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select a node in the Project **Explorer** pane.
3. In the Configuration Settings list, expand **System Testing**.
4. Select **Test Script Compiler, Messaging API, Report Generator** or

Target Deployment Port.

5. When you have finished, click **OK** to validate the changes.

Selecting Configurations

Although a project can use multiple Configurations, as well as multiple TDPs, there must always be at least one active Configuration.

The active Configuration affects build options, individual node settings and even wizard behavior. You can switch from one Configuration to another at any time, except during build activity, when the green LED flashes in the Build toolbar.

To switch Configurations:

- From the **Build** toolbar, select the Configuration you wish to use in the **Configuration** box.

Modifying Configurations

Configurations are based on the Target Deployment Ports (TDP) that are specified when you create a new project. In fact, a Configuration contains basic Configuration Settings for a given TDP applied to a project, plus any node-specific overridden settings.

Remember that although a project can use multiple Configurations, as well as multiple TDPs, there must always be at least one active Configuration.

Configuration Settings are a main characteristic of the project and can be individually customized for any single node in the Project Explorer.

To open the Configurations dialog box:

1. From the **Project** menu, select **Configurations**. This opens the **Configurations** dialog.

To create a new Configuration for a Project:

1. In the **Configurations** dialog box, click the **New...** button.
2. Enter a **Name** for the Configuration.
3. Select the **Target Deployment Port** to be used to create the Configuration.
4. Enter the **Hostname**, **Address** and **Port** of the machine on which the Target Deployment Port is to be compiled.
5. Click **OK**.
6. Click **Close**.

To remove a Configuration from a Project:

If you choose to remove a Configuration, all custom settings for that Configuration will be lost.

1. In the **Configurations** dialog box, select the Configuration to be removed.
2. Click the **Remove** button.
3. Click **Yes** to confirm the removal of the Configuration

To copy an existing Configuration:

This can be useful if you want several Configurations, with different custom settings, based on a unique Target Deployment Port.

1. In the **Configurations** dialog box, select an existing Configuration.
2. Click the **Copy To...** button
3. Enter a **Name** for the new Configuration.
4. Click **OK**.

Working with Projects

The project is your main work area in the GUI, as displayed in the **Project Explorer** window.

A project is a tree representation that contains nodes. Each node has its own individual Configuration Settings inherited from its parent node and can be individually executed.

Creating a Group

The Group node is designed to contain several application nodes. This allows you to organize workspace by grouping applications together.

This also allows you to build and run a specific group of application nodes without running the entire workspace.

To create a group node:

1. In the **Project Explorer**, right-click the workspace node or right-click any application node.
2. From the pop-up menu, select **Add Child and Group**.
3. In the **New Group** box, enter the name of the group.
4. Click **OK**.

Manually Creating a Test or Application Node

Application nodes and test nodes (for Test RealTime only) are the main building blocks of your workspace. An application node typically contains the source files required to build the application.

Test nodes contain the source under test, test scripts and any dependency files required for the test.

The preferred method to create an application node is to use the Activity Wizard, which guides you through the entire creation process.

However, if you are re-using existing components, you might want to create an empty application node and manually add its components to the workspace.

To manually add components to the application node.

1. In the **Project Explorer**, right-click the Workspace node or a Group node.
2. From the pop-up menu, select **Add Child** and **Files**.
3. In the File Selector, select the files that you want to add to the application node.
4. Click **Ok**.

Note Before running an application node created with this method, please ensure that all necessary files are present in the application node and that all Configuration Settings have been correctly set.

Creating an External Command Node

External Command nodes are custom nodes that allow you to add a user-defined command line at any point in the project tree.

This is particularly useful when you need to run a custom command line during test execution.

To add an external command to a workspace:

1. In the **Project Explorer**, right-click the node inside which you want to create the test, application or external command node
2. From the pop-up menu, select **Add Child** and **External Command**.
3. To move the node up or down in the workspace, right-click the

external command node and select **Move Up** or **Move Down**.

To specify a command line for the external node:

Once the External Command node has been created, you can specify the command line that it will be carrying in the Configuration Settings dialog box:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Click the **External Command** node.
3. Enter the command in the **Command** box.
4. Click **OK**.

Note External Commands support the GUI Macro Language so that you can send variables from the GUI environment to your command line. See the GUI Macro Language section in the **Reference Manual** for further details.

Importing a Makefile

The GUI offers the ability to create a project from an existing makefile.

The makefile import feature creates a new project, reads the makefile and adds the source files found in the makefile to the project. The project is creating with the default Configuration Settings of the current Target Deployment Port (TDP).

Any other information contained in the makefile, such as compilation options must be entered manually in the Configuration Settings dialog box.

Any environment variables used within the makefile must be valid.

To import files from a makefile:

1. Close any open projects.

- 2 From the File menu, select Import and Import Makefile.
- 3 Use the file selector to locate a valid makefile and click Open.
- 4 Enter a name for the new project and click OK.
- 5 Select the correct Configuration in the Configuration toolbar.
- 6 In the **Project Explorer**, click **Settings**.
- 7 Enter any specific compilation options in the **Build** settings.
- 8 Click **OK**.

Refreshing the Asset Browser

The Asset Browser view of the Project Explorer window analyzes source files and extracts information about file contents (classes, methods, functions, etc...) as well as any dependency files. This capability allows you to navigate through your source files more easily and provides direct access to the components through the Text Editor.

When the automatic file tagging option is selected, the GUI refreshes the file information whenever a change is detected. However, you can use the Refresh Information command to update a single file or the entire project.

Note When many files are involved in the tagging process, the Refresh Information command may take several minutes.

To manually refresh a single file in the Asset Browser:

1. In the **Project Explorer**, select the **Asset Browser** tab.
2. Right-click the file or object that you want to refresh.
3. From the pop-up menu, select **Refresh Information**.

To refresh all project files:

- From the **Build** menu, select **Refresh Asset Browser**, or press the **F9**

key.

To activate or de-activate the automatic refresh:

With the automatic file tagging option, files are automatically refreshed whenever a file is loaded into the workspace or selected in the Project Explorer.

1. From the **Edit** menu, select **Preferences**.
2. Select the **Project** preferences node.
3. Select or clear the **Activate file tagging** option, and then click **OK**.

Deleting a Node

Removing nodes from a project does not actually delete the files, but merely removes them from the Project Explorer's representation.

To delete a node from the Project Explorer:

1. Select one or several nodes that you want to delete.
2. From the **Edit** menu, select **Delete** or press the **Delete** key.

Renaming a Node

Renaming a node in the Project Explorer involves modifying the properties of the node.

To change the name of a node:

1. In the **Project Explorer**, right-click the node that you want to modify.
2. Select **Properties** in the pop-up menu.
3. Change the **Name** of the node.
4. Click **OK**.

Viewing File Properties

You can obtain and change file or node properties by opening the **Properties** window.

To view file properties:

1. Right-click a file in the **Project Explorer**.
2. Select **Properties...** from the pop-up menu.

Excluding a Node from a Build

In some cases, you might want to temporarily exclude one or several nodes from the build process. This can be done directly in the Project Explorer, as described below, or through the Properties window.

Note If you exclude a node that contains child nodes, such as an application node, a group or even a project, none of the contents of the node are executed.

In the Project Explorer, excluded nodes are displayed with a 'x' symbol.

To exclude a node from the build:

1. In the **Project Explorer**, select the node that you want to exclude from the build.
2. In the **Properties** window set the Build property to No.

To cancel the exclusion of a node:

1. In the **Project Explorer**, select the node that you want to exclude from the build.
2. In the **Properties** window set the **Build** property to **No**.

Adding Files to the Project

The Project Explorer centralizes all Project files in a unique location. For Test RealTime to access and analyze source files, they must be accessible from the Project Explorer.

Files are automatically added when you use the Activity Wizard.

To add files to the Project Explorer:

1. In the **Project Explorer**, select the **Object Browser** tab
2. In the **Sort Method** box, select **By Files**.
3. From the **Project** menu, select **Add to Current Project** and **New File...**
4. This opens the file selector. In the file **Type** box, select the type of files that are to be added.
5. Locate and select one or several files to be added, and click **Open**.

The selected files will appear under the Source sections of the Project Explorer.

If you have the Automatic source browsing option enabled, your source files will be analyzed, making their components directly accessible in the Project Explorer.

Selecting Build Options

The GUI allows you to specify the items that will be performed during a build.

The **Stages** section contains the compilation options. In most cases, you will need to select the **All** option to ensure the test is up to date.

The **Runtime Analysis** section allows you to enable debugging and Runtime Analysis features.

To select build options:

1. From the **Build** toolbar, click the black arrow located next to the **Build** button to display the **Build Options** box.
2. Select the Runtime Analysis features (Memory Profiling, Performance Profiling, Code Coverage and Runtime Tracing) and build options to use them on the current node.

Building and Running a Node

You build and execute workspace nodes by using the **Build** button on the Build toolbar. The build process compiles, links, deploys, executes, and then retrieves results. However, you first have to specify the various build options.

You can use the **Build** command to execute any application node, as well as a single specific source file, a group node or even the whole project.

Note When you run the **Build** command, all open files are saved. This means that any unsaved changes will actually be taken into account for the build.

Before building a node:

1. Select the correct Configuration for your target in the build toolbar.
2. Exclude any temporarily unwanted nodes from the build.
3. Select the build options for the test.
4. If necessary, clean up files left by any previous executions by clicking the **Clean** button.

To build and execute the node:

1. From the Build toolbar, click the **Build** button.
2. During run-time, the Build Clock indicates the execution time and the green LED flashes. The Project Explorer displays a check mark next to

each item to mark progression of the build process.

3. When the build process is finished, you can view the related test reports.

To stop the execution:

- If you want to stop the execution of a node before it finishes, or if the application does not stop by itself, click the **Stop Build/Execution** button.

Cleaning Up Generated Files

In some cases, you might want to delete any files created by a build execution, such as to perform the build process in a clean environment or when you are running short of disk space.

Use the **Clean All Generated Files** command to do this.

To clean your workspace:

1. From the Build toolbar, click the **Clean All Generated Files** button.

Creating a Source File Folder

The Project Explorer Asset Browser provides a convenient way of viewing the source files in your project.

To make this even more convenient, you can create custom folders to accommodate any file types. This makes navigation through your source files even easier.

Note The Asset Browser provides a virtual navigation interface. The actual files do not change location. Use the **Properties Window** to view the actual file locations.

To create a custom folder:

1. In the **Asset Browser**, select the **By Files** sort method.
2. Right-click on an existing folder.
3. From the popup menu, select **New Folder...**
4. Enter a name for the new folder and a file filter for the desired file type.

Importing a Data Table (.csv File)

Rational Test RealTime Component Testing for C, Ada and C++ provide the ability to import **.csv** table files and to turn these into standard header files. Once included in your **.ptu** or **.otd** test script, this data can be used by the test driver script or the application under test.

Such **.csv** files can be produced by third party applications, for example Microsoft Excel.

To import a .csv file into a test node:

1. From the **Project Explorer**, right click an existing test node.
2. From the pop-up menu, select **Add File...**
3. Locate and select the **.csv** file and click **OK**.
4. By default, added files are excluded from the build. Click the **Excluded** marker to allow the file to be built.

CSV File Format

The input file format must respect the following formatting rules:

- The default separator is a semicolon character (";"). This can be changed in the General Settings dialog box.
- The first line contains the names of the variable arrays
- The second line optionally specifies the data type: string, char or **int**,

long, **float** and **double**, which can be **signed** or **unsigned**. if this information is not specified, then **int** is assumed by default.

- Each following line contains the data for the corresponding array
- When a blank file is encountered, an end of array is assumed. Any further values for that array will be ignored.

When the test node is built, Test RealTime produces a *<filename>.h* header file, where *<filename>* is based on the name of the input *<filename>.csv* file.

Use the arrays produced by the *.csv* file by including *<filename>.h* into your test script or source code.

Example

This is an example of a valid **table.csv** data table:

```
var_A;var_B;var_C
int;signed int;float
12;34;45.2345
14;2;3.142
;-5;0
```

This produces the following corresponding **table.h** file:

```
int var_A[]={12,14};
signed int var_B[]={34,2,-5};
float var_C[]={45.2345,3.142,0};
```

Opening a Report

Because of the links between the various views of the GUI, there are many ways of opening a test or runtime analysis report in Test RealTime. The most common ones are described here.

To open a report from the Project Explorer:

1. Execute your test with the **Build** command.
2. Right-click the test node.

3. From the pop-up menu, select **View Report** and then the appropriate report.

Note Reports cannot be viewed before the test has been executed.

To manually open a report file:

1. From the **File** menu, select **Open...** or click the **Open** icon in the Standard toolbar.
2. In the **Type** box of the File Selector, select the appropriate file type.
3. Locate and select the report files that you want to open.
4. Click **OK**.

Note Some reports require opening several files. For instance, when manually opening a UML sequence diagram, you must select at the complete set of **.tsf** files as well as the **.tdf** file generated at the same time. A mismatch in **.tsf** and **.tdf** files would result in erroneous tracing of the UML sequence diagram.

Report Viewers

The GUI opens the report viewer adapted to the type of report:

- The UML/SD Viewer displays UML sequence diagram reports.
- The Report Viewer displays test reports for Test RealTime and Memory Profiling reports for Java.
- The Code Coverage Viewer displays code coverage reports.
- The Memory Profiling Viewer and Performance Profiling Viewer display Memory Profiling for C, Ada and C++ and Performance Profiling results.

Debug Mode

The Debug option allows you to build and execute your application under a debugger.

The debugger must be configured in the Target Deployment Port.

Note: Before running in Debug mode you must change the Compilation and Link Configuration Settings to support Debug mode. For example set the **-g** option with most Linux compilers.

Editing Preferences

Rational TestRealTime has many **Preference** settings that allow you to configure various components of the graphical user interface.

To edit product preferences:

1. From the **Edit** menu, select **Preferences**.
2. In the tree-view, select the component that you want to configure.
3. Make any changes to the preferences.
4. Click **OK**.

Project Preferences

The **Project Preferences** dialog box lets you set parameters for the Test RealTime project.

In the **Preferences** dialog box, select **Project** to change the project preferences.

- **Automatic file tagging:** Select this option to activate the Project Explorer's automatic parsing mode, in which all source code and script components are automatically listed. If disabled, you will have to

manually refresh the File View each time you modify the structure of a file.

- **Calculate static metrics:** Select this option to ensure that static metrics are recalculated whenever a file is added, modified or refreshed in the Project Explorer window.

Connection Preferences

The **Preferences** dialog box allows you to customize the Test RealTime GUI.

The **Connections** node of the **Preferences** dialog box lets you set the network parameters for the graphical user interface.

1. In the **Preferences** dialog box, select the **General** node and **Connections**.
 - **Allow remote connections:** This allows external commands and tools to send messages to the GUI over a network. For example, this enables the Runtime Tracing on-the-fly capability on remote hosts.
 - For information only, the **Current TCP/IP port** is automatically selected by GUI.
2. Click **OK** to apply your changes.

Activity Wizards

The Start Page provides with a full set of activity wizards to help you get started with a new project or activity.

To start a new activity wizard:

1. From the **Start Page**, click **New Activities**
2. Select the activity of your choice.

New Project Wizard

When Test RealTime starts, the Start page offers to either open an existing project or create a new project. The New Project wizard creates a brand new project.

To create a new project:

1. From the **Start Page**, select **New Project**.
2. In the **Project Name**, enter a name for the project.
3. In the **Location** box, change the default directory if necessary and click **Next** to continue.
4. Select one or several Target Deployment Ports for the new project. The Wizard creates a Configuration based on each selected Target Deployment Port. Later, when working with the project, any changes are made to the Configuration Settings, not to the Target Deployment Port itself.
5. Click the **Set as Active** button to set the current TDP. The active port is the default Configuration to be used in your project.
6. Click **Finish**

Once your project has been created, the wizard opens the **Activities** page.

Runtime Analysis Wizard

The Runtime Analysis Wizard helps you create a new application node in the Project Explorer. Basically, an application node represents the build of your C, C++, Ada or Java source code, which is very similar to most other integrated development environments (IDE). You can actually use this graphical user interface as your primary IDE.

With Test RealTime, you simply add to this application node the options required to run any of the following runtime analysis features:

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

To create an application node with the Runtime Analysis Wizard:

1. Use the **Start Page** or the **File** menu to open or create a project.
2. Ensure that the correct Configuration is selected in the **Configuration** box.
3. On the **Start Page**, select **Activities** and choose the **Runtime Analysis** activity.
4. On the **Application Files** page, click **Add** to add your source code files to the list. This opens a file selector.
Use the **Move Up** and **Move Down** buttons to change the order in which files appear in the application node, and subsequently are compiled. Use the **Remove** button to remove files from the selection.

Click **Next** to continue.

5. Select the C procedures and functions, C++ or Java classes or Ada units that you want to analyze.
Use the **Select File** and **Deselect File** buttons to specify the files that contain the components that you want to analyze. The **Select All** and **Deselect All** buttons to select or clear all components.
Click **Next** to continue.
6. If you are creating a Java application node, set the basic settings that are required for the program to compile:
 - **Class path:** Click the ... button to create or modify the Class Path parameter for the JVM
 - **Java main class:** Select the name of the main class
 - **Jar creation:** Specifies whether to build an optional **.jar** file, as well as

the basic **.jar** related options

Click **Next** to continue.

6. Enter a name for the application node.
By default, the new application node inherits Configuration Settings from the current project. If necessary, click **Settings...** to access the Configuration Settings dialog box. This allows you to change any particular settings for the new application node as well as its contents.

Click **Next** to continue.

7. In the **Summary** page, check that all the parameters are correct, and click **Finish**.

The wizard creates an application node that includes all of the associated source files.

You can now select your build options to apply any of the runtime analysis features to the application under analysis.

Component Testing Wizard

The Component Testing Wizard helps you create a new Component Testing test node in your project for C, C++, Ada and Java

For each script type, the wizard analyzes the source code under test to extract class and method information and will produce a corresponding test script template using the following test script types:

- C Test Script Language
- C++ Test Driver Script Language
- C++ Contract-Check Language
- Ada Test Script Language

- JUnit Test Harness

You use the generated test script template to elaborate your own test cases.

You can later add to this test node any of the runtime analysis features included in Test RealTime.

The first decision you have to make with the Component Testing wizard is to select the level of automation for the test generation process. The wizard offers several modes:

- **Single Mode:** In C and Ada, this mode creates one test node for each source file under test. In C++ and Java, it creates one test node for all selected source code component.
- **Multiple Mode:** This creates a single test node for each selected source code component.
Single Mode and Multiple Mode are only available when several source files or units are selected.
- **Typical Mode:** In this mode, the wizard automatically stubs all dependency classes or functions that are found in the source under test. This is faster when producing many unit tests.
- **Expert Mode:** This mode allowing you to manually drive generation of the test harness. This provides more flexibility in sophisticated software architectures.

To run the Component Testing Wizard from the Start Page:

1. Use the **Start Page** or the **File** menu to open or create a project.
2. Ensure that the correct Configuration is selected in the **Configuration** box. The selected programming language impacts the type of Component Testing test node to be created.
3. On the **Start Page**, select **Activities** and choose the **Component Testing** activity.
4. The **Application Files** page opens. Use the **Add** and **Remove** buttons

to build a list of source files to add to your project. The **Up** and **Down** buttons allow you to select the order in which each file is to be compiled. The **Project Settings** button allows you to override the default Configuration Settings, and recalculates the testability metrics.

5. The **Components Under Test** page lists various static testability metrics for the selected source files. Use this list to choose the source code units (packages, classes, functions) that require testing. Click **Metrics Diagram** to select the units under test from a graph representation.

To run the Component Testing Wizard on a source code component:

1. Use the **Start Page** or the **File** menu to open or create a project.
2. Ensure that the correct Configuration is selected in the **Configuration** box. The selected programming language impacts the type of Component Testing test node to be created.
3. In the **Project Explorer**, select the **Object Browser** tab.
4. Right click an object, package or source file.
5. From the popup menu, select **Test...**

To create a test node with the Component Testing wizard:

1. On the **Test Mode** page select **Expert** or **Typical Mode** and click **Next>** to continue.
2. If you are in Expert Mode, select the simulated, integrated (C and Ada) and Additional files that are required to compile your application. See the following sections for more information:
 - Integrated, Simulated and Additional Files for C and Ada
 - Simulated, Additional and Included Files for C++
 - Simulated and Additional Classes for Java

Click **Next>** to continue.

3. Enter a **Test Name**. This is the name of the new test node.
Click **Advanced Options** to modify the default test generation options. These advanced options are those offered by the **Source Code Parser** command line of the selected language. See the **Test RealTime Reference Manual** for more information.
Click **Settings** to set the Configuration Settings. You can always modify the test node Configuration Settings later if necessary, from the Project Explorer.
Click **Next>** to continue.
4. Review the **Summary**. This page provides a summary of the selected options and the files that are to be generated by the wizard.
Click **Next>** to create the test node based on this information.
5. The Test G
Click **Finish** to quit the Component Testing Wizard.
6. Generation Result page displays progression of the test node creation process.
The wizard creates an application node that includes all of the associated source files.

You can now complete your Component Testing test scripts in the Text Editor. Refer to the **Test RealTime Reference Manual** for information about the actual language semantics.

System Testing Wizard

The System Testing Wizard helps you create a new System Testing test node in your project.

Basically, a System Testing node contains a blank test script as well as a set of Virtual Testers for message-based testing.

You can later add to this test node any of the runtime analysis features included with Test RealTime.

To create a System Testing node:

1. Enter the name of the new System Testing node.
 2. Enter the Test Script Selection information.

First, select whether you want to create a blank **.pts** test script file, or if you want to reuse an existing test script. In both cases you will need to enter a name for the **.pts** test script.

This page requires that you select the source files that are used to build your application among the source files that are currently in your workspace.

Next, use the **Add** and **Remove** buttons to build a list of interface files. The Interface Files List must contain files that define the communication routines of your application.

Click **Next** to continue.
 3. Specify the *include* directories for your application.

Use the **Add** and **Remove** buttons to build a list of *include* directories. These are all the directories that contain files that are *included* by your application's source code. Use the **Up** and **Down** buttons to indicate the order in which they are searched.

Click **Next** to continue. If you chose to create a new **.pts** test script, this brings you straight to step 6.
 4. Create a set of Virtual Testers.

See Configuring Virtual Testers for information about this page. If necessary, click the Configure Settings button to access the Configuration Settings for the selected Virtual Tester node.

Click **Next** to continue.
 5. Deploy the Virtual Testers onto your target hosts.

See Deploying Virtual Testers for information about this page.

Click **Next** to continue.
 6. Perform a quick review of the options in the **Summary Page**, and use the **<Back** button if necessary to make any changes.
- **Test Script File:** indicates the name of the **.pts** test script

- **Interface Files:** lists the interface files defining the communication routines of your application.
 - **Included Directories:** lists the directories containing files *included* by your application.
 - **Virtual Testers:** lists the Virtual Tester that are to be deployed by the System Testing node.
7. Click the **Finish** button to launch the generation of the System Testing node with the corresponding Virtual Testers.

The wizard creates a test node with the associated test scripts. The test node appears in the Project Explorer.

If you chose to create a new **.pts** test script, you can now write your System Testing test script in the Text Editor and then configure and deploy your Virtual Testers.

Refer to the **Test RealTime Reference Manual** for information about the System Testing Language (STL).

Metrics Diagram

As part of the Component Testing wizard, Test RealTime provides static testability metrics to help you pinpoint the critical components of your application. You can use these static metrics to prioritize your test efforts.

The graph displays a simple two-axis plot based on the static metrics calculated by the wizard. The actual metrics on each axis can be changed in the Metrics Diagram Options dialog box.

Each unit (function, package or class, depending on the current Configuration language) is represented by a checkbox located at the intersection of the selected testability metrics values.

Move the mouse pointer over a checkbox to display a tooltip with the names of the associated units. To test a unit, select the corresponding checkbox.

Test RealTime also provides a Static Metrics Viewer, which is independent from the Component Testing wizard and can be accessed at any time.

To access the wizard Metrics Diagram:

1. From the Start Page, run the Component Testing wizard.
2. From the **Components under Test** page, click **Metrics Diagram**.

To select a unit for test:

1. If necessary, click **Options** to set the two most relevant metrics for your application. This displays each unit at the intersection point of the two values.
2. Move the mouse pointer over a checkbox to display a tooltip with the name of the unit.
3. Select the most relevant units to test. Units under test are displayed in the list box.
4. Click **OK** to validate the selection.

Advanced Options

The **Advanced Options** dialog box allows you to specify a series of advanced test generation parameters in the Component Testing wizard. In most cases, you can leave the default values.

The actual options available in this dialog box depend on the programming language of the current Configuration:

- C or Ada
- C++

- Java

Advanced options may override the default Configuration Settings for the current project. In this case, the overridden settings are applied to the generated test node.

Component Testing for C and Ada

The following advanced options are available in the Component Testing wizard with a C or Ada Target Deployment Port:

- **Tested file:** name of the source file under test
- **Test script and path:** location and name of the generated Ada test script template
- **Test static/private data or functions:** specifies whether the file under test is included in a #include statement.
- **Additional options:** allows you to add specific command line options for the C or Ada Source Code Parser. See the **Command Line Reference** section in the **Rational Test RealTime Reference Manual** for further information.

Component Testing for C++

The following advanced options are available in the Component Testing wizard for C++:

- **Tested file:** name of the source file under test.
- **Test driver path and filename:** specifies whether an **.otd** test driver script is to be generated.
- **Contract-Check script:** specifies whether an **.otc** Contract Check driver script is to be generated.
- **Test script and path:** location and name of the generated **.otd** test driver script template.

- **Directory for Contract-Check script files:** sets the location where the **.otc** Contract Check script files are created.
- **Additional options:** allows you to add specific command line options for the C++ Source Code Parser. See the Line Command section in the **Rational Test RealTime Reference Manual** for further information.
- **Ignore #line directive:** by default, the Test Generation Wizard analyzes **#line** directives, although use of preprocessed files with Component Testing for C++ is not recommended. Select this option when **#line** directives should be ignored.
- **Test union and struct as class:** tells the Test Generation Wizard to consider classes defined with the **struct** or **union** keyword as candidate classes. This option is only available if the **auto-select candidate classes** was selected on the **File and Classes under Test** page.
- **Test each template instance:** tells the wizard to generate C++ Test Script Language code for each instance of a template class. If this option is selected, there must be template class instances in the source file under test. By default, the Test Generation Wizard generates a single portion of C++ Test Script Language code for a template class.
- **Overwrite previous test scripts:** tells the wizard to overwrite any previously generated **.otc** or **.otd** test scripts. If this option is not selected, no changes will be made to any existing **.otc** or **.otd** test scripts.
- **Path for included header files:** specifies how include file names must be analyzed.
- Select **Relative** for relative filenames.
- Select **Absolute** for absolute filenames.
- Select **Copy** to use include the path as specified.
- **Included files:** use the Add and Remove buttons to add and remove directories in the list. The include file list used by the Component

Testing wizard are kept in the generated test node settings.

Component Testing for Java

The following advanced options are available in the Component Testing wizard for C++:

- **Test driver name:** name of the generated Java test script template.
- **Directory for generated files:** sets the location of generated files.
- **Testing framework:** allows you to override the TDP setting to the J2SE or J2ME framework.
- **Test class prefix:** specifies the prefix for test class names.

Command Line Interface

5

Rational Test RealTime was designed ground-up to provide seamless integration with your development process. To achieve this versatility, the entire set of features are available as command line tools.

In most cases when a CLI is necessary, the easiest method is to develop, set up and configure your project in the graphical user interface and to use the command line to launch the GUI and run the corresponding project node.

The complete syntax and command line reference for each tool is covered in the **Reference Manual**.

Running a Node from the Command Line

Although the product contains a full series of command line tools, it is usually much easier to create and configure your runtime analysis specifications inside the graphical user interface (GUI). The CLI would then be used to simply launch the GUI with a project or project node as a parameter.

By doing this, you combine the ease and simplicity of the GUI with the ability to execute project nodes from a CLI.

Note This functionality can be used to execute any node in a project, including group nodes, application nodes, test nodes or the entire project.

To run a specific node from a command line:

1. Set up and configure your project in the GUI.
2. Save your project and close the GUI.
3. Type the following command
studio -r <node>{[.<node>]} <project_file>

where <node> is the node to be executed and <project> is the **.rtp** project file.

The <node> hierarchy must be specified from the highest node in the project (excluding the actual project node) to the target node to be executed, with periods ('.') separating each item:

<node>{[.<node>]}

Example

The following command opens the **project.rtp** project in the GUI, and runs the *app2* application node, located in *group1* of the sub-project *subproject1*:

```
studio -r subproject1.group1.app2 project.rtp
```

Command Line Runtime Analysis for C and C++

The runtime analysis features for C and C++ include:

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

These features use Source Code Insertion (SCI) technology. When analyzing C and C++ code, the easiest way to implement SCI features from the

command line is to use the C and C++ Instrumentation Launcher.

The Instrumentation Launcher is designed to fit directly into your compilation sequence; simply add the **attolcc** command in front of your usual compilation or link command line.

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

To perform runtime analysis on C or C++ source code:

1. First, set up the necessary environment variables. See Setting Environment Variables.
2. Edit your usual makefile with the following command line:

```
attolcc [-options] [--settings] -- <compiler command line>
```

Where *<compiler command line>* is the command that you usually invoke to build your application.

For example:

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

Please refer to the **Instrumentation Launcher** section of the **Reference Manual** for information on attolcc options and settings, or type **attolcc --help** on the command line.

3. After execution of your application, in order to process SCI dump information (i.e. the runtime analysis results), you need to separate the single output file into separate, feature-specific, result files. See Splitting the SCI Dump File.
4. Finally, run the graphical user interface to view the reports.

Command Line Runtime Analysis for Java

The runtime analysis features for Java covered in this section include:

- Performance Profiling
- Code Coverage
- Runtime Tracing

These features use Source Code Insertion (SCI) technology. Memory Profiling for Java relies on JVMPI instead of SCI technology. Please refer to the JVMPI Agent section of the **Reference Manual**.

The easiest way to implement SCI from the command line is to use the Java Instrumentation Launcher: *javic*. The product provides two methods for use of *javic*:

- **Java Instrumentation Launcher:** designed to fit directly into your compilation sequence; simply add the **javic** command in front of your usual compilation or link command line
- **Java Instrumentation Launcher for Ant:** this integrates *javic* with the Apache Jakarta Ant utility

For details of command line usage and option syntax, see the **Reference Manual**.

To perform runtime analysis on Java source code:

1. First, set up the necessary environment variables. See Setting Environment Variables.
2. Edit your usual makefile by adding the Java Instrumentation Launcher to the command line:

```
javic [-options] -- <compiler command line>
```

Where *<compiler command line>* is the command that you usually invoke to build your application.

Please refer to the Instrumentation Launcher section of the **Reference Manual** for information on the options and settings.

3. After execution, to obtain the final test results, as well as any SCI dump information, you need to separate the output file into separate result files. See Splitting the SCI Dump File.
4. Finally, run the graphical user interface to view the test reports.

Command Line Component Testing for C, Ada and C++

Use Component Testing for C and Ada and Component Testing for C++ to test individual components of your C, C++ and Ada source code.

To perform component testing on C, C++ or Ada source code:

1. First, set up the necessary environment variables. See Setting Environment Variables.
2. Generate a set of test script templates based on your source files by using the Source Code Parser. See corresponding **Source Code Parser** command line section in the **Reference Manual**.
3. Use the generated **.ptu**, **.otc** or **.otd** templates to write a test script. See the **Reference Manual** for test script syntax.
4. If you are using an **.otc** Contract Check script, set up an **options.h** header file. See Preparing an Options Header File.
5. Compile the generated test harness source file. See Compiling the Test Harness
6. If you are using any of the runtime analysis features, instrument and compile the source code. See Instrumenting and Compiling the Source Code.
If not, simply compile your source code with your usual compiler.

7. Set up the TDP configuration file, called **product.h**. See Preparing a Products Header File.
8. Compile the TDP Library. See Compiling the TDP Library.
9. Link the compiled files together to create an executable test binary. See Linking the Application.
10. Execute the test binary. See Running the Test Harness or Application.
11. After execution, to obtain the final test results, as well as any SCI dump information, you need to separate the output file into separate result files. See Splitting the SCI Dump File.
12. Run the **Report Generator** to produce a test report. See the corresponding **Report Generator** command line section in the **Reference Manual**.
13. Finally, run the Graphical User Interface to view the test reports.

Command Line Component Testing for Java

Use Component Testing for Java to test individual components of your Java source code.

To perform component testing on Java source code:

1. First, set up the necessary environment variables. See Setting Environment Variables.
2. Generate a set of test script templates based on your source files by using the Source Code Parser. See corresponding Source Code Parser command line section in the Reference Manual.
3. Use the generated **.ptu**, **.otc** or **.otd** templates to write a test script. See the **Reference Manual** for test script syntax.
4. Compile the generated test harness source file. See Compiling the Test Harness.

5. If you are using any of the runtime analysis features, instrument and compile the source code. See Instrumenting and Compiling the Source Code.
If not, simply compile your source code with your usual compiler.
6. Set up the TDP configuration file **Products.java**. See Preparing a Products Header File.
7. Compile the TDP Library. See Compiling the TDP Library.
8. Link the compiled files together to create an executable test binary. See Linking the Application.
9. Execute the test binary. See Running the Test Harness or Application.
10. After execution, to obtain the final test results, as well as any SCI dump information, you need to separate the output file into separate result files. See Splitting the SCI Dump File.
11. Run the **Report Generator** to produce a test report. See the corresponding **Report Generator** command line section in the **Reference Manual**.
12. Finally, run the Graphical User Interface to view the test reports.

Command Line System Testing for C

Use System Testing to test message-based systems and subsystems written in C.

To perform message based testing on a system:

1. First, set up the necessary environment variables. See Setting Environment Variables.
2. Write a System Testing **.pts** test script. See the **Reference Manual** for test script syntax.
3. Write a System Testing **.spv** supervisor script. See the **Reference**

Manual for test script syntax.

Note Manually created supervisor scripts may be overwritten by the Test RealTime graphical user interface.

4. Compile the generated test harness source file. See Compiling the Test Harness.
5. If you are using any of the runtime analysis features, instrument and compile the source code. See Instrumenting and Compiling the Source Code.
If not, simply compile your source code with your usual compiler.
6. Set up the TDP configuration file, called **product.h**. See Preparing a Products Header File.
7. Compile the TDP Library. See Compiling the TDP Library.
8. Link the compiled files together to create an executable test binary. See Linking the Application.
9. Ensure that the System Testing agents are running on all remote target hosts. See Installing System Testing Agents.
10. Run the supervisor script on the supervisor machine (the machine running Test RealTime) with the following command:

```
atsspv <supervisor.spv>
```

where supervisor is the name of the **.spv** supervisor script.

11. Run the **System Testing Report Generator** to produce a test report. See the corresponding **Report Generator** command line section in the **Reference Manual**.
12. Finally, launch the Graphical User Interface to view the test reports.

Command Line Tasks

Setting Environment Variables

The command line interface (CLI) tools require several environment variables to be set.

These variables determine, for example, the Target Deployment Port (TDP) that you are going to use. The available TDPs are located in the product installation directory, under **targets**. Each TDP is contained in its own sub-directory.

Prior to running any of the CLI tools, the following environment variables must be set:

- **TESTRTDIR** indicates the installation directory of the product
- **ATLTGT** and **ATUTGT** specify the location of the current TDP: **\$TESTRTDIR/targets/<tdp>**, where *<tdp>* is the name of the TDP.
- **PATH** must include an entry to **\$TESTRTDIR/bin/<platform>/<os>**, where *<platform>* is the hardware platform and *<os>* is the current operating system.

You must also add the product installation **bin** directory to your **PATH**.

Note Some command-line tools may require additional environment variables. See the chapters dedicated to each command in the **Reference Manual** section.

Test RealTime

If you are running Rational Test RealTime, the following additional environment variables must be set:

- **ATUDIR** for Component Testing, points to **\$TESTRTDIR/lib**
- **ATS_DIR**, for System Testing, points to **\$TESTRTDIR/bin/<platform>/<os>**, where *<platform>* is the hardware platform and *<os>* is the current operating system.

Library Paths

UNIX platforms require the following additional environment variable:

- **On Solaris and Linux platforms:** **LD_LIBRARY_PATH** points to **\$TESTRTDIR/lib/<platform>/<os>**
- **On HP-UX platforms:** **SH_LIB** points to **\$TESTRTDIR/lib/<platform>/<os>**
- **On AIX platforms:** **LIB_PATH** points to **\$TESTRTDIR/lib/<platform>/<os>**

where *<platform>* is the hardware platform and *<os>* is the current operating system.

Example

The following example shows how to set these variables for Test RealTime with a **sh** shell on a Suse Linux system. The selected Target Deployment Port is ***clinuxgnu***.

```
TESTRTDIR=/opt/Rational/TestRealTime.v2002R2
ATCDIR=$TESTRTDIR/bin/intel/linux_suse
ATUDIR=$TESTRTDIR/lib
ATS_DIR=$TESTRTDIR/bin/intel/linux_suse
ATLTGT=$TESTRTDIR/targets/clinuxgnu
ATUTGT=$TESTRTDIR/targets/clinuxgnu
LD_LIBRARY_PATH=$TESTRTDIR/lib/intel/linux_suse
PATH=$TESTRTDIR/bin/intel/linux_suse:$PATH
export TESTRTDIR
export ATCDIR
export ATUDIR
export ATS_DIR
export ATLTGT
export ATUTGT
```



```
export LD_LIBRARY_PATH
export PATH
```

Preparing an Options Header File

This step is necessary if you are using:

- System Testing for C
- **.otc** Contract Check feature

Before you can compile a generated source file, you must set up a file named **options.h**, which contains compilation parameters for such files.

How to prepare the options.h file:

1. From the sub-directory lib of the selected Target Deployment Port, copy a file named **options_model.h** to a directory of your choice, and rename it to **options.h**.

The directory of your choice may be the directory where the generated source files or instrumented source files are located.

2. Open **options.h** in a text editor and add the following *define* at the beginning of the file:

```
#define ATL_WITHOUT_STUDIO
```

3. Make any necessary changes by adjusting the corresponding macros in the file.

The **options_model.h** file is self-documented, and you can adjust every macro to one of the values listed. Each macro is set to a default value, so you can keep everything unchanged if you don't know how to set them.

Take note of the directory where this file is stored, you will need it in order to compile the generated or instrumented source files.

Preparing a Products Header File

Before you can compile the TDP library source files, you must set up a file named **products.h** for C and C++ or **Products.java** for Java. This file contains the options that describe how the TDP library files are to be compiled.

To set up a products header file

1. For C and C++, copy the **product_model.h** file from the **lib** sub-directory of the current Target Deployment Port to a directory of your choice, and rename it to **products.h**.
2. The directory of your choice may be the directory where the generated source files or instrumented source files are located.
3. For Java, copy the **Products_defaults.java.txt** file from the **lib** sub-directory of the current Target Deployment Port to **com/rational/test/Products.java**.
4. Open **products.h** or **Products.java** in a text editor and add the following *define* at the beginning of the file:

```
#define ATL_WITHOUT_STUDIO
```
- 5.. Make any necessary changes by adjusting the corresponding macros in the file.

The **product_model.h** file is self-documented, and you can adjust every macro to one of the values listed. Each macro is set to a default value, so you can keep everything unchanged if you don't know how to set them.

Note Pay attention to correctly set the macros starting with **USE_**, because these macros set which features of Test RealTime you are using. Certain combinations are not allowed, such as using several test features simultaneously.

Ensure that the **ATL_TRACES_FILE** macro correctly specifies the name of the trace file which will be produced during the execution. If you are using

Component Testing, this value may be overridden by a Test Script Compiler command line option.

Take note of the directory where this file is stored, you will need it in order to compile the generated or instrumented source files.

Instrumenting and Compiling the Source Code

The runtime analysis features (Memory Profiling, Performance Profiling, Code Coverage and Runtime Tracing) as well as Component Testing for C++ Contract Check all use SCI instrumentation technology to insert analysis and SCI dump routines into your source code.

Requirements

Before compiling an SCI-instrumented source file, you must make sure that:

- A working C, C++, Java or Ada compiler is installed on your system
- If you use Component Testing for C++, you have prepared a valid **options.h** file
- If you compile on a target different from the host where the generated file has been produced, the instrumented file must have been produced using option **-NOPATH**, and the sub-directory lib of the selected Target Deployment Port directory must be copied onto the target.

There are two alternatives to instrument and compile your source code:

- Using the Instrumentation Launcher in your standard makefile
- Using the Instrumentor and Compiler separately.

Instrumentation Launcher

The Instrumentation Launcher replaces your actual compiler command in your makefiles. This launcher transparently takes care of source code

preprocessing, instrumentation and compiling.

See the command line information for the Instrumentation Launcher in the **Reference Manual**.

Instrumentation and Compilation

Alternatively, you can use the actual Instrumentor command line tools to instrument the source files.

See the command line information for each Instrumentor in the **Reference Manual**.

If you are compiling on a different target, you must copy the TDP `/lib` directory over to that target.

Add to the include search path the `/lib` sub-directory that you have copied onto the target. In C and C++, use the `-I` compiler option. In Java, add the directory to the CLASSPATH.

After this, simply compile the instrumented source file with your compiler.

Compiling the TDP Library

Before you can link your test harness or your instrumented application, you must compile the Target Deployment Port library. This section describes how to do this.

Requirements

To compile the Target Deployment Port library, make sure that:

- A working C or C++ Test Script Compiler is installed on your system
- You have prepared a valid Products file

Compilation

Depending on the language of your source file:

- For C: compile the **TP.c** file
- For C++: compile the **TP.cpp** file
- For Ada: compile the contents of the **/lib** directory
- For Java: set the **CLASSPATH** to the TDP **/lib** directory

Do not forget to add to the *include* search path the directory where the **products.h** or **Products.java** file is located (usually with option **-I** or **/I**, depending on the compiler).

Configuration Settings

A wide variety of compilation flags can be used by the command line tools, allowing you to select sub-components of the application under test. These flags are equivalent to the Test Configuration Settings dialog box of the graphical user interface and are covered in the **Reference Manual**.

Default settings are contained in the following Perl script. You can use this file to define your own customized configuration settings.

```
<InstallDir>/lib/scripts/BatchCCDefaults.pl
```

To run this script, type the following command:

```
$TESTRTDIR/bin/<cpu>/<os>/perl -I$TESTRTDIR/lib/perl  
$TESTRTDIR/lib/scripts/TDPBatchCC.pl <my_env.pl>
```

where *<cpu>* is the architecture platform of the machine, *<os>* is the operating system, and *<my_env.pl>* is your customized copy of the **BatchCCDefaults.pl** file

The **TESTRTDIR** and **ATLTGT** environment variables must have been previously set.

Compiling the Test Harness

Each of the test compilers converts a test script into test harness source code. This section explains how to compile the test harness source file.

Requirements

In order to compile a generated source file, you must check that:

- A working C, C++ or Ada compiler is installed on your system
- If you are using System Testing, you have prepared a valid **options.h** file
- If you are compiling on a target different from the host where the file was generated, the generated file must have been produced using the **-NOPATH** option (available with every test compiler), and the **/lib** sub-directory of the Target Deployment Port directory must be copied onto the target.

Compilation

If you are using Component Testing, System Testing or Component Testing for C++ alone without any of the runtime analysis features, then simply compile the generated test harness source file with your C or C++ compiler.

If you are compiling on a remote target, do not forget to add to the *include* search path the **/lib** sub-directory that you have copied onto the target.

If you are using SCI instrumentation features (Memory Profiling, Performance Profiling, Code Coverage, Runtime Tracing and C++ **.otc** contract check), use the specific command line options for the Instrumentor in the **Reference Manual**.

Linking the Application

Once you have compiled all your source files, you need to link them to build an executable. This section describes linkage specifics when using a test or runtime analysis feature.

Requirements

In order to compile an instrumented source file, you must check that:

- A working C, C++ or Ada linker is installed on your system
- You have compiled every source file, including any instrumented source files, of your application under test
- If using a Component Testing for C, Ada or C++, or System Testing, you have compiled the test harness.
- You have compiled the Target Deployment Port library.

Linking

If you are using only runtime analysis feature (Runtime Tracing, Code Coverage, Memory Profiling, Performance Profiling, C++ Contract Check), you just have to add the Target Deployment Port library object to the object files linked together. If you are using a test feature, you also have to add the tester object to the linked files.

Running the Test Harness or Application

Once you have produced a binary tester or instrumented application, you want to run it in order to obtain test or SCI analysis information.

By default, the generated SCI dump file is named **atlout.spt**.

To run the test application binary:

1. Check that the current directory is correct, relatively to the previously specified trace file, if the trace files was specified with a relative path.
2. Run the binary. When the application terminates, the trace file should be available.

Splitting the Trace Dump File

When you use several features together, the executable produces a multiplexed trace file, containing several outputs targeting different features from Test RealTime. By default, the trace file is named **atlout.spt**.

Requirements

In most cases, you must split the **atlout.spt** trace file into several files for use with each particular Report Generator or the product GUI.

To do this, you must have a working *perl* interpreter. You can use the *perl* interpreter provided with the product in the **/bin** directory.

To split the trace file:

- Use the **atlsplit** tool supplied in the **/bin** directory of Test RealTime:

```
atlsplit atlout.spt
```

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

- **.rio test result files:** process with a 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

Troubleshooting Command Line Usage

The following information might help if you encounter any problems when using the command line tools.

Failure	Response
Compilation fails	Ensure that the selected Target Deployment Port matches your compiler; there may be several Target Deployment Ports for one OS, each of which targets a different compiler. If you are unsure, you can check the full name of a Target Deployment Port by opening any of the .ini files located in the Target Deployment Port directory.
Compiler reports that options.h is missing	Ensure that you have correctly prepared the options.h file, and that this file is located in a directory that is searched by your compiler (this is usually specified with -I or /I option on the compiler command line).
Compiler reports that TP.h file is missing	If you are compiling on a target different from the host where the generated file has been produced, double-check the above specific requirements to compilation on a different target. If the test script compiler and C/C++ Test Script Compiler are executed on the same machine, ensure you have not used the -NOPATH option on the test compiler command line, and that the ATLTGT environment variable was correctly set while the test compiler was executed.
Compilation fails	Ensure that the selected Target Deployment Port matches your compiler; there may be several Target Deployment Ports for one OS, each of which targets a different compiler. If you are unsure, you can check the full name of a Target Deployment Port by opening any of the .ini files located in the Target Deployment Port directory.
Compiler reports that options.h is missing	Ensure that you have correctly prepared the options.h file, and that this file is located in a directory that is searched by your compiler (this is usually specified with -I or /I option on

the compiler command line).

Compiler reports that
TP.h file is missing

If you are compiling on a target different from the host where the generated file has been produced, double-check the above specific requirements to compilation on a different target.

If the test compiler and C/C++ compiler are executed on the same machine, ensure you have not used the **-NOPATH** option on the test compiler command line, and that the **ATLTGT** environment variable was correctly set while the test script compiler was executed.

Linkage fails because of
undefined references

Ensure you have successfully compiled the Target Deployment Port library object, and have included it in your linked files

Ensure you have correctly configured the **products.h** options file.

If you are using a test feature, ensure that you are linking both source under test and additional files. You may also want to add some stubs in your **.ptu** or **.otd** test script.

Ensure the options set in **options.h** (if required) are coherent with the options set in **products.h**.

Errors are reported
through *#error* directives

You may have selected a combination of options in **products.h** which is incompatible. The error messages help you to locate the inconsistencies.

Working with Other Development Tools

6

Rational Test RealTime was designed as a versatile product that integrates within your existing development environment.

Working with Configuration Management

The GUI provides an interface that allows you to control your project files through a configuration management (CM) system such as Rational ClearCase and submit software defect report to a Rational ClearQuest system

You can also set up the GUI to use a CM system of your choice.

Working with Rational ClearCase

Rational ClearCase is a software configuration management (SCM) tool providing version control, workspace management, process configurability, and build management. With ClearCase, your development team gets a scalable, best-practices-based development process that simplifies change management – shortening your development cycles, ensuring the accuracy of your releases, and delivering reliable builds and patches for your previously shipped products.

By default, the product offers configuration management support for ClearCase. You can however customize the product to support different configuration management software. When using Rational ClearCase you

can instantly control your files from the product **Tools** menu.

Note Before using ClearCase commands, select Rational ClearCase as your CMS tool in the CMS Preferences.

To start source-controlling one or several files:

1. Select one or several files in the **Project Explorer** window.
2. From the Tools menu, select Rational ClearCase and Add to Source Control.

To check out the latest version of one or several files from ClearCase:

1. Select one or several files in the **Project Explorer** window.
2. From the Tools menu, select Rational ClearCase and Get Latest Version.

To check in one or several files into ClearCase:

1. Select one or several files in the **Project Explorer** window.
2. From the **Tools** menu, select **Rational ClearCase** and **Check In**.

To check out one or several files from ClearCase:

1. Select one or several files in the **Project Explorer** window.
2. From the **Tools** menu, select **Rational ClearCase** and **Check Out**.

To undo the check out of one or several files:

1. Select one or several files in the **Project Explorer** window.
2. From the Tools menu, select Rational ClearCase and Undo Check Out.

To compare a file with a previous version:

1. Select one or several files in the **Project Explorer** window.
2. From the **Tools** menu, select **Rational ClearCase** and **Check Out**.

To show the history of a controlled file:

1. Select a files in the **Project Explorer** window.
2. From the Tools menu, select Rational ClearCase and Show History.

To the ClearCase properties of a controlled file:

1. Select a files in the **Project Explorer** window.
2. From the **Tools** menu, select **Rational ClearCase** and **Show Properties**.

Please refer to the documentation provided with Rational ClearCase for more information.

Working with Rational ClearQuest

Rational ClearQuest is a defect and change tracking (DCT) tool designed to operate in a client/server environment. It allows you to easily track defects and change requests, target your most important problems or enhancements to your product. ClearQuest helps you determine the quality of your application or component during each phase of the development cycle and helps you track the release in which a feature, enhancement or bug fix appears.

By default, the product offers defect tracking support for ClearQuest. When using ClearQuest you can directly submit a defect report from the test or runtime analysis report in the product.

To submit a defect report from the product:

1. In the **Report Explorer**, right-click a **Failed test**.
2. From the pop-up menu, select **Submit ClearQuest Defect Report**.
3. This opens the **ClearQuest Submit Defect** window, with information about the **Failed test**.
4. Enter any other necessary useful information, and click **OK**.

Please refer to the documentation provided with Rational ClearQuest for more information.

CMS Preferences

The **Preferences** dialog box allows you to change the settings related to the integration of the product with Rational ClearCase or other configuration management software (CMS).

To change configuration management settings:

1. Select the **CMS** node.
 - **Repository directory:** Use this box to specify the location of the vault directory for the CMS tool.
 - **Selected Configuration Management System:** Use this box to select Rational ClearCase or a different CMS tool. Before setting this option, make sure that the CMS system has been configured in Tools menu.
2. Click **OK** to apply your changes.

ClearQuest Preferences

The **Preferences** dialog box allows you to specify the location of the Rational ClearQuest database.

Please refer to the documentation provided with ClearQuest for more information.

To change ClearQuest preferences:

1. Select the **ClearQuest** node.
 - **Schema Repository:** Use this box to select the schema repository you want to use.
 - **Database:** Use this box to enter the location of the ClearQuest database.
 - **User Name** and **Password:** Enter the user information provided by your ClearQuest administrator.
2. Click **OK** to apply your changes.

Customizing Configuration Management

Out of the box, the product offers configuration management support for Rational ClearCase, but the product can be configured to use most other Configuration Management Software (CMS) that uses a vault and local repository architecture and that offers a command line interface.

To configure the product to work with your version control software:

1. Add a new CMS tool to the Toolbox with the command lines for checking files into and out of the configuration management software. This activates the **Check In** and **Check Out** commands in the Project Explorer and the ClearCase Toolbar.
2. Set up version control repository in CMS Preferences.

Working with Rational Rose RealTime

Rational Rose RealTime is a software development environment tailored to the demands of real-time software. Developers use Rose RealTime to create models of the software system based on the Unified Modeling Language (UML) constructs, to generate the implementation code, compile, then run and debug the application.

Installing Rose RealTime Integration

For the integration between these products to be fully operational, Test RealTime v2002 *Release 2* and Rose RealTime v2002 (Release 1 or 2) must be installed on the same machine.

Windows Installation

If you installed the Test RealTime after Rose RealTime, the installation procedure automatically adds new menus to Rose RealTime for direct

access to the features of the product.

If not, from the Windows **Start** menu, select **Programs, Test RealTime, Tools** and **Install Rational Test RealTime add-in for Rose RealTime** to add the new menu items to Rose RealTime.

UNIX Installation

If you installed the product after Rose RealTime, the installation procedure automatically adds new menus to Rose RealTime for direct access to the features of the product.

To install the plugin you must run a script. This is because it modifies the Rose RealTime configurations in the users home directory.

The Rose RealTime installation script, for Solaris for example, can be found in the directory:

```
<installdir>/<product>.v2002R2/bin/sun4/RoseRT
```

where *<installdir>* is the directory containing the Rational products.

First read the README.txt file in this directory.

The installation script is called update_register.sh. Running it will install the plugin only for the user running the script.

If not, please re-install Test RealTime after the installation of Rose RealTime.

Using the Product with Rose RealTime

Before using Rational Test RealTime as a Rose RealTime plug-in, you must first open or create a model within Rose RealTime.

To activate Runtime Analysis features:

1. From Rose RealTime, open the **Component Specification** of the

components that you want to observe and select the **Test RealTime** tab.

2. Select **Enable Component Instrumentation**.
3. In the **Coverage** section, select the code coverage type
4. Select **Enable Memory Profiling**, **Enable Performance Profiling** and **Enable Runtime Tracing** to specify the Runtime Analysis features that you want to activate. The **Additional Options** box allows you to add other options to the Instrumentation Launcher command line.
5. Activate **Add Target Deployment Port Object Files** if you want to link the selected component with the TDP.

This is required when producing an executable. For a library component, this depends on whatever components are linked to the library.

This option also adds a new version of **cmdCommand.obj** to the object file list if such a file exists in *<InstallDir>\bin\intel\RoseRT\<TDP>*, where *<InstallDir>* is the Test RealTime installation directory and *<TDP>* is the name of the current TDP. This object file dumps SCI traces when the user clicks on the **Stop** button in Rose RealTime.

6. Select **Support Multi-threaded Code Generation** if necessary. Optionally, you can enter a new location and file name for the trace file in **Output Trace File Name**. By default, *<model directory>\atlout.spt* is used.
7. Click **OK**.
8. In Rose RealTime, from the **Tools** menu, select **Rational Test RealTime** and **Enable Instrumentation of Selected Components**. You must repeat this whenever you change any of the options described above.

To run a build with the runtime analysis features:

- In Rose RealTime, click the **Build Component** button, or from the **Build** menu, select **Build** or **Rebuild**.

These commands generate the code and *makefile*, and launch the product instrumentation with the selected options.

To run the instrumented binary:

1. Just like a standard Rose RealTime application, from the **Build** menu, select **Run** or click the **Run** button.
2. Then, click **Start** and, when appropriate, **Stop**.

Collecting Trace Dump Data

Rational's Source Code Insertion (SCI) technology is designed to minimize overhead. The instrumented code stores information in memory (except for the Runtime Tracing feature) and dumps this SCI data when the program terminates. To use this technique, you must add a call to a dumping function in your source code:

```
extern "C" _atl_obstools_dump(int);  
...  
_atl_obstools_dump(1);
```

In some cases, such as in embedded applications, it is not practical to dump traces upon exit. See *Generating Trace Dumps* for more information.

To connect the SCI data dump to the Rose RealTime Stop button:

1. Add the following code to the **cmdCommand.cc** file.

At the beginning of the file:

```
#include <RTDebugger.h>  
#include <RTMemoryUtil.h>  
#include <RTObserver.h>  
#include <RTTcpSocket.h>  
#include <stdio.h>  
extern "C" _atl_obstools_dump(int);
```

In the **RTObserver::cmdCommand** method:

```
else if( 0 == RTMemoryUtil::strcmp( commandString, "stop"  
) )  
{  
_atl_obstools_dump(1);  
}
```

```

printf("TestRT dump\n");
haltByProbe = 0;
resumeToRun = 0;
debugger->step( 0U );
}

```

2. Re-compile this file and add the **cmdCommand.obj** to the **Additional Object Files** section of the model's **Component Specification** window

Note For Visual C++ 6.0, such an object file is already provided in:

<install dir>\bin\intel\RoseRT\VC6

where <install dir> is the Test RealTime installation directory.

3. By default, when executing the model, press the Rose RealTime **Stop** button to ensure that trace information is uploaded.

Any other code point could be used to dump the traces, as long as the chosen code point is linked to a specific event a particular message or an external event in order to force the dump.

Viewing Results from Rose RealTime

To view the results with Test RealTime report viewers:

1. In Rose RealTime, from the **Tools** menu, select **Rational Test RealTime, Viewer** and select:
 - **With Model Code Coverage** to open the Code Coverage viewer of the product only on the code included in the actions of each transition and with 2 additional coverage levels for *State* and *Transition* coverage.
 - **With Code Coverage** to open the Code Coverage viewer of the product with the entire source code.

In both cases, Runtime Tracing, Memory Profiling and Performance Profiling work on the entire code.

To view coverage information in a Rose RealTime state diagram:

- In Rose RealTime, from the **Tools** menu, select **Rational Test RealTime, Model Code Coverage** and **Load**. This displays a coverage

report on each State Diagram.

Note You must run the product viewer before loading Code Coverage information on Rose RealTime.

Advanced Rose RealTime Integration

To use a cross compiler:

When using a compiler that produces code for a non-native platform, you must set up two Target Deployments Ports for both the native and the target platform.

- Locate the corresponding Target Deployment Ports. These TDPs must contain an **attolcc** Instrumentation Launcher binary.
- In the **TDP.txt** file located in the Rose RealTime installation directory, write a line for each Target Deployment Ports based on the following example:

```
NT40T.x86-VisualC++-6.0 , cvisual6
```

To compile with a makefile:

If you chose not to use the Rose RealTime environment for compilation and link, but instead to use a *makefile* to perform these tasks, you can use the Rational Test RealTime Instrumentation Launcher tools as described below:

- Modify your compiler command as follows:

```
CC = attolcc <options> -- cc  
LD = attolcc <options> -- ld (if necessary)
```

attolcc is the Instrumentation Launcher which must be available in the Target Deployment Port, in the **/cmd** directory. This directory must be in your **PATH**.

<options> are the instrumentation options. See the **Reference Manual** for more information about the Instrumentation Launcher command line.

To display the report

The instrumented application produces the **atlout.spt** file at the end of the execution.

- Run the following command:

```
studio *.fdc *.tsf atlout.spt atlout.tio atlout.tdf  
atlout.tqf atlout.tpf
```

This launches the Test RealTime graphical user interface. The **.fdc** and **.tsf** files are static files generated by the instrumentation. The four last files are created by the product to store the traces for each component.

Troubleshooting Rose RealTime Integration

In some cases, conflicts or problems may prevent the Rose RealTime integration to work as expected. The following tables sum up some of the issues that may occur, and explains how to solve them.

Project Instrumentation and Compilation

Instrumentation options cannot be changed:

The component or model is read-only. Change the component to read-write status.

An .fdc correspondence file is not found during instrumentation:

The component Cov or Cov/Model directory may have been destroyed, for example by a Clean command. To restore the lost information, run the Enable Instrumentation of Selected Component command.

New settings are ignored after performing an Enable Instrumentation of Selected Component command:

Quick Build does not regenerate makefiles. Run the Rebuild command instead of a Quick Build.

An error message states that an Instrumentor is missing during instrumentation:

Another component for which no Instrumentation Launcher (**attolcc**) is available, or no link exists between the Rose RealTime code generation and the TDP, has been enabled with Enable Component Instrumentation.

Only enable components for which a complete configuration exists.

Project Link

An application should not be instrumented with instrumented libraries:

Activate the Add TDP option for the application component. The plug-in automatically scans application dependencies and adds the TDP.Obj of instrumented libraries to the User Obj.

Note Instrumentation options must be the same for all libraries.

An application should not be instrumented with external instrumented libraries:

The Rose RealTime plug-in does not know where TDP is generated when external components are used. In this case, create an external library that contains TP.obj.

Execution

Multithreading issues:

Check that the Multithreading instrumentation setting is correctly configured.

Link issues:

When multiple subcomponents are involved in a component (libraries and binary), check that instrumentation options are the same for all components and that the TDP.obj is correctly linked.

Instrumentation issues

Check that no warning message appears during instrumentation. It may be necessary to exclude one or several components from instrumentation (**attolcc -exunit**). See the **Reference Manual** for further information about Instrumentation Launcher command line options.

Missing Results

Files are missing when the Test RealTime is launched to display report files. Code Coverage results are missing or display the entire application as uncovered.

The runtime analysis trace dump was interrupted. Dumps can take a long time, especially when the Memory Profiling feature is in use. See Generating SCI Dumps for more information.

Missing files on another component:

The plug-in offers to display all the results for enabled components. Disable the any components that are not under analysis.

No coverage results on a diagram

Check that the component was correctly generated with the Code Coverage instrumentation option.

Check that the component is enabled for instrumentation. The Plug-in only changes state diagrams for enabled components.

Check that the component is not read-only, such as for an inherited diagram.

Working with Rational TestManager

You can open and run test cases associated with Test RealTime from TestManager.

When you execute a Test RealTime test node from Test Manager, the Test RealTime GUI does not appear and the test node is run silently. To obtain full feedback of test execution, run the test node directly from Test RealTime.

To open a test or application node from TestManager:

1. In TestManager, from the **File** menu, select **Open Test Script**. Alternatively, click the **Open** button on the **Implementation** tab of a **Test Case Properties** window.
2. Select a workspace in the **Select a Workspace within the Project** window. The **Select a Test Node / Group Node** lists all the test nodes and group nodes within the workspace.

The corresponding project opens in Test RealTime.

To execute a test node from TestManager:

- In TestManager, execute test cases associated with Test RealTime like any other test cases. No special steps are required.

Before Using the TestManager Integration

All of the Test RealTime integration functionality is accessed and performed in TestManager.

A Rational project must be enabled with the **Enable Rational Project** tool in order to be used with Test RealTime.

The Rational project and Test RealTime project can be located on any network-accessible drive, but test execution must occur locally, on the Windows machine. TestManager does not support remote target execution of Test RealTime tests.

Test Manager associates test cases with nodes in the Test RealTime Project Explorer.

You can only associate TestManager test cases with Test RealTime test nodes. application nodes are not supported.

Although you can associate a test case with a Test RealTime group node, it is recommended that a one-to-one correspondence between the test case and the individual test node is preserved.

To associate a TestManager test case with a test node:

1. In TestManager, access the properties window of a test case and select the **Implementation** tab.
2. In the Automated Implementation section, click **Select** and choose **Rational Test RealTime**.
3. In the **Rational Test RealTime Test Selection** box, click **Browse** to select an **.rtp** Test RealTime project file and click **Open**.

Note When the **.rtp** file is located on a network drive, indicate the location with a UNC path ("**\\machine_name\directory\file**") instead of using a mapped drive letter ("**G:\directory\file**").

The **Select a Workspace within the Project** window appears, displaying a list of workspaces contained in the project.

5. Select a workspace in the **Select a Workspace within the Project** window. The **Select a Test Node / Group Node** lists all the test nodes and group nodes within the workspace.
6. Select the test node or group node that you want to associate with the TestManager test case, and click **OK**. You can specify either a single test node or a Group node that can contain several test nodes.

The text box in the Automated Implementation section now displays the following path to the test node:

<group node name>.<test node name>

Note Click **Options** in the **Implementation** tab after selecting a Test RealTime test or group node, to view the path to the Test RealTime project, the workspace name, and the test or group node names.

Installing TestManager Integration

Integration with TestManager is only available for the Windows version of Test RealTime.

Both Test RealTime 2002.05 and TestManager 2002.05 must be installed on the same machine.

To enable Test RealTime integration with TestManager:

1. From the Windows **Start** menu, select **Programs, Test RealTime, Tools** and **Install Rational TestManager Integration** to install the additional files for the Test RealTime support in TestManager.
2. From the Windows **Start** menu, select **Programs, Test RealTime, Tools** and **Enable Rational Project for TestManager Integration**. This tool updates a Rational project for use with Test RealTime tests. This menu item only appears once the **Install Rational TestManager Integration**

tool has been run.

3. Log in to an existing Rational project. A dialog box then confirms that the project has been enabled.

The **Enable Rational Project** tool must be run once for each Rational project. You cannot enable more than one Rational project at the same time.

Submitting a ClearQuest Defect from TestManager

In TestManager, you can submit a defect relating to a Test RealTime test node as part of the ClearQuest integration.

To automatically submit the script name, you must submit the defect from the Script Start line and not from the User Defined line. This is because TestManager does not submit the Script Name from a User Defined line.

Viewing Results in TestManager

Once a Test RealTime test node has been executed, the results are accessible from the TestManager LogViewer.

For each TestManager test case, the LogViewer displays a User Defined line per Test RealTime test node. This means that if a test case was associated with a group node, and the group node contained five test nodes, then five **User Defined** lines are shown in the test log for this particular test case. Each line has its own associated *Pass* or *Fail* status

In the properties window of a User Defined line, on the **General** tab:

- For a passed test: The Failure Description field indicates:
All <x> tests passed
where <x> is the total number of tests performed by a particular test node

- For a failed test: The Failure Description field indicates:
<x> **tests failed.** <y> **tests passed**
where <x>+<y> is the total number of tests performed by a particular test node.

In the properties window of a User Defined line, on the **View Test RealTime Logs** tab, click the **Open** button to view the test and runtime analysis reports for the selected test node.

Note These files are copies of the original test scripts and source files solely intended for report purposes. Any changes to these files are not made to the actual test scripts and source files. Open the original files in Test RealTime for debugging purposes.

Working with Rational TestManager

Rational TestManager is used to manage all aspects of testing and all sources of information related to the testing effort throughout all phases of a software development project.

Test RealTime integration with TestManager enables the following features:

- Association of TestManager test inputs, via test cases, with Test RealTime test nodes and Group nodes.
- Execution of Test RealTime tests from within TestManager
- Local copy of Test RealTime test and runtime analysis results, test scripts, and referenced source code in the Rational project log folder, all of which can be baselined along with other Rational project log files
- Test RealTime test and runtime analysis results available within the Test RealTime GUI directly from a LogViewer test log
- Facilitated debug efforts by maintaining the original test and runtime analysis results in the Test RealTime project.

- Execution support of multiple Test RealTime tests in multiple projects stored on multiple machines from within a single TestManager test suite.

Working with Microsoft Visual Studio

Installing Microsoft Visual Studio Integration

Integration with Microsoft Visual Studio is only available for the Windows versions of Test RealTime.

Either Test RealTime and Microsoft Visual Studio 6.0 must be installed on the same machine.

To enable the product integration with Visual Studio, from the Windows **Start** menu, select **Programs, Test RealTime, Tools** and **Install Rational Test RealTime add-in for Microsoft Visual Studio 6.0** to add the new menu items to Microsoft Visual Studio.

Configuring Microsoft Visual Studio Integration

Test RealTime provide a special setup tool to configure runtime analysis features with Microsoft Visual Studio 6.0.

Note Integration with Microsoft Visual Studio is only available with the Windows version of the product.

Configuration

The **Rational Test RealTime Setup for Microsoft Visual Studio** tool allows you to set up and activate coverage types and instrumentation options for

Test RealTime runtime analysis features, without leaving Microsoft Visual Studio.

To run the product Setup for Microsoft Visual Studio:

In Microsoft Visual Studio, two new items are added to the **Tools** menu:

- **Test RealTime Viewer:** this launches the Test RealTime user interface, providing access to reports generated by Test RealTime runtime analysis and test features.
- **Test RealTime Options:** this launches the Rational Setup for Microsoft Visual Studio tool.

The following commands are available:

- **Apply:** Applies the changes made
- **OK:** Apply the choices made and leave the window
- **Enable** or **Disable:** Enables or disables the runtime analysis features
- **Cancel:** Cancels modifications

Code Coverage Instrumentation options

See About Code Coverage and the sections about coverage types.

- **Function instrumentation:**
 - Select **None** to disable instrumentation of function inputs, outputs and termination instructions.
 - Select **Functions** to instrument function inputs only.
 - Select **Exits** to instrument function inputs, outputs and termination instructions.
- **Function calls instrumentation (C only):**
 - Select **None** to disable function call instrumentation.

- Select **Calls** to enable function call instrumentation.
- **Block instrumentation**
 - Select **None** to disable block instrumentation.
 - Select **Statement Blocks** to instrument simple blocks only.
 - Select **Implicit Blocks** to instrument simple and implicit blocks.
 - Select **Loops** to instrument implicit blocks and loops.
- **Condition instrumentation (C only)**
 - Select **None** to disable condition instrumentation
 - Select **Basic** to instrument basic conditions
 - Select **Modified/Multiple** to instrument multiple
 - Select **Forced** to instrument forced multiple conditions
- **No Ternaries Code Coverage:** when this option is selected, simple blocks corresponding for the ternary expression true and false branches are not instrumented
- **Instrumentation Mode:** see Information Modes for more information.
 - **Pass mode:** allows you to distinguish covered branches from those not covered.
 - **Count mode:** The number of times each branch is executed is displayed in addition to the pass mode information in the coverage report.
 - **Compact mode:** The compact mode is similar to the Pass mode. But each branch is stored in one bit instead of one byte

to reduce overhead.

Other Options

- **Dump:** this specifies the dump mode:
 - Select **None** to dump on exit of the application
 - Select **Calling** to dump on call of the specified function
 - Select **Incoming** to dump when entering the specified function
 - Select **Returning** to dump when exiting from the specified function
- **Static Files Directory:** allows you to specify where the .fdc and .tsf files are to be generated
- **Runtime Tracing:** this option activates the Runtime Tracing runtime analysis feature
- **Memory Profiling:** this option activates the Memory Profiling runtime analysis feature
- **Performance Profiling:** this option activates the Performance Profiling runtime analysis feature
- **Other:** allows you to specify additional command-line options that are not available using the buttons. See the **Test RealTime Reference Manual** for a complete list of Instrumentor options.

Technical Support

7

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

Glossary

Additional Files

Source files that are required by the test script, but not actually tested.

API

Application Programmer Interface. A reusable library of subroutines or objects that encapsulates the internals of some other system and provides a well-defined interface. Typically, it makes it easier to use the services of a general-purpose system, encapsulates the subject system providing higher integrity, and increases the user's productivity by providing reusable solutions to common problems.

Application

A software program or system used to solve a specific problem or a class of similar problems.

Application node

The main building block of your application under analysis. It contains the source files required to build the application.

Assertion

A predicate expression whose value is either true or false.

Asynchronous

Not occurring at predetermined or regular intervals.

Black box testing

A software testing technique whereby the internal workings of the item being tested are not known by the tester.

Boundary

The set of values that defines an input or output domain.

Boundary condition

An input or state that results in a condition that is on or immediately adjacent to a boundary value.

Branch

When referring to the Code Coverage feature, a branch denotes a generic unit of enumeration. For a given branch, you specify the coverage type. Code Coverage instruments this branch when you compile the source under test.

Branch coverage

Achieved when every path from a control flow graph node has been executed at least once by a test suite. It improves on statement coverage because each branch is taken at least once.

Breakpoint

A statement whose execution causes a debugger to halt execution and return control to the user.

Bug

An error or defect in software or hardware that causes a program to malfunction.

Build

The executable(s) produced by a build generation process. This process

may involve actual translation of source files and construction of binary files by e.g. compilers, linkers and text formatters.

Build generation

The process of selecting and merging specific versions of source and binary files for translation and linking within a component and among components.

Check-in

In configuration management, the release of exclusive control of a configuration item.

Check-out

In configuration management, the granting of exclusive control of a configuration item to a single user.

Class

A representation or source code construct used to create objects. Defines public, protected, and private attributes, methods, messages, and inherited features. An object is an instance of some class. A class is an abstract, static definition of an object. It defines and implements instance variables and methods.

Class contract

The set of assertions at method and class scope, inherited assertions, and exceptions.

Class invariant

An assertion that specifies properties that must be true of every object of a class.

Clear box testing

A software testing technique whereby explicit knowledge of the internal

workings of the item being tested are used to select the test data. Test RealTime leverages the power of source code analysis to initiate the creation of white box tests.

Code Coverage

Test RealTime feature whose function is to measure the percentage of code coverage achieved by your testing efforts, using a variety of powerful data displays to ensure all portions of your code are exercised and thus verified as properly implemented.

Complexity

A characteristic of software measured by various statistical models.

Component

Any software aggregate that has visibility in a development environment, for example, a method, a class, an object, a function, a module, an executable, a task, a utility subsystem, an application subsystem. This includes executable software entities supplied with an API.

Component Testing

The Test RealTime feature used to automate the white box testing of individual software components in your system, facilitating early, proactive debugging and provided a repeatable, well-defined process for runtime analysis.

Computational complexity

The study of the time (number of iterations) and space (quantity of storage) required by algorithms and classes of algorithms.

Configuration

It is a Target Deployment Port, applied to a Project, plus node-specific settings.

Configuration management

A technical and administrative approach to manage changes and control work products.

Container class

A class whose instances are each intended to contain multiple occurrences of some other object.

Coverage

The percentage of source code that has been exercised during a given execution of the application.

Cyclomatic complexity

The $V(g)$ or cyclomatic number is a measure of the complexity of a function which is correlated with difficulty in testing. The standard value is between 1 and 10. A value of 1 means the code has no branching. A function's cyclomatic complexity should not exceed 10.

Debug

To find the error or misconception that led to a program failure uncovered by testing, and then to design and to implement the program changes that correct the error.

Debugger

A software tool used to perform debugging.

Defect

An incorrect or missing software component that results in a failure to meet a functional or performance requirement.

Destructor

A method that removes an active object.

Embedded system

A combination of computer hardware and software, and perhaps additional mechanical or other parts, designed to perform a dedicated function. In some cases, embedded systems are part of a larger system or product, as is the case of an anti-lock braking system in a car.

Equivalence class

A set of input values such that if any value is processed correctly (incorrectly), then it is assumed that all other values will be processed correctly (incorrectly).

Error

A human action that results in a software fault.

Event

Any kind of stimulus that can be presented to an object: a message from any client, a response to a message sent to the virtual machine supporting an object, or the activation of an object by an externally managed interrupt mechanism.

Exception

A condition or event that causes suspension of normal program execution. Typically it results from incorrect or invalid usage of the virtual machine.

Exception handling

The activation of program components to deal with an exception. Exception handling is typically accomplished by using built-in features and application code. The exception causes transfer to the exception handler, and the exception handler returns control to the module that invoked the module that encountered the exception.

Garbage collector (Java)

The process of reclaiming allocated blocks of main memory (garbage) that are (1) no longer in use or (2) not claimed by any active procedure.

Included Files

Included files are normal source files under test. However, instead of being compiled separately during the test, they are included and compiled with the object test driver script.

Inheritance

A mechanism that allows one class (the subclass) to incorporate the declarations of all or part of another class (the superclass). It is implemented by three characteristics: extension, overriding, and specialization.

Instrumentation

The action of adding portions of code to an existing source file for runtime analysis purposes. The product uses Rational's source code insertion technology for instrumentation.

JUnit

JUnit is an open source testing framework for Java. It provides a means of expressing how the application should work. By expressing this in code, you can use JUnit test scripts to test your code.

Memory profiling

Test RealTime feature whose function is to measure your code's reliability as it pertains to memory usage. Applicable to both Application and Test Nodes, the memory profiling feature detects memory leaks, monitors memory allocation and deallocation and provides detailed reports to simplify your debugging efforts.

Method (Java, C++)

A procedure that is executed when an object receives a message. A method is always associated with a class.

Model

A representation intended to explain the behavior of some aspects of [an artifact or activity]. A model is considered an abstraction of reality.

Node

Any item that appears in the Project Explorer. This includes test nodes, application nodes, source files or test scripts.

Package (ADA)

Program units that allow the specification of groups of logically related entities.

Package (Java)

A group of types (classes and interfaces).

Performance profiling

Test RealTime feature whose function is to measure your code's reliability as it pertains to performance. Applicable to both Application and Test nodes, the performance profiling feature measures each and every function, procedure or method execution time, presenting the data in a simple-to-read format to simplify your efforts at code optimization.

Polymorphism

This refers to a programming language's ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes.

Postcondition

An assertion that defines properties that must hold when a method completes. It is evaluated after a method completes execution and before the message result is returned to the client.

Precondition

An assertion that defines properties that must hold when a method begins execution. It defines acceptable values of parameters and variables upon entry to a module or method.

Predicate expression

An expression that contains a condition (conditions) that evaluates true or false.

Procedure (C)

A procedure is a section of a program that performs a specific task.

Project

The project is your main workspace as shown in the Project Explorer. The project contains all the files required to build, analyze and test an application.

Requirement

A desired feature, property, or behavior of a system.

Runtime Tracing

The Test RealTime feature whose function is to monitor code as it executes, generating an easy-to-read UML-based sequence diagram of events. Perfect for developers trying to understand inherited code, this feature also greatly simplifies the debugging process at the integration level.

Scenario

An interaction with a system under test that is recognizable as a single unit of work from the user's point of view. This step, procedure, or input event may involve any number of implementation functions.

SCI

Source Code Insertion. Method used to enable the runtime analysis functionality of Test RealTime. Pre-compiled source code is modified via the insertion of custom commands that enable the monitoring of executing code. The actual code under test is untouched. The testing features of Test RealTime do not require SCI.

SCI dump

Data that is dumped from a SCI-instrumented application.

Sequence diagram

A sequence diagram is a UML diagram that provides a view of the chronological sequence of messages between instances (objects or classifier roles) that work together in an interaction or interaction instance. A sequence diagram consists of a group of instances (represented by lifelines) and the messages that they exchange during the interaction.

Snapshot

In Memory Profiling for Java, a snapshot is a memory dump performed by the JVMPI Agent whenever a trigger request is received. The snapshot provides a status of memory and object usage at a given point in the execution of the Java program.

Subsystem

A subset of the functions or components of a system.

System Testing

The Test RealTime feature dedicated to testing message-based applications. It helps you solve complex testing issues related to system interaction, concurrency, and time and fault tolerance by addressing the functional, robustness, load, performance and regression testing phases from small, single threads or tasks up to very large, distributed systems.

TDP

Target Deployment Port. A versatile, low-overhead technology enabling target-independent tests and runtime analysis despite limitless target support. Its technology is constructed to accommodate your compiler, linker, debugger, and target architecture.

Template class

A class that defines the common structure and operations for related types. The class definition takes a parameter that designates the type.

Test driver

A software component used to invoke a component under test. The driver typically provides test input, controls and monitors execution, and reports results.

Test harness

A system of test drivers and other tools to support test execution.

Test node

The main building block of your test campaign. It contains one or more test scripts as well as the source code under test.

Transition

In a state machine, a change of state.

UML

Unified Modeling Language. A general-purpose notational language for specifying and visualizing complex software, especially large, object-oriented projects.

Unit

Generic term referring to language specific code elements such as procedures, classes, functions, methods, packages.

Unit Testing

See Component Testing.

White box testing

See Clear box testing.