Rational. IBM Rational Test RealTime

IBM

Version 7.0.0
Windows, UNIX, and Linux

User Guide

GI11-6755-00

Rational® IBM Rational Test RealTime

Version 7.0.0

Windows, UNIX, and Linux

User Guide

GI11-6755-00

Before using this information, be sure to read the general information under *Notices* on page 291

7th edition (May 2006)

This edition applies to version 7.0.0 of IBM Rational Test RealTime and to all subsequent releases and modifications until otherwise indicated in new editions. This edition replaces GI-6350-00.

Table Of Contents

Preface

Thank you for using IBM Rational Test RealTime!

This information center contains extensive information on a broad range of subjects that will help you enhance your software testing experience with IBM Rational Test RealTime.

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.

IBM 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 tool set for embedded, real-time and networked systems created in any cross-development environment. Automated testing, code coverage, memory leak detection, performance profiling, UML tracing, code review - 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.

For more information about Rational Test RealTime, visit the product Web site at:

http://www.ibm.com/software/awdtools/test/realtime

About Rational software from IBM

Rational software from IBM helps organizations create business value by improving their software development capability. The Rational software development platform integrates software engineering best practices, tools, and services. With it, organizations thrive in an on demand world by being more responsive, resilient, and focused. Rational's standards-based, cross-platform solution helps software development teams create and extend business applications, embedded systems and software products. Ninety-eight of the Fortune 100 rely on Rational tools to build better software, faster. Additional information is available at http://www.ibm.com/software/rational.

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.
- Mozilla Firefox 1.0 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.

Contacting IBM Customer Support

If you have questions about installing, using, or maintaining this product, contact IBM Customer Support as follows:

The IBM software support Internet site provides you with self-help resources and electronic problem submission. The IBM Software Support Home page for Rational products can be found at:

http://www.ibm.com/software/rational/support/

Voice Support is available to all current contract holders by dialing a telephone number in your country (where available). For specific country phone numbers, go to:

http://www.ibm.com/planetwide/

Note When you contact IBM Customer Support, please be prepared to supply the following information:

- Your name, company name, ICN number, telephone number, and e-mail address
- Your operating system, version number, and any service packs or patches you have applied
- Product name and release number
- Your PMR number (if you are following up on a previously reported problem)

Downloading the IBM Support Assistant

The IBM Support Assistant (ISA) is a locally installed serviceability workbench that makes it both easier and simpler to resolve software product problems. ISA is a free, stand-alone application that you download from IBM and install on any number of machines. It runs on AIX, (RedHat Enterprise Linux AS), HP-UX, Solaris, and Windows platforms.

ISA includes these features:

- Federated search
- Data collection
- Problem submission
- Education roadmaps

For more information about ISA, including instructions for downloading and installing ISA and product plug-ins, go to the ISA Software Support page:

http://www.ibm.com/software/support/isa/

Chapter 1. Installing, migrating, configuring

Full instructions for installing IBM Rational Test RealTime are included in the **IBM Rational Test RealTime Installation Guide**, which is supplied on the product CD or as part of the download.

Example projects

IBM Rational Test RealTime is provided with a range of example projects aimed at demonstrating most of the features of the product. You may use them to familiarize yourself with those features. Do not hesitate to review and manipulate the source files and scripts provided in these examples.

Most examples are designed to run directly with a default Configuration.

To access open an example project:

1. From the Start page, click Examples on the left side of the page. This opens the Examples page.
2. Click any of the example projects to open them in the product.

Example	Language	Description
BaseStation C	C/C++	Main sample project covering most test and runtime analysis features. This sample is used for the C and C++ tutorial.
BaseStation Java	Java	Main sample project covering most test and runtime analysis features. This sample is used for the Java tutorial.
Broadcast Server	C/C++	A sample that demonstrates the use of System Testing for C with C++ code and multithreaded components. This example is provided as is. Usage of System Testing with C++ is not supported.
Chained List	C++	This sample shows how to test chained lists with Component Test for C++.
ABWL Check Frequency	C	This sample demonstrates the Memory Profiling manual check feature for checking ABWL and MFWL errors. See Checking for ABWL and FMWL errors for more information.
Data Pool	C	This project shows how to incorporate data tables in .csv format in your tests. See Importing a Data Table (.csv File).
Enum Ada	Ada	A project that demonstrates Component Testing for Ada for testing *enum* types.
EuroDollarMidlet J2ME	Java	This sample shows how to use Component Testing for Java on J2ME (Java 2.0 Micro Edition) applications.
Generic Ada	Ada	A sample demonstrating how to test generic units in Ada. See Testing Generic Compilation Units.
Histogram Ada	Ada	This project demonstrates array handling and overriding ENVIRONMENT statements with Component Test for Ada.
Philosopher	C	A sample C application for Runtime Analysis in a multithreaded environment (Windows only)

Dinner Party	C++	A Component Testing for C++ sample application in a multithreaded environment with class inheritance (Windows only).
Shape	C++	A simple class inheritance example for Component Testing for C++.
Shape Ada	Ada	This example demonstrates Component Testing on object-oriented Ada.
Shared Library	C++	This sample demonstrates how to use shared library files in your applications. See Using shared libraries for more information.
Stack	C	A simple example project on a stack application for System Testing for C.
Stub Ada	Ada	A simple example project demonstrating the usage of stubs with Component Testing for Ada. See Stub simulation Ada.
Stub C	C	A simple example project demonstrating the usage of stubs with Component Testing for C. See Stub Simulation in C.
Task Ada	Ada	A simple example project demonstrating how to test Ada tasks with Component Testing for Ada.
Add	C	An extremely short example to help you to develop new TDPs.
Template Cpp	C++	An example of Component Testing for C++ on C++ templates
Testing Ada	Ada	This sample demonstrates how to test various variable types in Ada.
Testing C	C	This sample demonstrates how to test various variable types in C.
Test Suite Ada	Ada	Use this sample to validate any changes made to an Ada Target Deployment Port.
Test Suite C	C	Use this sample to validate any changes made to a C Target Deployment Port.

Target deployment technology overview

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.ibm.com/developerworks/rational/library/4015.html

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

Downloaded TDPs can be freely used an 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 IBM Professional Services, please contact IBM via one of these methods:

- Contact your IBM Sales Representative directly.
- If you don't know your Sales Representative, contact IBM Rational Customer Support via this link:
  http://www.ibm.com/software/rational/support/

Reconfiguring a TDP for a compiler or JDK

During installation of IBM 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 JDK:

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**.

Configuring target deployment ports

This section covers the Target Deployment Port (TDP) technology used by IBM Rational Test RealTime to support embedded target platforms.

This chapter is intended for advanced users of the product. Advanced knowledge of the target compiler, platform, development and test environment are required for Target Deployment Port customization tasks. Knowledge of Perl scripts is also required.

Determining target requirements

The following tables lists the minimum requirements that your development environment must provide to enable use of each feature of Test RealTime:

- C, C++ and Ada requirements
- Java requirements

C, C++ and Ada Requirements

The following table lists the requirements for each feature of the product.

	Comp. Testing for C and Ada	**Comp. Testing for C++**	**System Testing for C #VT=1**	**System Testing for C #VT>1**	**Code Coverage**	**Runtime Tracing**	**Memory Profiling**	**Perf. Profiling**
Data Retrieval Capability	R	R	R	R	R	R	R	R
Free Data Space					S	S	S	S
Free Stack Space					S	S	S	S
Mutex		R			M	M	M	M
Thread Self and PrivateData		R				M	M	M
Clock Interface			R	R				R
Heap Management			R	R			R	
High Speed Link						R		
Task Management		M		R		M	M	M
BSD Sockets				R				

Ada	N/A	N/A	N/A	N/A	N/A	N/A

- **R:** Mandatory requirement. If this requirement is unavailable, then it may not be possible to use the product on the platform.
- **S:** Required only for stand alone use of a Runtime Analysis feature. If you are not using a Component Testing feature, these requirements are not mandatory.
- **M:** Required only if the application under test is a multi-threaded application running on a preemptive multi-tasking mechanism.

  Note Only the Component Testing for C and Ada and Code Coverage features support the Ada language. System Testing for C can, however, be used to send messages to an Ada-written application if C bindings exist for that feature.

Java Requirements

The following table lists the requirements for each feature of the product.

	Comp. Testing for Java	**Code Coverage**	**Runtime Tracing**	**Mem. Profiling**	**Perf. Profiling**
Data Retrieval Capability	R	R	R	R	R
Free Data Space		S	S	S	S
Free Stack Space		S	S	S	S
Thread Adaptation		R	R		R
Clock Adaptation					R
JVMPI Support				R	
Heap Settings				R	

- **R:** Mandatory requirement. If this requirement is unavailable, then it may not be possible to use the product on the platform.
- **For stand alone:** Required for stand alone use of a runtime analysis feature - i.e. used without Component Testing for Java.

Data Retrieval Capability

Test programs or instrumented applications need to generate a text file on the host - this is how information is gathered to prepare Test RealTime reports.

The Target Deployment Port gathers this report data by obtaining the value of a (char *) global variable, containing regular ASCII codes, from the application or test driver running on the target machine.

This retrieval can be accomplished in whichever way is most practical for the target. It could be through file system access, a socket, specific system calls or a debugger script. Most known environments allow at least some form of I/O.

At least one form of data retrieval capability is required.

Free Data Space

All runtime analysis tools are based on Rational Source Code Insertion (SCI) technology. The overhead introduced by this technology is dependent both on the selected instrumentation level and on code complexity.

The Code Coverage feature requires the most free data space. The overhead for default Code Coverage levels (procedure/method entries and decisions) typically increases code size by 25%. Runtime Tracing, Memory Profiling and Performance Profiling introduce a significantly lower overhead (about 16 bytes per instrumented file).

The Component Testing features of Test RealTime do not typically require additional free memory because it is rare for the entire application to be run on the target.

Free Stack Space

The stack size should not be optimized for the requirements of the original application. The Test RealTime instrumentation process adds a few bytes to the stack and inserts calls to the TDP embedded runtime library.

Since, based on experience, it is difficult to identify stack overflow, the user should assume that each instrumented function requires, on average, an extra 30 bytes for local data.

Mutex

This customization is required by all runtime analysis tools of the product if the application under test uses a preemptive scheduling mechanism. A mutual exclusion mechanism is required to ensure uninterrupted operation of critical sections of the Target Deployment Port.

Thread Self and Private Data

It must be possible to retrieve the current identifier of a thread, and it must be possible to create thread-specific data (e.g. pthread_key_create for POSIX).

Time management

A clock interface is not necessary for the Component Testing for C and Ada, for C++, Memory Profiling and Performance Profiling features, but it is required for Performance Profiling and System Testing for C. The goal is to read and return a clock value (Performance Profiling) and to provide time out values (System Testing for C).

If you are using Performance Profiling and System Testing for C with Component Testing and the clock interface does exist, then Component Testing indicates time measurements for each function under test and the Runtime Tracing feature timestamps all messages.

Heap Management

This customization is required by Memory Profiling and System Testing for C only.

Both Memory Profiling and System Testing for C need to allocate memory dynamically.

Memory Profiling also tracks and records memory heap usage, based on the standard **malloc** and **free** functions. However, it can also handle user-defined or operating system dependent memory usage functions, if necessary.

High-Speed Link

For Runtime Tracing On-the-Fly only.

To use the Runtime Tracing feature without a testing feature, a high-speed link between the host and target machine is required in order to take full advantage of the On-the-Fly tracing mode. This is because Runtime Tracing-instrumented code "writes a line" to the host for each entry point and exit point of every instrumented function. This means that as the application is running, a continuous flow of messages is written to the host. Understandably, a 9600 bit rate, for example, would not be sufficient for use of the Runtime Tracing feature with an entire application.

Note that the Code Coverage, Memory Profiling and Performance Profiling features store their data in static target memory, and data is only sent back to the host at specified flush points (with the Runtime Tracing feature, static memory is also flushed when it becomes full). Technically, a Memory Profiling, Performance Profiling, and Code Coverage instrumented application can run for weeks without seeing a growth in consumed memory; nothing need be sent to the host until a user-defined flush point is reached.

Task Management

Runtime analysis features require task management capabilities when they are used to monitor multi-threaded applications.

When the System Testing feature for C executes more than one virtual tester, full task management capabilities must be available. In other words, System Testing for C should be able to start a task, stop a task, and get the status of a task.

BSD Socket Compliance

When the System Testing feature for C executes more than one virtual tester, the target must be BSD socket compliant. This is necessary because System Testing for C uses TCP/IP sockets to enable communication between System Testing Agents and the System Testing Supervisor, as well as to enable virtual tester RENDEZVOUS synchronization.

If, in fact, the target host is BSD socket-compliant, then it is guaranteed that you can address the Data Retrieval Capability and the High-Speed Link requirements.

Thread Adaptation

This is required by all Java runtime analysis tools except Memory Profiling for Java.

The **waitForThreads** method must wait for the last thread to terminate before dumping results and exiting the application.

On J2ME platforms, this method is empty.

Clock Adaptation

This customization is required for the Performance Profiling feature

- The **getClock** method must return the clock value, represented as a *long*.
- The **getClockUnit** method must return an array of bytes representing the clock unit.

JVMPI Support

The Java Virtual Machine (JVM) must support the JVM Profiler Interface (JVMPI) technology used for memory monitoring.

This is required for Memory Profiling for Java.

Heap Settings

This customization is part of the JVMPI support settings.

If available, the dynamic memory allocation required by the feature is made through standard *malloc* and *free* functions.

If the use of such routines is not allowed on the target, fill **JVMPI_SIZE_T**, **jvmpi_usr_malloc** and **jvmpi_usr_free** types and functions with the appropriate code.

Determining target architecture support

If your target can be used in Standard or User Mode, then it is fully supported by Test RealTime.

However, if your target can only be used in Breakpoint Mode, then you must ask yourself the following questions to determine if your target platform has enough data retrieval capability to be supported by Test RealTime:

- Does this debugger provide access to symbols?
- Is there a command language?
- Is there a way to run commands from a file?
- Can a command file be executed automatically when the debugger starts, either from a particular filename or from an option of the command line syntax.
- Is there a command to stop the debugger? (The execution process must be blocked until execution is terminated and the trace file is generated.)
- Is there a way to set software breakpoints?
- Is there a way to log what happens into a file?
- Is there a way to dump the contents of a variable in any format, or to dump a memory buffer and log the value?
- Can the debugger automatically run other debugger commands when a breakpoint is reached, such as a variable dump and resume; or, alternatively, does the debugger command language include loop instructions?

If the answer to any of these questions is "No", then no data retrieval capability exists. Therefore, test and runtime analysis feature execution on the target machine will not be possible with Test RealTime.

Retrieving data from the target platform

Data Retrieval is accomplished through the association of the Target Deployment Port library functions with an execution procedure.

The following examples demonstrate the Standard, User, and Breakpoint Modes, based on a simple program which writes a text message to a file named "cNewTdp\\atl.out".

Standard Mode Example: Native

```
#define RTRT_FILE FILE *
RTRT_FILE usr_open(char *fileName)
  { return((RTRT_FILE)(fopen(fileName,"w"))); }
void usr_writeln(RTRT_FILE f,char *s)
  { fprintf(f,"%s",s); }
void usr_close(RTRT_FILE f)
  {fclose(f) ;}
char atl_buffer[100];
void main(void)
{
RTRT_FILE f ;
strcpy(atl_buffer,"Hello World ");
f=usr_open("cNewTdp\\atl.out");
usr_writeln(f,atl_buffer);
usr_close(f);
}
```

Execution command : a.out

When executing a.out, cNewTdp\atl.out will be created, and will contain "Hello World".

User Mode Example: BSO-Tasking Crossview

Source code of the program running on the target:

```
#define RTRT_FILE int
RTRT_FILE usr_open(char *fName) { return(1); }
void usr_writeln(RTRT_FILE f,char *s) { _simo(1,s,80); }
void usr_close(RTRT_FILE f) { ; }
char atl_buffer[100];
void main(void)
{
  RTRT_FILE f ;
  strcpy(atl_buffer,"Hello World");
  f=usr_open("cNewTdp\\atl.out");
  usr_writeln(f,atl_buffer);
  usr_close(f);
}
```

Execution command from host:

```
xfw166.exe a.out -p TestRt.cmd
```

Content of TestRt.cmd:

```
1 sio o atl.out
r
q y
```

In this example, usr_open and usr_close functions are empty. Priv_writeln uses a BSO-Tasking function, _simo, which allows to send the content of the s parameter on the channel number 1 (an equivalent of a file handle).

On another side, on the host machine, the Crossview simulator (launched by the xfw166.exe program) is configured by the command

```
1 sio o atl.out
```

indicating to the simulator running on the host, that any character being written on the channel number 1 should be logged into a file name atl.out

The next command is to run the program, and quit at the end.

The original needs, which was to have cNewTdp\atl.out file be written on the host has to completed by a script on the host machine, consisting in moving the atl.out generated in the current directory into the cNewTdp directory. The complete execution step would be in Perl:

```
SystemP("xfw166.exe a.out -p TestRt.cmd");
If ( ! -r atl.out ) { Error…. return(1);}
move("atl.out","cNewTdp/atl.out");
```

Breakpoint-Mode :

In all the breakpoint mode examples, the usr_ functions are empty.

Breakpoint Mode Example: Keil MicroVision

Source code of the program running on the target:

```
#define RTRT_FILE int
RTRT_FILE usr_open(char *fName) { return(1); }
void usr_writeln(RTRT_FILE f,char *s) {;}
void usr_close(RTRT_FILE f) { ; }
char atl_buffer[100];
void main(void)
```

```
{
RTRT_FILE f ;
strcpy(atl_buffer,"Hello World");
f=usr_open("cNewTdp\\atl.out");
usr_writeln(f,atl_buffer);
usr_close(f);
}
```

Execution command from host:

```
uv2.exe -d TestRt.cmd
```

Content of TestRt.cmd:

```
load a.out
func void out(void) {
int i=0;
while(atl_buffer[i]) printf("%c",atl_buffer[i++]);
printf("\n");
}
bs usr_writeln,"out()"
bs usr_close
reset
log > Tmpatl.out
g
exit
```

In this example, all the usr_ functions are empty. The intelligence is deported into the TestRt.cmd script which a command file for the debugger.

It first loads a.out executable program. It then defines a function, which prints the value of atl_buffer in the MicroVision command window. Then it sets two breakpoints. The first one in usr_writeln, and the second one in usr_close. When usr_writeln is reached, the program halts, and the debugger automatically runs his out() function, which print the value of atl_buffer into its command window. When usr_close is reached, the program halts.

Then, the debugger scripts resets the processor, and logs anything that happens in the debugger command window into a file named Tmpatl.out. It then starts the execution, (which finally halts when usr_close is reached as no action is associated with this breakpoint) and exits.

The final result is contained into Tmpatl.out, which should be cleanup by the host (a little decoder in Perl for example) to give the final expected cNewTdp\atl.out file containing "Hello World". The global execution step in Perl would be:

```
SystemP("uv2.exe -d TestRt.cmd") ;
# Decode and clean Tmpatl.out and write the results in
# cNewTdp\atl.out
Decode_Tmpatl.out_Into_Final_Intermediate_Report();
```

Breakpoint Mode Example: PowerPC-SingleStep

Source code of the program running on the target:

```
#define RTRT_FILE int
RTRT_FILE usr_open(char *fName) { return(1); }
void usr_writeln(RTRT_FILE f,char *s) { _simo(1,s,80); }
void usr_close(RTRT_FILE f) { ; }
char atl_buffer[100];
void main(void)
{
RTRT_FILE f ;
strcpy(atl_buffer,"Hello World");
f=usr_open("cNewTdp\\atl.out");
usr_writeln(f,atl_buffer);
usr_close(f);
}
```

Execution command from host:

```
simppc.exe TestRt.cmd
```

Content of TestRt.cmd:

```
debug a.out
break usr_close
break usr_writeln -g -c "read atl_buffer >> Tmpatl.out"
go
exit
```

As in the previous example, all the usr_ functions are empty. The intelligence is deported into the TestRt.cmd script which a command file executed when the SingleStep debugger is launched.

It first loads the executable program, a.out by the debug command.

Then it sets a breakpoint at usr_close function, which serves as an exit-point, then set a breakpoint in the usr_writeln function. The -g flag of the break commmand indicates to continue the execution, whilest the -c specifies a command that should be executed before continuing. This command (read) writes the value of the atl_buffer variable into Tmpatl.out.

The SingleStep debugger then starts the execution. When it stops, it means than usr_close has been reached. It then executes the exit command, to terminate the debugging session.

The final result is contained into Tmpatl.out, and should be cleaned-up by the host (a little decoder in Perl for example) to produce the final expected cNewTdp\atl.out file containing "Hello World".

Based on the "Hello World" program, we should now focus on automating the execution step and having atl.out being written.

Retrieving Data from the Target Host

All test and runtime analysis tools of the product must be able to retrieve the value of a global (char *) variable from an application running on the target machine and then write that value to a text file on the host machine. (The variable will contain only ASCII values).

This retrieval may be the result of a specific program running on the target, of an adapted execution procedure on the host, or both.

To perform data retrieval, the program generated or instrumented by the product is linked with the Target Deployment Port data retrieval functions and type definition.

For example, in the C language, the type definition and data retrieval functions are:

```
#define RTRT_FILE <Type>
RTRT_FILE priv_init(char *fName);        /* fName: file name to be
written on the host */
RTRT_FILE priv_open(char *fName);          /* fName: file name to be
written on the host */
void priv_writeln(RTRT_FILE f,char *data); /* data is the data that
should be printed in the file */
void priv_close(RTRT_FILE f);              /* Close the host file */
```

These data retrieval functions are called by the Target Deployment Port library. Depending on the nature of the target platform, some or all of these routines may be empty.

Never ending applications

When designing embedded and real-time software, it is frequent to have applications which are designed to loop indefinitely and do not exit.

A good method for data retrieval for this type of application can be as follows:

- Set the **usr_init** routine to only return **0**.
- Set the **usr_open** routine to open the file in append mode and return the handle of the file.
- Set the **usr_write** to write the buffer **s** into the file **f**.
- Set the **usr_close** routine to close the file **f**.
- Set the **atexit** function to **NONE**
- Use the **On Function Return** setting in the **Snapshot** page of the **General Runtime Analysis** settings (or the **-DUMPRETURNING** option from the Instrumentor command line) from a function which is reachable but not executed frequently. The best is to setup a dedicated function which is executed on demand.

For this type of application, it can be useful to close the output file at the end of each dump in order to use the file for reporting in Studio even if the application is still running. To do this, compile the files with the following compiler option:

```
-D_ATCPQ_RESET=_ATCPQ_CLOSE
```

Target System Categories

Target platforms can be classified into three categories, characterized by their data-retrieval method:

- Standard Mode
- User Mode
- Breakpoint Mode

Standard Mode

This kind of target system allows use of a regular **FILE *** data type and of the **fopen**, **fprintf** and **fclose** functions found in the standard C library. Such systems include, for example, all UNIX or Windows platforms, as well as LynxOS or QNX.

If the standard C library is usable on the target, use these regular **fopen**/**fprintf**/**fclose** functions for TDP data retrieval. This is by far the easiest data retrieval option.

- If your target system is compliant with the Standard Mode category, data retrieval is assured.

User Mode

On *User Mode* systems, the standard C library calls described above are not available but other calls that send characters to the host machine are available. This could be a simple *putchar*-like function sending a character to a serial line, or it could be a method for sending a string to a simulated I/O channel, such as in the case of a microprocessor simulator.

- If your target system is using an operating system, there are usually functions that enable communication between the host machine and the target. Therefore, data retrieval capability is assured.
- If your target system allows use of a standard socket library, User Mode is always possible - thus data retrieval is assured.

Breakpoint Mode

On breakpoint mode systems, no I/O functions are available on the target platform. This is usually the case with small target calculators, such as those used in the automotive industry, running on a microprocessor simulator or emulator with no operating system.

If no communication functions are available on the target platform, the best alternative is to use a debugger logging mechanism, assuming one exists.

Note **In breakpoint mode, some compilers and linkers ignore empty functions and remove them from the final a.out binary. As the debugger must use these routines to set breakpoints, you must ensure that the linker includes these functions - any associated symbols must be in the map file. Currently, all of the priv_ functions for C and C++ contain a small amount of dummy code to avoid this issue; however, you might need to add dummy code for Ada.**

When using breakpoint mode it is necessary to implement a debugger script to perform the following actions in the given order:

- Download the executable on the target.
- Set three break points: **priv_exit**, **priv_close**, and **priv_writeln**.
- Start the execution
- Each time the **priv_writeln** break point is reached, dump the **atl_buffer** and resume.
- Quit the debugger when any other break point is reached.

Troubleshooting target deployment ports

If you are experiencing problems related to implementing a TDP on a particular target, the following troubleshooting guide might help you to find a solution.

If a problem persists, do not hesitate to contact IBM Customer support for help.

Problem	Solution
In breakpoint mode, the **priv_exit** breakpoint is never reached	The **priv_exit** breakpoint is only reached when problems are found by the TDP. See Breakpoint mode in Target system categories.
When using Runtime Analysis tools alone, no results are produced and the **atl_obstools_dump** function is never reached.	This usually occurs when the application never ends or when the entry point is not a **main** function.
When using Runtime Analysis tools alone, no results are produced and the application uses a custom exit function instead of the standard function.	Add the following line to the ANA ? `#pragma attol exit_instr = "exit","<ExitFunction>"` See Never ending applications.
When using Runtime Analysis tools alone, no results are produced and the application does not use **main** as an entry point.	When using **main** as an entry point, the instrumentor automatically adds a call to the dump at the end of this procedure. If you do not use **main** as an entry point or if this file has not been instrumented then this call is not generated. The solution is the same as for an application which never exits. See Never ending applications.
When using Runtime Analysis tools alone, no results are produced although the **atl_obstools_dump** function is reached.	Check that in the **Initialized global variable support and checks** section of the TDP the initialization is supported. If not then unset **RTRT_VARIABLE_INIT_SUPPORTED**. Check that memory is reintialized to 0 before execution. If not then set **RTRT_RAM_SET_RAMDOMLY**.
When using Runtime Analysis tools alone, no results are produced and in breakpoint mode, the **priv_close** breakpoint is never reached.	The **priv_close** breakpoint is executed at the end of the execution to close the result file. If however **atexit** is not set to **NONE**, then **priv_close** is not explicitly called.

	If the exit function does not perform the call, change the TDP to set **atexit** to **NONE**.
When I collect coverage data for a component test, either: • The viewer shows coverage for **ATU.h**, and each trace has a call to **ATU.h**, or • The viewer shows the source code, even for all included files.	The **#line x** statements are not generated properly by the compiler in the **.i** file. Find the compiler option to properly generate these #lines during the preprocessing phase.
I cannot tell whether my compiler is supported.	We have never encountered any problems supporting any C compiler whatsoever with target deployment port technology. This, however, is not true for all C++ compilers. For example, Borland C++ is not supported.
There is no TDP for Memory Profiling for Java	Memory Profiling for Java uses the JVMPI mechanism provided by the JVM and does not rely on target deployment port technology. If JVMPI is not available, for example with J2ME, there is no way to perform memory analysis. In this case, there are two options: - Implement your own JVMPI mechanism on the target JVM. - Implement another means of tracking memory usage on the target

Using the TDP Editor

Using the TDP Editor

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

The TDP Editor is made up of 4 main sections:

- **A Navigation Tree:** Use the navigation tree on the left to select customization points.
- **A Help Window:** Provides direct reference information for the selected customization point.
- **An Edit Window:** The format of the **Edit** Window depends on the nature of the customization point.
- **A Comment Window:** Lets you to enter a personal comment for each customization point.

In the Navigation Tree, you can click on any customization point to obtained detailed reference information for that parameter in the **Help** Window. Use this information to customize the TDP to suit your requirements.

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:

1. 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**.

Opening a Target Deployment Port

Target Deployment Ports can be viewed and edited with the TDP Editor supplied with Test RealTime.

To start the TDP Editor:

1. In Test RealTime, from the **Tools** menu, select **TDP Editor** and **Start**.

   or

2. From the command line, type **tdpeditor**.

To open a TDP:

1. From the **File** menu, select **Open**.
2. In the targets directory, select an **.xdp** file and click **Open**.

Creating a Target Deployment Port

To create a new Target Deployment Port (TDP), the best method is to make a copy of an existing TDP that requires minimal modifications.

Naming Conventions

By convention, the TDP directory name starts with a **c** for the C and C++ languages, **ada** for the Ada language or **j** for Java, followed by the name of the development environment, such as the compiler and target platform.

The name of the **.xdp** file generally follows the same convention.

The name of the top-level node can be a user-friendly name, as it is to be displayed in the Test RealTime GUI.

To create a new TDP:

1. In the TDP Editor, from the File menu, select New.

1. In the **Language Selection** box, select the language used for this TDP.

   The TDP Editor uses this information to create a template, which already contains most of the information required for the TDP.

2. Right click the top level node in the tree-view pane, which contains the name of the TDP. Select **Rename**. and enter a new name for this TDP. This name identifies the TDP in the Test RealTime GUI and can be more explicit than the TDP filename (see Naming Conventions).
3. As a good practice, in the **Comment** section, enter contact information such as your name and email address. This makes things easier when sharing the TDP with other users.

To save the new TDP:

1. From the **File** menu, select **Save As**.
2. Save your new TDP with a filename that follows the naming conventions described above. The actual location of the **.xdp** file is not relevant. The TDP Editor automatically creates a directory with the same name as the **.xdp** file and saves the **.xdp** file at that location.

Editing Customization Points

Use the Navigation Tree on the left to select customization points. A Target Deployment Port can be subdivided into four primary sections:

- **Basic Settings:** This section specifies default file extensions, default compilation and link flags, environment variables and custom variables required for your target environment. This section allows you to set all the common settings and variables used by Test RealTime and the different sections of the TDP. For example, the name and location of the cross compiler for your target is stored in a Basic Settings variable, which is used throughout the compilation, preprocessing and link functions. If the compiler changes, you only need to update this variable in the Basic Settings section.
- **Build Settings:** This section configures the functions required by the Test RealTime GUI integrated build process. It defines compilation, link and execution Perl scripts, plus any user-defined scripts when needed. This section is the core of the TDP, as it drives all the actions needed to compile and execute a piece of code on the target. All files related to the Build settings are stored in the TDP **cmd** subdirectory
- **Library Settings:** This section describes a set of source code files as well as a dedicated customization file (**custom.h**), which adapt the TDP to target platform requirements. This section is definitively the most complex and usually only requires customization for specialized platform TDPs (unknown RTOS, no RTOS, unknown simulator, emulator, etc.). These files are stored in the TDP **lib** subdirectory.
- **Parser Settings:** This section modifies the behavior of the parser in order to address non-standard compiler extensions, such as for example, non-ANSI extensions. This section allows Test RealTime to properly parse your source code, either for instrumentation or code generation purposes. The resulting files are stored in the TDP **ana** subdirectory.

To edit a customization point

1. In the Navigation Tree, select the customization point that you want to edit.
2. In the Help Window, read the reference information pertaining to the selected customization point. Use this information fill out the Edit window.
3. As a good practice, enter any remarks or comments in the Comments window.

After making any changes to a TDP, you must update the TDP in Test RealTime to apply the changes to a project.

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 Configuration is used.

If you make any changes to the Basic Settings of a TDP with the TDP Editor, any project settings that are read from the TDP 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**.

Using a Post-generation Script

In some cases, it can be necessary to make changes to the way the TDP is written to its directory beyond the possibilities offered by the TDP editor.

To do this, the TDP editor runs a post-generation Perl script called **postGen.pl**, which can be launched automatically at the end of the TDP directory generation process.

To use the postGen script:

1. In the TDP editor, right click on the **Build Settings** node and select **Add child** and **Ascii File**.
2. Name the new node **postGen.pl**.
3. Write a perl function performing the actions that you want to perform after the TDP directory is written by the TDP Editor.

Example

Here is a possible template for the **postGen.pl** script file:

```
sub postGen
{
     $d=shift;
#    the only parameter passed to this function is the path to the
target directory
#    here any action to be taken can be added
}
1;
```

The parameter **$d** contains <*tdp_dir*>/<*tdp_name*>, where <*tdp_dir*> is a chosen location for the TDP directory (by default, the **targets** subdirectory of the product installation directory), and <*tdp_name*> is the name of the current TDP directory

Chapter 2. Profiling with runtime analysis

The runtime analysis feature set of Test RealTime 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.

Runtime analysis overview

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

These features use source code insertion 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.
- Contract Check (for C++ only) verifies behavioral assertions during execution of the code and produces a Contract Check sequence diagram.

Each of these runtime analysis tools 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.

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

To use the runtime analysis tools:

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.
4. Select the runtime analysis tools 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 analysis and profiling reports.

The runtime analysis tools can be run within a test by simply adding the runtime analysis setting to an existing test node.

The runtime analysis tools for C and C++ can also be used in an Eclipse development environment.

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++, Ada and Java software coverage based on optimized source-code instrumentation.

How Code Coverage Works

When an application node is executed, the source code is instrumented by the Instrumentor (**attolcpp**, **attolcc1**, **attolada** or **javi**). The resulting source code is then executed and the Code Coverage feature outputs an **.fdc** and a dynamic **.tio** file.

These files can be viewed and controlled from the Test RealTime GUI. Both the **.fdc** and **.tio** files need to be opened simultaneously to view the report.

Of course, these steps are mostly transparent to the user when the test or application node is executed in the Test RealTime GUI or Eclipse (for C and C++).

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. This 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.

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		
ATC coverage		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

Block coverage

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

- Statement blocks
- Statement and decision blocks
- Statement, decision, and loop blocks

- Asynchronous transfer of control (ATC) blocks

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 ;
    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.

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.

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).

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.

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 **Rational 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 **Rational 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.

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

Block coverage

When running the Code Coverage feature on C 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 afunction whose return value depends
on the availability of a system resource, for example */
while ( ( ( result = try ())!=0 )&&
(++i <5 ));
return result;
} /* 3 logical blocks */
```

Call coverage

When analyzing C 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 ) );
      return tmp;
    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 */
```

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).

There are 4*2 basic conditions in the following example:

```
/* 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 at least 5 basic conditions in this example:

```
/* 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 ... */
  }
}
```

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;
}
```

In this example, there are 6 basic conditions (FALSE and TRUE of each), 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

Therefore the 4 following 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;
}
```

In this example, 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.

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**.

Note The last bracket '}' in a function after a return statement is always displayed in red in the coverage report, even if the function reports 100% coverage.

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

Block 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;
  }           /* potentially terminal statement */
  return 0;   /* 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.

Method 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

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.

Additional Statements

Non-coverable Statements

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

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();
  }
}
```

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

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.

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**.

Filtering coverage types

Depending on the language selected, the Code Coverage feature offers (see Coverage Types for more information):

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

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:

1. 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:

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

Coverage source report

You can use the standards keys (arrow keys, home, end, etc.) to move about and to select the source code. The Code Coverage source report displays covered and uncovered lines of code colors. You can change these colors in the Code Coverage report preferences.

Note In C source files, the last bracket '}' in a function after a return statement is always displayed as uncovered in the coverage report, even if the function reports 100% coverage.

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** tooltip.

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.

Coverage 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 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.

Cleaning code coverage report files

Code Coverage produces reports on each execution of the application under test. After many executions, the .tio coverage report files can become quite large and take up a lot of disk space.

You can use the **-CLEAN** option with the **attolcov** command to remove unused and obsolete traces and to regain some space without losing your execution history.

You can use the **-MERGETESTS** command line option to merge all the specified **.tio** coverage report files together.

To clean the .tio coverage report files:

1. Run the following command line:
   ```
   attolcov <oldfiles.tio> -clean=<newfile.tio> -mergetests
   ```
   where <oldfiles.tio> is a list of old .tio coverage report files and <newfile.tio> is the new .tio coverage report file.

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.

- 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.

How Memory Profiling for C and C++ Works

When an application node is executed, the source code is instrumented by the C or C++ Instrumentor (**attolcpp** or **attolcc1**). The resulting source code is then executed and the Memory Profiling feature outputs a static **.tsf** file for each instrumented source file and a dynamic **.tpf** file.

These files can be viewed and controlled from the Test RealTime GUI. Both the **.tsf** and **.tpf** files need to be opened simultaneously to view the report.

Of course, these steps are mostly transparent to the user when the test or application node is executed in the Test RealTime GUI or Eclipse (for C and C++).

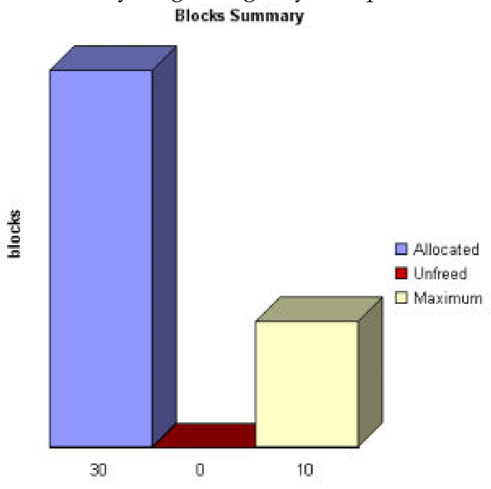
Memory Profiling Results for C and C++

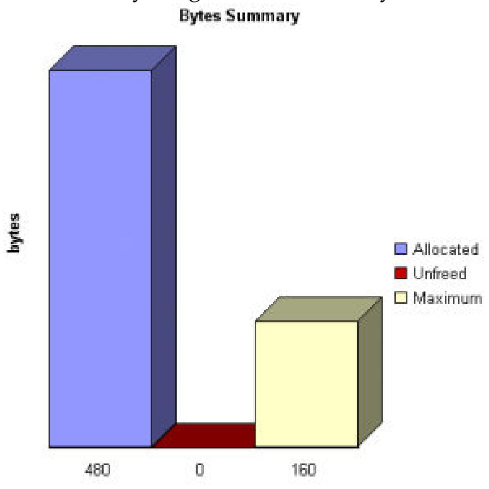
After execution of an instrumented application, the Memory Profiling report provides a summary diagram and a detailed report for both byte and memory block usage.

A memory block is a number of bytes allocated with a single *malloc* instruction. The number of bytes contained in each block is the actual amount of memory allocated by the corresponding allocation instruction.

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 Error Messages

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)
- Freeing Invalid Memory (FIM)
- 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 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

Freeing Invalid Memory (FIM)

An FIM message indicates that the program is trying to free allocated memory with the wrong instruction.

This message can occur when the memory free instruction mismatches the memory allocation instruction.

For example, a FIM occurs when memory is freed with a **free** instruction when it was allocated with a **new** instruction.

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.

Memory Profiling checks for ABWL errors whenever free() or dump() routines are called, or whenever the free queue is actually flushed.

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 at the expense of increased memory usage. You can change the Red Zone size in the Memory Profiling Settings.

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

Note Unlike IBM Rational PurifyPlus, the ABWL error in the Test RealTime Memory Profiling tool 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 Warning Messages

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 and C++ 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 and C++ 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.

Checking for ABWL and FMWL errors

By default, Memory Profiling checks for ABWL and FMWL errors whenever the routines are called, or whenever the free queue is actually flushed.

In some cases, it might be desirable to manually specify when to check for ABWL and FMWL errors, and on which functions.

By using the **ABWL and FMWL check frequency** setting you can order a check on:

- Each time the memory is dumped (by default).
- Each time a manual check macro is encountered in the code.
- Each function return.

The checks can be performed either on all memory blocks or only a selection of memory blocks.

Specifying a manual check

To indicate where you want an ABWL or FMWL check to occur in your source code, you insert an **_ATP_CHECK()** macro at the corresponding location. The syntax for the macro is:

```
#pragma attol insert _ATP_CHECK(@RELFLINE)
```

Each time this macro is encountered during execution, Memory Profiling checks for ABWL and FMWL errors on the specified blocks. The **@RELFLINE** parameter allows navigation from the Memory Profiling report to the corresponding line in the source code.

Selecting blocks to check

To create a selection of blocks that you specifically want to verify, you create a list in your source code using the **_ATP_TRACK()** macro variable. The syntax for this macro is:

```
#pragma attol insert _ATP_TRACK(<pointer>)
```

Example

A sample demonstrating how to use this feature is provided in the **ABWL Check Frequency** example project. See Example projects for more information.

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 version of Memory Profiling 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.

How Memory Profiling for Java works

When an application node is executed, the source code is executed. The Memory Profiling for Java feature uses the JVMPI mechanism to monitor the application. JVMPI outputs a dynamic **.jpt** file.

The .**jpt** file is split into a **.tpf** file and a **.txf** file, which can be viewed and controlled from the Test RealTime GUI. Both the **.tpf** and **.txf** files need to be opened simultaneously to view the report.

Of course, these steps are mostly transparent to the user when the test or application node is executed in the Test RealTime GUI.

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. 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:

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

To obtain a differential report:

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

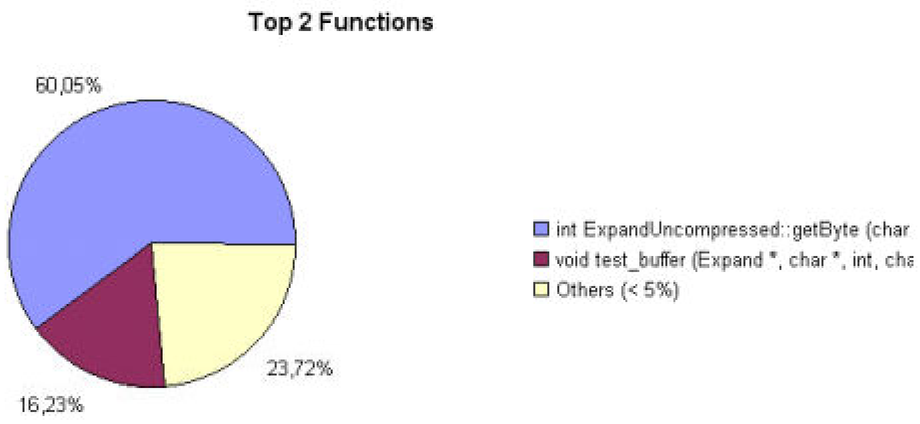
- You associate Performance Profiling with an existing test or application code.
- You build and execute your code in Test RealTime .
- The application under test, instrumented with the Performance Profiling feature, then directs output to the Performance Profiling Viewer, which a provides a detailed 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:** This value indicates the total time spent executing the function, exclusive of any calls to its descendants.
- **Function+descendant (F+D) time:** The total time spent executing the function and any of its descendants (any other functions called by this function).

  Note that since each of the descendants may have been called by other functions, it is not enough 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 each time the function was executed.
- **Min F+D:** The minimum time spent executing the function and any of its descendants.
- **Max F+D:** The maximum time spent executing the function and any of its descendants.

Note: The **Min** and **Max** values are optional, because their calculation uses a large amount of memory. To calculate these values, you must first activate the option in the Configuration Settings for the corresponding node.

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.

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:

1. From the File menu, select Export

Applying performance profile filters

Filters allow you to streamline a performance profile report by filtering out specific events. Use the **Filter List** dialog box to specify how events are to be detected and filtered.

The export and import facilities are useful if you want to share and re-use filters between Projects and users.

To access the Filter List:

1. From the **Performance Profile Viewer** menu, select **Filters** or click the **Filter** button in the Perfomance Profile 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:

1. Click the **Import** button.
2. Locate and select the **.xlf** file(s) that you want to import.
3. Click **OK**.

To export a filter event:

1. Select the filter that you want to export.
2. Click the **Export** button.
3. Select the location and name of the exported **.xlf** file.
4. Click **OK**.

Editing performance profile filters

Use the Filter Editor to create or modify filters that allow you to hide or show routines in the performance profile report, based on specified filter criteria.

By default, routines that match the filter criteria are hidden in the report. Use the **Invert filter** option to invert this behaviour: only routines that match the filter criteria are displayed.

Routine filters can be defined with one or more of the following criteria:

- **Name:** Specifies the name of a routine as the filter criteria.

- **Calls >** and **Calls <:** The number times the function was called is greater or lower than the specified value.
- **F Time >** and **F Time <:** Function time greater or lower than the specified value.
- **F+D Time >** and **F+D Time <**: Function and descendant time greater or lower than the specified value.
- **F Time (%) >** and **F Time (%) <:** Function time, expressed in percentage, greater or lower than the specified value.
- **F+D Time (%) >** and **F+D Time (%) <**: Function and descendant time, expressed in percentage, greater or lower than the specified value.
- **Average >** and **Average <:** The average time spent executing the function greater or lower than the specified value.

To define a routine filter:

1. In the Name box, specify a name for the filter.
2. Click **More** or **Fewer** to add or remove a criteria.
3. 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.
4. Add or remove a criteria by clicking the **More** or **Fewer** buttons.
5. Click **Ok**.

Runtime tracing

Runtime Tracing is a feature for monitoring real-time dynamic interaction analysis of your C, C++ and Java source code. Runtime Tracing uses exclusive Source Code Insertion (SCI) 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 Component Testing 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 a provides a real-time UML Sequence Diagram of your application's behavior.

How Runtime Tracing works

When an application node is executed, the source code is instrumented by the C, C++ or Java Instrumentor (**attolcc1**, **attolccp** or **javi**). The resulting source code is then executed and the Runtime Tracing feature outputs a static **.tsf** file for each instrumented source file as well as a dynamic **.tdf** file.

These files can be viewed and controlled from the Test RealTime GUI. Both the **.tsf** and **.tdf** files need to be opened simultaneously to view the report.

Of course, these steps are mostly transparent to the user when the test or application node is executed in the Test RealTime GUI or Eclipse (for C and C++).

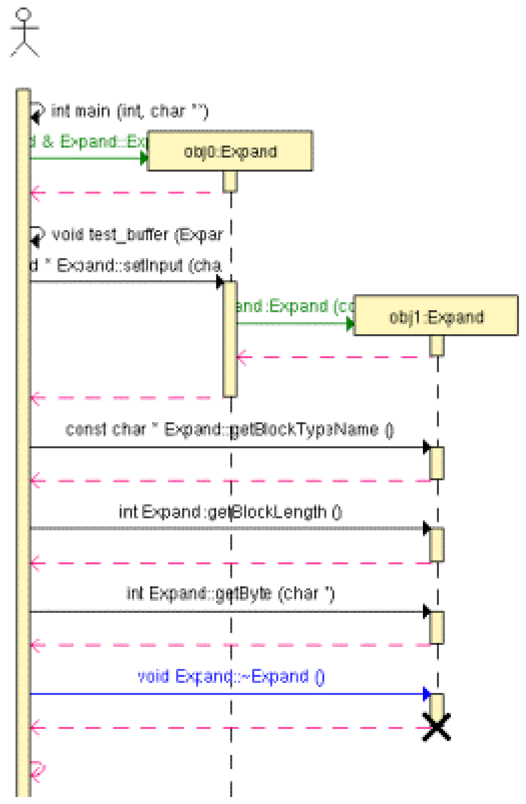
UML sequence diagram overview

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. For more information about UML sequence diagrams, see the UML sequence diagram reference.

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 as green arrow actions.
- Destructors are the blue arrows.
- 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:

1. 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:

1. Right-click an element of the object lifeline.

2. Select **Go to Head** or **Go to Destruction** in the pop-up menu.

To filter an instance out of the UML sequence diagram:

1. Right-click an element of the object lifeline.
2. Select **Filter instance** in the pop-up menu.

Tracing 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 engage 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:

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

Step-by-step tracing

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:

1. 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 ▼ 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:

1. Click the **Step** button in the UML/SD Viewer toolbar or press **F10**.

To skip to the end of execution:

1. 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:

1. Click the **Restart** button in the UML/SD toolbar.

To de-activate Step-by-Step mode

1. From the **UML/SD Viewer** menu, select **Display Mode** and **All**.

Using 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:

1. 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**.

Applying sequence diagram filters

Filters allow you to streamline a sequence diagram by filtering out specific event types. Use the Viewer's **Filter List** dialog box to specify how events are to be detected and filtered.

The export and import facilities are useful if you want to share and re-use filters between Projects and users.

To access the Filter List:

1. 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:

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:

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**.

Adding UML notes to source code

You can manually add your own notes inside your source code in order to make them display in the UML sequence diagram when runtime tracing is enabled. To do this, you must insert the following line, called an instrumentation pragma, in your C or C++ source code:

```
#pragma attol att_insert_ATT_USER_NOTE("Text")
```

This can be done automatically with the text editor.

To manually set the syntax coloring mode:

1. In a C or C++ source file, place your cursor at the line where you want a UML note to be displayed in the UML sequence diagram.
2. In the toolbar, click Add Note. This inserts the instrumentation pragma line in the source code:

3. Replace "Text" with a meaningful string that will be displayed in the note.

Viewing UML sequence diagrams

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 Diagrams

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.

Some elements in the sequence diagram provide links to the corresponding line in the source code. For example, if you click a message in a sequence diagram, the text editor opens the corresponding source file in the text editor.

Note If the source file is already open, it is not brought forward.

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.

Click the thread bar to open the **Thread Properties** window.

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**.

Thread properties

The **Thread Properties** window displays a list of all threads that are created during execution of the application. Threads are listed with the following properties:

- **Colour tab:** As displayed in the Thread Bar.
- **Thread ID:** A sequential number corresponding to the order in which each thread was created.
- **Name:** The name of the thread.
- **State**: Either **Sleeping** or **Running** state.
- **Priority:** The current priority of the thread.
- **Since:** The timestamp of the moment the thread entered the current state.

Click the title of each column to sort the list by the corresponding property

Thread Properties Filter

By default, the Thread Properties window displays the entire list of thread states during execution of the program.

To switch the Thread Properties Filter:

1. Click **Filter** to display reduce the display to the list of threads created by the application.
2. Click **Unfilter** to return the full list of thread states.

Filtering sequence diagram 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.
5. 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.
6. You can use wildcards (*) or regular expressions by selecting the corresponding option.
7. Click the button to enable or disable case sensitivity in the criteria.
8. You can add or remove a criteria by clicking the **More** or **Fewer** buttons.
9. 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 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. From the **Edit** menu, select **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.

Exporting a sequence diagram to a text file (.csv)

The UML/SD Viewer can generate sequence diagram results in a **.csv** text file. A **.csv** file is a text file presented as a table. You can import these results into a text editor, a spreadsheet application or use them to operate a file *diff* comparison for non-regression evaluation.

You can specify the format used to generate the .csv text file in the Data table preferences.

To generate a .csv text file from a sequence diagram:

1. After running an application or test node with Runtime Tracing, open a sequence diagram.
2. From the **Runtime Trace** menu, select **Generate CSV**.
3. In the **Generate CSV** window, specify the name of the text file.
4. Select **Generate columns header** to insert a line with column titles at the top of the file.
5. In the **Columns** list, select the sequence diagram elements that you want to export to the text file. Use the **Up** and **Down** buttons to change the order.
6. In the **Additional Filters** list, select any sequence diagram elements that you want to filter out of the report.
7. Click **Preview** to see how the table will appear in a spreadsheet application. The **CSV Preview** window is limited to the first 100 lines. Click **Close** to exit the preview.
8. Click **OK**.

Advanced runtime tracing

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 **Build Settings > Target Deployment Port build** 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.
- **Record and display thread info:** When selected, the UML Sequence Diagram displays a note each time a new thread is created and each time a thread's schedule is changed.

To access the multi-thread build settings:

1. In the **Project Explorer**, click the **Open Settings...** button.
2. Select one or several nodes in the Configuration pane.
3. Select **Build > Target Deployment Port build**.
4. Set the **Multi-threaded application** and **Maximum number of threads** settings.
5. Select **Runtime Analysis > Runtime Tracing > Runtime options**.
6. Set **Record and display thread info** to **Yes** or **No**.
7. When you have finished, click **OK** to validate the changes.

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.

To enable 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 **Runtime Analysis > Runtime Tracing > Runtime options**.
4. Set the **Partial Runtime Tracing flush** setting to **Yes** or **No** to activate or disable the mode.
5. When you have finished, click **OK** to validate the changes.

To toggle message dump from within the source code:

1. To do this, 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 Runtime Tracing 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.

Flushing 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. In the **Project Explorer**, select the highest level node from which you want to activate split trace mode.
2. Click the **Open Settings...** button.
3. Select **Runtime Analysis** and the **Runtime Tracing** settings.

   Select **Trace Control**.

   Set the **Size (Kb)** of each split **.tdf**. The default size is 5000 Kb.

   Specify a **File Name Prefix** for the split **.tdf** filenames. The prefix is followed by a 4-digit number that identifies each file.
4. 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.

Trace Probes for C

The Trace Probes 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.

Upon execution of the instrumented binary, the probes record information on the exchange of specified messages, including message content and a time stamp. Probe trace results can then be processed and displayed 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.

How Trace Probes work

The first step is to manually add specific macros to your C source code.

When the test or application node is executed, the Probe Processor produces an instrumented source file. which is functionally identical to the original, but which generates extra message tracing results.

The resulting source code is then executed and the Trace Probe feature outputs a **.rio** output file for each probe instance.

A **.tsf** static trace file is generated during instrumentation, and the **.rio** output file is processed and transformed into a **.tdf** file. These files can be viewed and controlled from the Test RealTime GUI. Both the **.tsf** and **.tdf** files need to be opened simultaneously to view the UML sequence diagram report.

Of course, these steps are mostly transparent to the user when the test or application node is executed in the Test RealTime GUI.

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()** macro must be called before any probe activity can occur; for example, it can be placed at the start of the application.

The **atl_end_trace()** macros must be called after all probe activity has ended; for example, when the application terminates.

Other macros must be placed inside the source code, at 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 section on Probe Macros in Reference for a complete definition of each probe macro.

To activate the Trace Probe feature:

1. In the Project Browser, select the application or System Testing node on which you want to use the feature.
2. Click Settings and open the **Probe control** 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**.

Trace Probe output modes

By default, the message traces are written to the **.rio** output file. However, in some cases, this may not be practical, therefore the Trace Probe feature can be configured to send trace information to a temporary buffer before writing to a file.

To change the way traces are stored, specify the trace mode as specified in the **Probe Control Settings**:

- **DEFAULT:** In this mode, the message traces are written directly to the **.rio** output file.
- **FIFO:** Binary format traces are directed to a temporary *first-in first-out* memory buffer before writing to the **.rio** file when the **atl_dump_trace** macro is encountered. This mode is intended for embedded or realtime applications which may not be able to access a filesystem when running.
- **FILE:** Binary format traces are written to a low footprint temporary file before writing to the **.rio** file when the **atl_dump_trace** macro is encountered. This mode is intended for embedded or realtime applications which may not have enough memory or processing power to continuously write to the **.rio** file. In this case for example, a second application could be set up to read the file and generate the **.rio** result file.
- **USER:** Uses methods, described in a user-defined **probecst.c** file to direct traces to a user-defined format before writing to the **.rio** file when the **atl_dump_trace** macro is encountered. See Customizing the USER output mode for more information.
- **IGNORE:** Use this setting to ignore trace probe macros on compilation. In this case, the binary is compiled without instrumentation.

When **FIFO**, **FILE** or **USER** are selected, the traces must be flushed to the **.rio** file with a specific **atl_dump_trace** macro placed in a source file.

Use the **DEFAULT** output mode whenever possible. In most other cases, the **FIFO** or **FILE** should be enough and can be optimized using parameters provided in the Reference section.

Only use **USER** mode if none of the other settings are practical for your application. Using the USER output mode requires that you rewrite your own **probecst.c** and **probecst.h** using the files provided with the product as a template. See Customizing the USER output mode for more information.

When using the **USER** mode, you must specify the location of the user-defined **probecst.c** and **probecst.h** files in the **USER** custom files directory setting. See Probe control settings for details.

Traces Probes and System Testing for C

You can use Trace Probes to produce a System Testing .pts test script based on probe activity.

When a probed application is executed, the **.rio** result file is processed, which produces a **.pts** test script for System Testing for C.

The Script generation flags setting allows you to specify the command line arguments that will be used to generate the .pts test script. The available flags are:

```
-accuracy=<time>
-polling=<time>
```

These values express the desired accuracy and polling intervals to be used in the **.pts** test script, where <*time*> is expressed in milliseconds (ms).

You can edit and reuse this script in later tests to replay the exact same data exchanges in a System Testing for C test node.

Customizing the USER output mode

The **USER** output mode for Trace Probes requires that you rewrite user-defined **probecst.c** and **probecst.h** based on the files provided with the product.

Only use the **USER** mode if the **DEFAULT**, **FIFO** or **FILE** modes are not practical for your application.

To rewrite your own routines, make a copy of the **probecst.c** and **probecst.h** that are provided with the product and use them as a template. These files are located in the following directory located in the installation directory of the product:

```
/lib/probe/probecst/fifo
```

Note These are the files that are used for the **FIFO** output mode, therefore ensure that any changes that you make are performed on copies of these files.

The implementation delivered in the **FIFO** mechanism is based on a circular buffer. The instrumented application sends traces to the intermediate storage buffer, by using the **atl_write_probe** function. The traces can then be read by the **atl_read_probe** function.

You can modify this file to adapt the probe mechanism to your application and platform.

For example, when using **USER** mode, the main probed application may store messages in binary format in a shared memory or pipe, whereas a dedicated "dump application" can be written to read the shared memory or pipe and to generate the **.rio** result file.

By using this method, the probed application can still run with minimal overhead while another process generates the **.rio** result file either on the fly or after the execution of the probed application.

Whichever storage mechanism you use, it is important that the dump application runs within the same hardware architecture as the main application to avoid misalignment or little-big endian problems.

When using the **USER** mode, you must specify the location of the user-defined **probecst.c** and **probecst.h** files in the **USER** custom files directory setting. See Probe control settings for details.

The **probecst.c** file contains definitions for the Trace Probe macro functions. These are detailed below. For the usage and syntax of the Trace Probe macros, please refer to the Reference section. For each function, the **probecst.c** file contains comments that should help you to rewrite each of these functions.

The following functions must be executed during the execution of the probed application:

- atl_create_probe
- atl_end_probe
- atl_write_key
- atl_write_probe

The following functions can be executed when the probed application ends or after the application has finished in a dedicated dump application:

- atl_open_probe
- atl_close_probe
- atl_read_probe

atl_start_trace

The atl_start_trace function executes atl_create_probe. It must be called before any other macros, once for each instance. Its role is to open, create and initialize the intermediate storage media used to keep messages in the intermediate binary format.

atl_end_trace

The **atl_start_trace** function executes **atl_end_probe**. It must be called at the end of the application, once for each instance. Its role is to close the intermediate storage media used to keep messages in the intermediate binary format.

atl_send_trace and atl_recv_trace

The **atl_send_trace** and **atl_recv_trace** functions execute **atl_write_probe** in order to dump the message to the intermediate storage media.

It is important that the **.rio** result file retains the message sequence. Therefore, ensure that data is recorded in the execution order.

atl_write_probe

The role of the **atl_write_probe** function is to record the following data:

- The complete message, the length of the message is provided to help.
- The date of the event.
- An internal code.
- The key format.

If your USER mechanism required the use of intermediate storage, the **atl_dump_trace** must be called after the **atl_end_trace** macro.

atl_dump_trace()

This macro can be either part of the probed application or part of a dedicated dump program that would be executed after the main application, depending on what is practical in your application.

The **atl_dump_trace()** macro executes, for each instance,

- atl_open_probe,
- **atl_read_probe** for each recorded message, and
- atl_close_probe.

atl_open_probe

The role of the **atl_open_probe** function is to reopen the intermediate storage and point to the first recorded message.

atl_close_probe

The role of the **atl_close_probe** function is to close, destroy or free the memory of the intermediate storage.

atl_read_probe

The role of the **atl_read_probe** function is to retrieve the following data from the intermediate storage:

- The message as it was recorded during the execution.
- A timestamp of the message.
- An internal code.
- The key format of the message.

atl_select_trace

The role of the **atl_select_trace** function is to execute **atl_write_key** in the API. The code of this function must not be customized. It must be copied from the original **probecst.c** without any change.

Profiling shared libraries

In order to perform runtime analysis on a shared library, you must create an application node containing both a small program that uses the library, and a reference link to the library.

After the execution of the application node, the runtime analysis results are located in the application node.

To profile a shared library:

1. Add the library to your project as described in Using library nodes.
2. Create an empty application node:
3. Right-click a group or project node and select **Add Child** and **Application** from the popup menu.
4. Enter the name of the application node
5. Inside the application node, create a source file containing a short program that uses the shared library.
6. Link the application node to the shared library:
7. Right-click the application or test node that will use the shared library and select **Add Child** and **Reference** from the popup menu.
8. Select the library node and click **OK**.
9. Select the application node, click the Settings button, and set the Build options to include the runtime analysis tools that you want to use.
10. Build and execute the application node.

Example

An example demonstrating how to use Runtime Analysis tools on shared libraries is provided in the **Shared Library** example project. See Example projects for more information.

Chapter 3. Checking with static analysis

The static analysis features of IBM Rational Test RealTime allow you to measure the complexity of your source code and to check the adherence to coding guidelines.

These tools are able analyze the source code providing without compiling or running the application.

- Static metrics provide statistic indicators of code complexity.
- Code review performs in-depth verification of the source code against a set of rules that implement best practices, coding guidelines, and standards.

These static analysis features can be used together with any of the automated testing features and runtime analysis features.

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

To use the static analysis features:

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.
4. Select the runtime analysis tools 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 analysis and profiling reports.

The runtime analysis options can be run within a test by simply adding the runtime analysis setting to an existing test node.

Static metrics

Statistical measurement of source code properties is an extremely important matter when you are planning a test campaign or for project management purposes. Test RealTime provides a Metrics Viewer, which displays detailed source code complexity data and statistics for your C, C++, Ada, and Java source code.

How the static metrics tool works

Metrics are updated each time a file is modified. Static metrics can be computed each time a node is built, but can also be calculated without executing the application.

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

Viewing Static Metrics

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.

To compute static metrics without executing the application:

1. In the **Project Browser**, select a node.
2. From the **Build** menu, select **Options** or click the **Build Options** button in the toolbar.
3. Clear all build options. Select only **Source compilation** and **Static metrics**.
4. Click the **Build** toolbar button.

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.

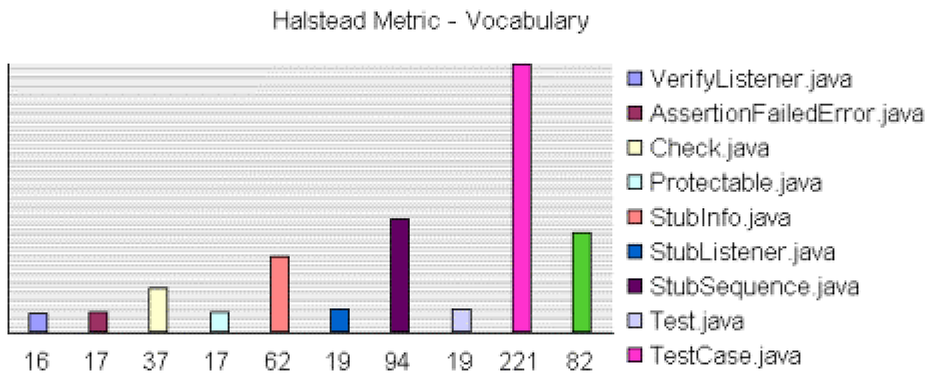
- **Total statements:** total number of statement in child nodes
- **Maximum statements:** the maximum number of statements
- **Maximum level:** the maximum nesting level
- **Maximum V(g):** the highest encountered cyclomatic number
- **Mean V(g):** the average cyclomatic number
- **Standard deviation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

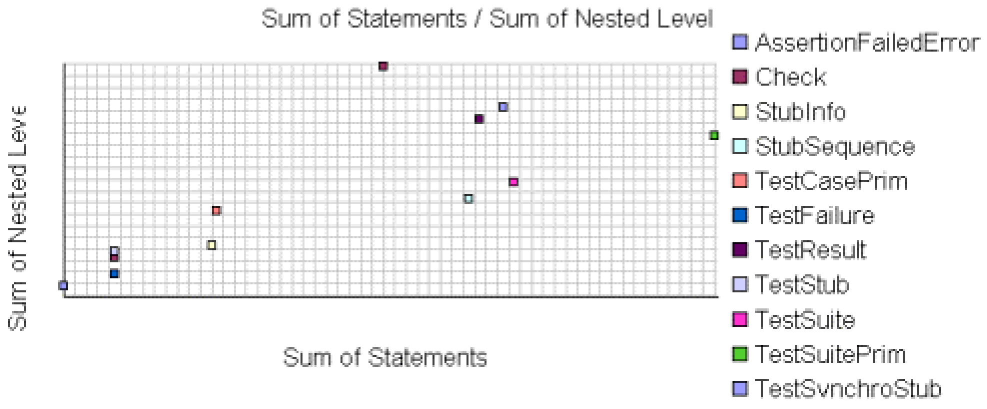
1. From the **Metrics** menu, select **Halstead Graph for Root Page**.
2. Select another metric to display.

Object view

Root Level Summary

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

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 operator instances
N_2	Number of operand 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	$N_2 + N_2$
V	Volume	$N * \log2\ n$
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 formulae, k is the *stroud* number, which has an arbitrary default value of 18. With experience, you can adjust the stroud number to adapt the calculation of the estimated testing time (T) to your own 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.

Code review

Automated source code review is a method of analyzing code against a set of predefined rules to ensure that the source adheres to guidelines and standards that are part of any efficient quality control strategy. Test RealTime provides an automated code review tool, which reports on adherence to guidelines for your C source code.

Among other guidelines, the code review tool implements rules from the MISRA-C:2004 standard, **Guidelines for the use of the C language in critical systems**.

How the code review tool works

When an application or test node is built, the source code is analyzed by the code review tool. The tool checks the source file against the predefined rules and produces a **.crc** report file that can be viewed and controlled from the Test RealTime GUI.

Of course, these steps are mostly transparent to the user when the test or application node is built in the Test RealTime GUI.

Code review can be performed each time a node is built, but can also be calculated without executing the application.

Configuring code review rules

The code review tool uses a set of predefined rules that you can either disable or set the severity level to Warning or Error.

By default all rules are enabled and produce either an error or a warning in the code review report. You can save multiple customized rule policies.

The default rule policy is located in the <*installation directory*>**/plugins/Common/lib/confrule.xml** file.

> **Note** All new projects use the default rule configuration file. When you make any changes to the policy, a new policy file is automatically saved in the project folder.

To disable or set the severity level of code review rules:

1. Select a node in the **Project Explorer** pane and click the **Settings** button.
2. In the Configuration Settings list, select **Code Review**.
3. Select the Rule configuration setting and click **Edit**. This opens the **Rule Configuration** window.
4. On the left, select the rule that you want to configure. Rules are grouped into categories. When a rule is selected, its description is displayed on the right.
5. On the right, select the severity level:

- **Disabled**: The selected rule is ignored and is not displayed in the code review report.
- **Warning**: When any non-compliances are found, a warning is displayed in the code review report.
- **Error**: When any non-compliances are found, an error is displayed in the code review report.

6. Select **Show only the first occurrence** to only show the first occurrence of a non conformance in a file.
7. Select **OK** to save the current configuration or **Save As** to create a new rule configuration file.

Running a code review

You can use the code review tool on any test or application node or a single source file. The code review tool is run on the source code whenever you build the file.

To enable the code review tool on a source file, test or application node:

1. In the **Project Browser**, select the node that you want to review.
2. Select **Build** > **Options** from the menu or click the **Settings** button and select **Build** > **Build options**.
3. Select **Code Review**.

To perform a code review without compiling and executing the application:

1. In the **Project Browser**, select the node that you want to check.
2. Select **Build** > **Options** from the menu or click the **Settings** button and select **Build** > **Build options**.
3. Clear all build options except **Code Review**.
4. Click the **Build** toolbar button.

Viewing code review results

The GUI displays code review results in the Report Viewer.

Reloading a Report

If a code review report has been updated since the moment you have opened it in the report viewer, you can use the Reload command to refresh the display:

To reload a report:

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

Exporting a Report to HTML

Code review results can be exported to an HTML file.

To export results to an HTML file:

1. From the **File** menu, select **Export** and **Export Project Report in HTML files format**.
2. In the **HTML Export Configuration** window, select **Code Review**.
3. Specify an output directory and click **Export**.

Ignoring a rule on a portion of code

In some cases, it can be useful to temporarily ignore a rule non-conformance on a short portion of source code, while providing a justification of why you are allowing the non-conformance.

You can justify a non-conformance in the source code, for a specified number of lines and for the first or all occurrences of the error, by adding the following pragma lines to your source code:

- **#pragma attol crc_justify:** ignores the first occurrence of a specified non conformance
- **#pragma attol crc_justify_all:** ignores all occurrences of a specified non conformance

You must provide an explanation of why you are ignoring the rule. The justification text is included in the code review report.

To justify a rule discrepancy on a portion of code:

1. Open the source file in the text editor and locate the lines of code that you want the rule to ignore.
2. Before the portion of code, add the following line:
   ```
   #pragma attol crc_justify (<rule>[,<lines>],"<text>")
   ```
   to justify the first non-conformance encountered, or
   ```
   #pragma attol crc_justify_all (<rule>,<lines>,"<text>")
   ```
   to justify all non conformances, where:

- <*rule*> is the name of the code review rule (for example: "Rule M8.5").
- <*lines*> is the number of lines. By default the pragma only applies to the next line.
- <*text*> is the justification of why the rule is ignored here.

Example

The following example causes all non-conformances to the rule M8.5 in the 3 next lines to be ignored and explained in the code review report.

```
#pragma attol crc_justify_all (Rule M8.5, 3, "Rule M8.5 does not apply
to the 3 following lines")
```

Understanding code review reports

The Code Review report lists the rules that produced and error or a warning.

Report explorer

The Report Explorer window displays a list of rules that were broken for each source file and function. You can use there elements in this view to navigate through the report.

Report summary

At the top of the Code Review report a summary provides information about the general configuration, the date and the number of analyzed files.

It also lists the number of errors and warnings that were encountered.

Code review details

The code review report lists the rules for which errors or warnings were detected. It also provides information about the location of the error. You can click the title to go directly to the corresponding line in the source code.

Chapter 4. Testing software components

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.
- Component Testing for Java uses the JUnit framework to address J2ME and J2SE 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 tools.

To use a component 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

The Component Testing for C feature of Test RealTime provides a unique, fully automated, and proven solution for applications written in C, dramatically increasing test productivity.

How Component Testing for C Works

When a test node is executed, the Component Testing Test Compiler (**attolpreproC**) compiles both the test scripts and the source under test. This preprocessing creates a **.tdc** file. The resulting source code generates a test driver.

If any Runtime Analysis tools are associated with the test node, then the source code is also instrumented with the Instrumentor (**attolcc1**) tool.

The test driver, TDP, stubs and dependency files all make up the test harness.

The test harness interacts with the source code under test and produces test results. Test execution creates a **.rio** file.

The **.tdc** and **.rio** files are processed together the Component Testing Report Generator (**attolpostpro**). The output is the **.xrd** report file, which can be viewed and controlled in the Test RealTime GUI.

Of course, these steps are mostly transparent to the user when the test node is executed in the Test RealTime GUI.

Component Testing for C Overview

Component Testing for C interacts with your source code through the C Test Script Language. The **Rational Test RealTime Reference Manual** contains full reference information about each of these languages.

Testing with Component Testing for C is as simple as following these steps:

- Set up your test project in the GUI
- Write a **.ptu** test script
- Run your tests
- View the results.

Upon execution, 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.

Integrated, Simulated and Additional Files

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.

Testing shared libraries

In order to test a shared library, you must create a test node containing the .ptu component test script that uses the library, and a reference link to the library.

After the execution of the test node, the runtime analysis and component test results are located in the application node.

To test a shared library:

1. Add the library to your project:
2. Right-click a group or project node and select **Add Child** and **Library** from the popup menu.
3. Enter the name of the Library node
4. Right-click the Library node and select **Add Child** and **Files** from the popup menu.
5. Select the source files of the shared library.
6. Run the Component Testing wizard as usual on the source file of your library. This creates a test node containing the test scripts and the source file.
7. Delete the source file from the test node.
8. Create a reference to the shared library in the test node:
9. Right-click the application or test node that will use the shared library and select **Add Child** and **Reference** from the popup menu.
10. Select the library node and click **OK**.
11. Build and execute the test node.

Example

An example demonstrating how to test shared libraries is provided in the **Shared Library** example project. See Example projects for more information.

Writing a Test Script

When you first create Component Testing for C test node with the Component Testing Wizard, Test RealTime produces a **.ptu** test script template based on the source under test.

To write the test script, you can use the Text Editor provided with Test RealTime.

Component Testing for C uses the C Test Script Language. Full reference information for this language is provided in the Reference section.

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.

Test script filenames must contain only plain alphanumerical characters.

A typical Component Testing **.ptu** test script looks like this:

```
HEADER add, 1, 1
<variable declarations for the test script>
BEGIN
SERVICE add
  <local variable declarations for the service>
  TEST 1
  FAMILY nominal
  ELEMENT
    VAR variable1, INIT=0, EV=0
    VAR variable2, INIT=0, EV=0
    #<call to the procedure under test>
  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 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. In this case, the ampersand must be the very first character on that line; no spaces or tabs should precede it.
- Statements must be shorter than 2048 characters, although this limit may be lower on some platforms.

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 related 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**.
- **FAMILY:** Qualifies the test case to which it is attached. The qualification is free (in this case **nominal**). A list of qualifications can be specified (for example: **family**, **nominal**, **structure**) in the Tester Configuration dialog box.
- **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.

Using native C statements

In some cases, it can be necessary to include portions of C native code inside a **.ptu** test script for one the following reasons:

- To declare native variables that participate in the flow of a scenario. Such statements must be analyzed by the Component Testing Parser.
- To insert native code into a scenario. In this case, the code is ignored by the parser, but carried on into the generated code.

Analyzed native code

Lines prefixed with a # character are analyzed by Component Testing Parser.

Prefix statements with a **#** character to include native C variable declarations as well as any code that can be analyzed by the parser.

```
#int i;
#char *foo;
```

Variable declarations must be placed outside of Component Testing Language blocks or preferably at the beginning of scenarios and procedures.

Ignored native code

Lines prefixed with a @ character are ignored by the parser, but copied into the generated code.

To use native C code in the test script, start instructions with a @ character:

```
@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.

Automatically updating a .ptu test script

Changes that are made during the development process can sometimes impact the test script, for example when new functions are added after the test script was generated.

You can update a .ptu test script to automatically add new elements to SERVICES and INCLUDE blocks to reflect changes that were made to the source code. An update does not remove or modify any existing statements.

For the update to work, you must not edit any generated comment lines that start with **%c** or **%d** in the test script. The update command only works with **.ptu** test scripts that were generated with the Test RealTime 7.0.0 component test wizard that contain these **%c** and **%d** comment lines.

To update a .ptu test script

1. In the **Project Explorer**, right-click the **.ptu** test script that you want to update.
2. From the pop-up menu, select **Update**.
3. Edit the **.ptu** test script.

Testing variables

One of the main features of Component Testing for C is its ability to compare initial values, expected values and actual values of variables during test execution. In the C Test Script Language, this is done with the **VAR** statement.

The **VAR** statement specifies both the test start-up procedure and the post-execution test for simple variables. This instruction uses three parameters:

- **Name of the variable under test:** this can be a simple variable, an array element, or a field of a record. It is also possible to test an entire array, part of an array or all the fields of a record.
- Initial value of the variable: identified by the keyword INIT.
- Expected value of the variable after the procedure has been executed: identified by the keyword EV.

Declare variables under test with the **VAR** statement, followed by the declaration keywords:

- **INIT =** for an assignment
- **INIT ==** for no initialization
- **EV =** for a simple test.

It does not matter where the **VAR** instructions are located with respect to the test procedure call since the C code generator separates **VAR** instructions into two parts :

- The variable test is initialized with the **ELEMENT** instruction
- The actual test against the expected value is done with the **END ELEMENT** instruction

Many other forms are available that enable you to create more complex test scenarios.

Using C Expressions

Component Testing for C allows you to define initial and expected values with standard C expressions.

All literal values, variable types, functions and most operators available in the C language are accepted by Component Testing for C.

Example

The following example demonstrates typical use of the **VAR** statement

```
HEADER add, 1, 1
#with add;
BEGIN
SERVICE add
 # a, b, c : integer;
 TEST 1
  FAMILY nominal
  ELEMENT
   VAR a, init = 1, ev = init
   VAR b, init = 3, ev = init
   VAR c, init = 0, ev = 4
   #c := add(a,b);
  END ELEMENT
 END TEST
END SERVICE
```

Testing intervals

You can test an expected value within a given interval by replacing **EV** with the keywords **MIN** and **MAX**.

You can also use this form on alphanumeric variables, where character strings are considered in alphabetical order ("**A**"<"**B**"<"**C**").

Example

The following example demonstrates how to test a value within an interval:

```
TEST 4
FAMILY nominal
ELEMENT
VAR a, INIT in {1,2,3}, EV = INIT
VAR b, INIT = 3, EV = INIT
VAR c, INIT = 0, MIN = 4, MAX = 6
#c = add(a,b);
END ELEMENT
END TEST
```

Testing tolerances

You can associate a tolerance with an expected value for numerical variables. To do this, use the keyword **DELTA** with the expected value **EV**.

This tolerance can either be an absolute value (the default option) or relative (in the form of a percentage <*value*>%).

You can rewrite the test from the previous example as follows:

```
TEST 5
FAMILY nominal
ELEMENT
VAR a, INIT in {1,2,3}, EV = INIT
VAR b, INIT = 3, EV = INIT
VAR c, INIT = 0, EV = 5, DELTA = 1
#c = add(a,b);
END ELEMENT
END TEST
```

or

```
TEST 6
FAMILY nominal
ELEMENT
VAR a, INIT in {1,2,3}, EV = INIT
VAR b, INIT = 3, EV = INIT
VAR c, INIT = 0, EV = 5, DELTA = 20%
#c = add(a,b);
END ELEMENT
END TEST
```

Initializing without testing

It is sometimes difficult to predict the expected result for a variable; such as if a variable holds the current date or time. In this case, you can avoid specifying an expected output.

Example

The following script show an example of an omitted test:

```
TEST 7
FAMILY nominal
ELEMENT
VAR a, init in {1,2,3}, ev = init
VAR b, init = 3, ev = init
VAR c, init = 0, ev ==
#c = add(a,b);
END ELEMENT
END TEST
```

Testing expressions

To test the return value of an expression, rather than declaring a local variable to memorize the value under test, you can directly test the return value with the **VAR** instruction.

In some cases, you must leave out the initialization part of the instruction.

Example

The following example places the call of the ***add*** function in a **VAR** statement:

```
TEST 12
FAMILY nominal
ELEMENT
VAR a, init in {1,2,3}, ev = init
VAR b, init(a) with {3,2,1}, ev = init
VAR add(a,b), ev = 4
END ELEMENT
END TEST
```

In this example, you no longer need the variable ***c***.

All syntax examples of expected values are still applicable, even in this particular case.

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.

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.

Initial and Expected Value settings

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.

Arrays

Testing Arrays

With Component Testing for C, you can test arrays in quite the same way as you test variables. In the C Test Script Language, this is done with the **ARRAY** statement.

The **ARRAY** statement specifies both the test start-up procedure and the post-execution test for simple variables. This instruction uses three parameters:

- **Name of the variable under test:** species the name of the array in any of the following ways:
- To test one array element, conform to the C syntax: **histo[0]**.
- To test the entire array without specifying its bounds, the size of the array is deduced by analyzing its declaration. This can only be done for well-defined arrays.
- To test a part of the array, specify the lower and upper bounds within which the test will be run, separated with two periods (**..**), as in: `histo[1..SIZE_HISTO]`
- Initial value of the array: identified by the keyword INIT.
- Expected value of the array after the procedure has been executed: identified by the keyword EV.

Declare variables under test with the **ARRAY** statement, followed by the declaration keywords:

- **INIT =** for an assignment
- **INIT ==** for no initialization
- **EV =** for a simple test.

It does not matter where the **ARRAY** instructions are located with respect to the test procedure call since the C code generator separates **ARRAY** instructions into two parts :

- The array test is initialized with the **ELEMENT** instruction
- The actual test against the expected value is done with the **END ELEMENT** instruction

To initialize and test an array, specify the same value for all the array elements.

You can use the same expressions for initial and expected values as those used for simple variables (literal values, constants, variables, functions, and C operators).

Use the **ARRAY** instruction to run simple tests on all or only some of the elements in an array.

Testing Arrays with C Expressions

To initialize and test an array, specify the same value for all the array elements. The following two examples illustrate this simple form.

```
ARRAY image, INIT = 0, EV = INIT
ARRAY histo[1..SIZE_HISTO-1], INIT = 0, EV = 0
```

You can use the same expressions for initial and expected values as those used for simple variables (literal values, constants, variables, functions, and C operators).

Example

The following example highlights the ARRAY instruction syntax for C:

```
HEADER histo, 1, 1
##include "histo.h"
BEGIN
SERVICE COMPUTE_HISTO
  #int x1, x2, y1, y2;
  #int status;
  #T_HISTO histo;
  TEST 1
    FAMILY nominal
    ELEMENT
      VAR x1, init = 0, ev = init
      VAR x2, init = SIZE_IMAGE-1, ev = init
      VAR y1, init = 0, ev = init
      VAR y2, init = SIZE_IMAGE-1, ev = init
      ARRAY image, init = 0, ev = init
      VAR histo[0], init = 0, ev = SIZE_IMAGE*SIZE_IMAGE
      ARRAY histo[1..SIZE_HISTO-1], init = 0, ev = 0
      VAR status, init = 0, ev = 0
      #status = compute_histo(x1, y1, x2, y2, histo);
    END ELEMENT
  END TEST
END SERVICE
```

Testing arrays with pseudo-variables

Another form of initialization consists of using one or more pseudo-variables, as the following example illustrates:

```
TEST 3
  FAMILY nominal
  ELEMENT
    VAR x1, init = 0, ev = init
    VAR x2, init = SIZE_IMAGE•1, ev = init
    VAR y1, init = 0, ev = init
    VAR y2, init = SIZE_IMAGE•1, ev = init
    ARRAY image, init=(int)(100*(1+sin((float)(I1+I2)))), ev = init
    ARRAY histo[0..200], init = 0, ev ==
    ARRAY histo[201..SIZE_HISTO•1], init = 0, ev = 0
    VAR status, init ==, ev = 0
    #status = compute_histo(x1, y1, x2, y2, histo);
  END ELEMENT
END TEST
```

I1 and I2 are two pseudo-variables which take as their value the current values of the array indices (for **image**, from **0** to **199** for **I1** and **I2**). You can use these pseudo-variables like a standard variable in any C expression.

This lets you create more complicated test scripts in the case of very large arrays, where the use of enumerated expressions is limited.

For multidimensional arrays, you can combine these different types of initialization and test expressions, as the following example shows:

```
ARRAY image, init = {0 => I2, 1 => { 0 => 100, others => 0 },
& others => (I1 + I2) % 255 }
```

Testing large arrays

The maximum number of array elements that can be processed is 100. If you need to test an array that contains more than 100 elements, then you must split the initialization of the array over two or more initializations, as shown in the following example.

Example

The following initiatialization produces a **Too many INIT or VA values** error:

```
#int a[200];
ARRAY a, init=
{1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,20,1,2,3,4,5,6,7,8,9,30,1,2,3,
4,
5,6,7,8,9,40,1,2,3,4,5,6,7,8,9,50,1,2,3,4,5,6,7,8,9,60,1,2,3,4,5,6,7,8
,9,
70,1,2,3,4,5,6,7,8,9,80,1,2,3,4,5,6,7,8,9,90,1,2,3,4,5,6,7,8,9,100,1,2
,3,
4,5,6,7,8,9,110,1,2,3,4,5,6,7,8,9,120,1,2,3,4,5,6,7,8,9,130,1,2,3,4,5,
6,
7,8,9,140,1,2,3,4,5,6,7,8,9,150,1,2,3,4,5,6,7,8,9,160,1,2,3,4,5,6,7,8,
9,
170,1,2,3,4,5,6,7,8,9,180,1,2,3,4,5,6,7,8,9,190,1,2,3,4,5,6,7,8,9,200}
, ev=init
```

Instead, use the following expression:

```
#int a[200];
ARRAY z [0..99],
init={1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,20,1,2,3,4,5,6,7,8,9,30,1
,2
,3,4,5,6,7,8,9,40,1,2,3,4,5,6,7,8,9,50,1,2,3,4,5,6,7,8,9,60,1,2,3,4,5,
6,
7,8,9,70,1,2,3,4,5,6,7,8,9,80,1,2,3,4,5,6,7,8,9,90,1,2,3,4,5,6,7,8,9,1
00}
, ev=init
ARRAY z [100..199],
init={1,2,3,4,5,6,7,8,9,110,1,2,3,4,5,6,7,8,9,120,1,2,3,4,5,6,7,8,9,13
0,
1,2,3,4,5,6,7,8,9,140,1,2,3,4,5,6,7,8,9,150,1,2,3,4,5,6,7,8,9,160,1,2,
3,
4,5,6,7,8,9,170,1,2,3,4,5,6,7,8,9,180,1,2,3,4,5,6,7,8,9,190,1,2,3,4,5,
6,
7,8,9,200}
, ev=init
```

Testing arrays with lists

While an expression initializes all the **ARRAY** elements in the same way, you can also initialize each element using an enumerated list of expressions between brackets ({}). In this case, you must specify a value for each array element.

Furthermore, you can precede every element in this list of initial or expected values with the array index of the element concerned followed by the characters "**=>**". The following example illustrates this form:

```
ARRAY histo[0..3], init = {0 => 0, 1 => 10, 2 => 100, 3 => 10} ...
```

This form of writing the **ARRAY** instruction has the following advantages:

- It improves the readability of the list.
- It allows you to mix values without worrying about the order.

You can also use this form together with the simple form if you follow this rule: once one element has been defined with its array index, you must do the same with all the following elements.

If several elements in an array are to take the same value, specify the range of elements taking this value as follows:

```
ARRAY histo[0..3], init = { 0 .. 2 => 10, 3 => 10 } ...
```

You can also specify a value for all the as yet undefined elements by using the keyword **others**, as the following example illustrates:

```
TEST 2
  FAMILY nominal
  ELEMENT
    VAR x1, init = 0, ev = init
    VAR x2, init = SIZE_IMAGE-1, ev = init
    VAR y1, init = 0, ev = init
    VAR y2, init = SIZE_IMAGE-1, ev = init
    ARRAY image, init = {others=>{others=>100}}, ev = init
    ARRAY histo, init = 0,
    & ev = {100=>SIZE_IMAGE*SIZE_IMAGE, others=>0}
    VAR status, init ==, ev = 0
    #status = compute_histo(x1, y1, x2, y2, histo);
  END ELEMENT
END TEST
```

Note The form **{others =>** *<expression>*} is equivalent to initializing and testing all array elements with the same expression.

You can also initialize and test multidimensional arrays with a list of expressions, as follows. In this case, the previously mentioned rules apply to each dimension.

```
ARRAY image, init = {0, 1=>4, others=>{1, 2, others=>100}} ...
```

Note Some C compilers allow you to omit levels of brackets when initializing a multidimensional array. The Unit Testing Scripting Language does not accept this non-standard extension to the language.

Testing character arrays

Character arrays are a special case. Variables of this type are processed as character strings delimited by quotes.

You therefore need to initialize and test character arrays using character strings, as the following list example illustrates.

If you want to test character arrays like other arrays, you must use a format modification declaration (**FORMAT** instruction) to change them to arrays of integers.

Example

The following list example illustrates this type of modification:

```
TEST 2
  FAMILY nominal
  FORMAT T_LIST.str[] = int
  ELEMENT
    VAR l, init = NIL, ev = NONIL
    VAR s, init = "myfoo", ev = init
    VAR l•>str[0..5], init == , ev = {'m','y','f','o','o',0}
    #l = push(l,s);
  END ELEMENT
END TEST
```

Testing arrays with other arrays

The following example illustrates a form of initialization that consists of initializing or comparing an array with another array that has the same declaration:

```
TEST 4
  FAMILY nominal
  ELEMENT
    VAR x1, init = 0, ev = init
    VAR x2, init = SIZE_IMAGE•1, ev = init
    VAR y1, init = 0, ev = init
    VAR y2, init = SIZE_IMAGE•1, ev = init
    ARRAY image, init = extern_image, ev = init
    ARRAY histo, init = 0, ev ==
    VAR status, init ==, ev = 0
    #read_image(extern_image,"image.bmp");
    #status = compute_histo(x1, y1, x2, y2, histo);
  END ELEMENT
END TEST
```

Read_image and **extern_image** are two arrays that have been declared in the same way. Every element from the **extern_image** array is assigned to the corresponding **read_image** array element.

You can use this form of initialization and testing with one or more array dimensions.

Testing arrays of union elements

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
```

Structured Variables

Testing structured variables

To test all the fields of a structured variable, use a single instruction (**STR**) to define their initializations and expected values:

```
TEST 2
FAMILY nominal
ELEMENT
VAR l, init = NIL, ev = NONIL
STR *l, init == , ev = {"myfoo",NIL,NIL}
VAR s, init = "myfoo", ev = init
#l = push(l,s);
END ELEMENT
END TEST
```

You can only initialize and test structured variables with the following forms:

- INIT =
- INIT ==
- EV =
- EV ==

If a field of a structured variable needs to be initialized or tested in a different way, you can omit its initial and expected values from the global test of the structured variable, and run a separate test on this field.

The following example illustrates this:

```
TEST 4
  FAMILY nominal
  ELEMENT
    VAR l, init = NIL, ev = NONIL
    VAR *l, init == , ev = {,NIL,NIL}
    VAR s, init in {"foo","bar"}, ev = init
    VAR l->str, init ==, ev(s) in {"foo","bar"}
    #l = push(l,s);
  END ELEMENT
END TEST
```

Using field names, write this as follows:

```
VAR *l, init ==, ev = {next=>NIL,prev=>NIL}
```

Testing structured variables with C expressions

To initialize and test a structured variable or record, initialize or test all the fields using a list of native language expressions (one per field). The following example (taken from list.ptu) illustrates this form:

```
STR *l, init == , ev = {"myfoo",NIL,NIL}
```

Each element in the list must correspond to the structured variable field as it was declared.

Every expression in the list must obey the rules described so far, according to the type of field being initialized and tested:

- An expression for simple fields or arrays of simple variables initialized using an expression
- A list of expressions for array fields initialized using an enumerated list
- A list of expressions for structured fields

Using Field Names in Native Expressions

You can specify field names in native expressions by following the field name of the structure with the characters "**=>**", as follows:

```
TEST 3
  FAMILY nominal
  ELEMENT
    VAR l, init = NIL, ev = NONIL
    VAR *l, init == , ev = {str=>"myfoo",next=>NIL,prev=>NIL}
    VAR s, init = "myfoo", ev = init
    #l = push(l,s);
  END ELEMENT
END TEST
```

If you use this form, you do not have to respect the order of expressions in the list.

You can also use the position of the fields in the structure or record instead of the field names, on the basis that the field numbers begin at 1:

```
VAR *l, init ==, ev = {3 => NIL, 2 => NIL, 1 => "myfoo"}
```

As with arrays, you can also use a range for field positions, as follows:

```
VAR *l, init ==, ev = {1 => "myfoo", 2..3 => NIL}
```

Testing structured variables with other structured variables

You can initialize and test a structured variable or record using another structured variable or record of the same type. The following example illustrates this form:

```
STR *l, init == , ev = l1
```

Each field of the structured variable will be initialized or tested using the associated fields of the variable used for initialization or testing.

Omitting a Field's Initial and Test Values

You can only initialize and test structured variables with the following forms:

- INIT =
- INIT ==
- EV =
- EV ==

If a field of a structured variable needs to be initialized or tested in a different way, you can omit its initial and expected values from the global test of the structured variable, and run a separate test on this field.

The following example illustrates this:

```
TEST 4
  FAMILY nominal
  ELEMENT
    VAR l, init = NIL, ev = NONIL
    VAR *l, init == , ev = {,NIL,NIL}
    VAR s, init in {"foo","bar"}, ev = init
    VAR l->str, init ==, ev(s) in {"foo","bar"}
    #l = push(l,s);
  END ELEMENT
END TEST
```

Using field names, write this as follows:

```
VAR *l, init ==, ev = {next=>NIL,prev=>NIL}
```

C Unions

If the structured variable involves a C union (defined using the **union** instruction) rather than a structure (defined using the **struct** instruction), you need to specify which field in the union is tested. The initial and test value only relates to one of the fields in the union, whereas, for a structure, it relates to all the fields.

The **list.c** example demonstrates this if you modify the structure of the list, such that the value stored at each node is an integer, a floating-point number, or a character string:

list1.h:

```
enum node_type { INTEGER, REAL, STRING };
typedef struct t_list {
 enum node_type type;
 union {
 long integer_value;
 double real_value;
 char * string_value;
 } value;
 struct t_list * next;
 struct t_list * prev;
} T_LIST, * PT_LIST;
```

In this case, the test becomes:

```
HEADER list1, 1, 1
##include "list1.h"
BEGIN
SERVICE push1
  #PT_LIST l;
  #enum node_type t;
  #char s[10];
    TEST 1
    FAMILY nominal
    ELEMENT
      VAR l, init = NIL, ev = NONIL
      VAR t, init = my_string, ev = init
      VAR *l, init == ,
      & ev = {STRING,{string_value=>"myfoo"}, NIL,NIL}
      VAR s, init = "myfoo", ev = init
      #l = push1(l, t, s);
    END ELEMENT
  END TEST
END SERVICE
```

The use of **string_value =>** indicates that the chosen field in the union is **string_value**.

If no field is specified, the first field in the union is taken by default.

Stub Simulation

Stub simulation is based on the idea certain functions are to be simulated and are therefore replaced with other functions which are generated in the test driver. These generated functions, or stubs, have the same interface as the simulated functions, but the body of the functions is replaced.

These stubs have the following roles:

- Store input values to simulated functions
- Assign output values 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
- the method of passing each parameter (input, output, or input/output).

When using the Component Testing Wizard, you specify the functions that you want to stub. This automatically adds the corresponding code to the **.ptu** test script. On execution of the test, Component Testing for C generates the stub in the test driver, which includes:

- a variable array for the input values of the stub
- a variable array for the output values of the stub
- a body declaration for the stub function

Function Prototypes

When generating a stub for a function, Test RealTime considers the first prototype of the function that is encountered, which can be:

- The declaration of the function in an included header file.
- The declaration **DEFINE STUB** statement in the **.ptu** test script.

This means that the declaration of the function contained in the **DEFINE STUB s**tatement is ignored if the function was previously declared in a header file.

If an existing body of stubbed function is encountered, Test RealTime renames the existing body to **atl_stub_**<*function-name*> and the stubbed version of the function is used in the test driver.

Passing Parameters

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

Example

An example project called **Stub C** is available from the Examples section of the Start page. This example demonstrates the use of stubs in Component Testing for C. See Example projects for more information.

Stub Definition

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.

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;  /* simulated global variable */
 #char fic_err_msg[100]; /* simulated global variable */
 #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. The global variables must be initialized within the **.ptu** test script.

Stub Usage

Use the **STUB** statement to declare that you want to use a stub rather than the original function. You can use 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 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

Example

The following example illustrates use of a stub which simulates file access.

```
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 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 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.

Multiple stub calls

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
```

Multiple stub calls

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 stub calls

To check that a **STUB** is never called, even if an **ENVIRONMENT** containing the **STUB** is used, use the following syntax:

```
STUB write_file 0=>(4,"line")
```

No testing of the number of stub calls

If you do not 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 that are 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
```

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.

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.

Advanced Stubs

This section covers some of the more complex notions when dealing with stub simulations in Component Testing for Ada.

Creating Complex Stubs in C

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.

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 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
```

Simulating Functions with Varying Parameters

In some cases, functions may be designed to 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 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 **STUB**s. 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
```

Simulating Functions with *void** Parameters

When stubbing a function that takes void* type parameters, such as as `fct_sim(double c, void * d)`, the Source Code Parser generates incomplete code that might not compile.

Using `void* _out` means that the stub has to dereference a pointer to void, which is not possible.

When you are stubbing functions that take void* parameters, you must check and edit the **.ptu** test script in order to specify the real type that the stub has to dereference.

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 should 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
```

Simulating Functions with char* parameters

You can use Component Testing for C 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, the product treats all variables of this type as C strings. To specify a different behavior, you must use one of the following methods.

Pointers

Use the **FORMAT** command to specify that the test required is that of a pointer. For example:

```
HEADER charp, ,
#extern int CharPointer(char* pChar);
BEGIN

SERVICE CharPointer1

#char  *Chars;
#int ret;

TEST 1

ELEMENT
FORMAT Chars = void*
VAR Chars,    init = NIL,  ev = init
VAR ret,      init = 0,    ev = 0

#ret = CharPointer(Chars);
END ELEMENT

END TEST -- TEST 1

END SERVICE -- CharPointer1
```

Pointers to a Single char

Define the type as **_inout**, as in the following example.

```
HEADER charp, ,
#extern int CharPointer(char* pChar);
BEGIN

SERVICE CharPointer1

#char  AChar;
#int ret;
```

```
TEST 1

ELEMENT
VAR AChar,  init = 'A', ev = init
VAR ret,    init = 0,   ev = 'A'

#ret = CharPointer(&AChar);
END ELEMENT

END TEST -- TEST 1

END SERVICE -- CharPointer1
```

Arrays of Characters of Indeterminate Size

Use the **FORMAT** command to specify that the array is in fact an array of unsigned chars not chars, as the product considers that char arrays are C strings. For example:

```
HEADER charp, ,
#extern int CharPointer(char* pChar);
BEGIN

SERVICE CharPointer1

#char  Chars[4];
#int ret;

TEST 1

ELEMENT
FORMAT Chars = unsigned char[4]
ARRAY Chars,  init = {'a','b','c','d'},  ev = init
VAR ret,    init = 0,                  ev = 'a'

#ret = CharPointer(Chars);
END ELEMENT

END TEST -- TEST 1

END SERVICE -- CharPointer1
```

C strings

Use an array of characters in which the last character is the character '\0', a C string.

```
HEADER charp, ,
#extern int CharPointer(char* pChar);
BEGIN

SERVICE CharPointer1

#char  Chars[10];
#int ret;

TEST 1

ELEMENT
VAR Chars,  init = "Hello",  ev  = init
VAR ret,    init = 0,        ev = 'H'

#ret = CharPointer(Chars);
END ELEMENT

END TEST -- TEST 1

END SERVICE -- CharPointer1
```

Environments

Testing 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
```

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.

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]*.

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.

Advanced C Testing

Advanced C Testing

This section covers some of the more complex notions behind Component Testing for C.

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.

Testing Long Types

Test RealTime does not support 64-bit *long* types as standard. The **long long** and **_int64** types do not exist in the C Testing Language. However, a workaround does permit the use of long types within a **.ptu** test script.

1. Locate the **ana/atus_c.def** file in the TDP directory and verify that the following customization point exists.

   ```
   #define _int64 long
   ```

   If the line does not exist, you must add this customization point to the **ana/atus_c.def** file.

2. Locate the following line:

   ```
   #pragma attol sizeof(long)=32
   ```

   and replace the line with the two following lines:

   ```
   #pragma attol sizeof(long)=64
   #pragma attol sizeof(int)=64
   ```

   If the line does not exist, you must add both lines to the **ana/atus_c.def** file.

3. Within the **.ptu** test script, append an **L** to the notation of initial and expected *long* values, and use **h64** to format the results. For example:

   ```
   VAR MyVarLong, long#h64, init = 0xAAAAAAAAAAAAAAAAL, ev =
   0x0FFFFFFFFFFFFFFFL
   ```

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**.

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
```

Testing a String Pointer as a Pointer

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.

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.
```

Importing legacy 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 file format used by IBM Rational Test RealTime.

This means that any **.prj**, **.cmp**, and **.ses** files created with pre-v2002 versions of the product must be imported and converted in order to be used in a current 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. You can use previous Component Testing for C++ and System Testing tests in your current 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 V2001A Component Testing files that you wish to import. To import a complete UniTest or Test RealTime v2001A project, you must select all the **.prj**, .**cmp**, and **.ses** files from that project.
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 **Component Testing for C** 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.

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 **SERVICE**s or **TEST**s as defined in the **.ptu** test script. During execution, the Component Testing node plays the selected **SERVICE**s or **TEST**s.

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.

Viewing Reports

After test execution, depending on the options selected, a series of Component Testing for C test reports are produced.

Understanding Component Testing Reports

Test reports for Component Testing are displayed in the Test RealTime 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

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 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.

Comparing C Test Reports

The Component Testing comparison capability allows you to compare the results of the last two consecutive tests.

To activate the comparison mode, select **Compare 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.

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.

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 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.

  Note: Contract Check is part of the Component Testing for C++ feature. However, contract check scripts can also be used in application nodes, as a Runtime Analysis feature.

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.

How Component Testing for C++ Works

When a test node is executed, the Component Testing Test Compiler (**atoprepro**) compiles both the test scripts and the source under test. This preprocessing creates an **.ots** file. The resulting source code generates a test driver.

If any Runtime Analysis tools are associated with the test node, then the source code is also instrumented with the Instrumentor (**attolcpp**) tool.

The test driver, TDP, stubs and dependency files all make up the test harness.

The test harness interacts with the source code under test and produces test results. Test execution creates a **.tdf** file.

The **.ots** and **.tdf** files are processed together the Component Testing Report Generator (**atopospro**). The output is the **.xrd** report file, which can be viewed and controlled in the Test RealTime GUI.

Of course, these steps are mostly transparent to the user when the test node is executed in the Test RealTime GUI.

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++.

A typical Contract Check **.otc** test script is structured as follows:

```
CLASS <class to wrap>
```

```
{
   WRAP <method>
   REQUIRE <expression>
   ENSURE <expression>
   WRAP <method>
   REQUIRE <expression>
   ENSURE <expression>
}
```

See the Reference section for the semantics of the C++ Contract Check Language.

Note When an **.otc** contract check script is used in a test node, the related source files are always instrumented even if they are displayed as not instrumented in Project Explorer.

Contract Check in a Component Test

You can use the Component Testing wizard to set up a test node and create the C++ contract-check script templates or you can manually create a Component Testing for C++ test node to reuse existing test scripts.

The **.otc** contract-check script must be executed before an **.otd** Test Driver script, therefore the order in which both script types appear in the Test node is critical. This is important if you are manually creating a test node.

Contract Check Runtime Analysis

C++ Contract Check scripts can also be used in a simple application node.

In this case, you can either copy the **.otc** contract from an existing C++ component test node, or you can create an **.otc** contract check script manually.

The **.otc** contract-check script must be placed before any other item in the application node.

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++.

A typical Component Testing **.otd** test script looks like this:

```
INCLUDE "Test.dcl";
TEST CLASS TestClass1 {
    PROLOGUE   {
        <Declarations of variables>
        <Actions to be performed before executing this test class.>
    }
    TEST CASE Test1  {
        #method_under_test();
        CHECK (expression_must_be_true == true);
    }
    EPILOGUE   {
        <Actions to be performed when leaving the test class>
    }
    RUN {
        Test1;
    }
}
RUN {
    TestClass1   (File<char*>);
}
```

See the Reference section for the semantics of the C++ test driver language.

You can use the Component Testing wizard to set up a test node and create the C++ Test Driver script templates or you can manually create a Component Testing for C++ test node to reuse existing test scripts.

An **.otc** contract-check script must be executed before an .**otd** Test Driver script, therefore the order in which both script types appear in the Test node is critical. This is important if you are manually creating a test node.

See the Reference section 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.

Using native C++ statements

In some cases, it can be necessary to include portions of C++ native code inside an **.otc** or **.otd** test script for one the following reasons:

- To declare native variables that participate in the flow of a scenario. Such statements must be analyzed by the Component Testing Parser.
- To insert native code into a scenario. In this case, the code is ignored by the parser, but carried on into the generated code.

Analyzed native code

Lines prefixed with a # character are analyzed by Component Testing Parser.

Prefix statements with a **#** character to include native C++ variable declarations as well as any code that can be analyzed by the parser.

```
#int i;
#char *foo;
```

Variable declarations must be placed outside of Component Testing Language blocks or preferably at the beginning of scenarios and procedures.

Ignored native code

Lines prefixed with a @ character are ignored by the parser, but copied into the generated code.

To use native C++ code in the test script, start instructions with a @ character:

```
@for(i=0; i++; i<100) func(i);
@foo(a,&b,c);
```

You can add native code either inside or outside of C++ Test Script Language blocks.

Additional and included files

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:

- Additional files
- Included files

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.

Functions which are not located in an additional file or in a tested file are simulated by Component Testing for C++.

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.

To specify included files while creating a test node:

1. Select a valid C++ configuration and run the Component Testing wizard.
2. On the **Test Script Generation Settings** page (Step 3/5), expand **Components Under Test** and ***<Test Name>***. where *<Test Name>* is the name of the Test Node.
3. Scroll down the list to **Included Files**, select the value field and click the '**...**' button to enter a list of files.
4. Enter any other advanced settings and continue with the Component Testing wizard.

To specify additional files while creating a test node:

1. Select a valid C++ configuration and run the Component Testing wizard.
2. On the **Test Script Generation Settings** page (Step 3/5), select **General** and switch the **Test Mode** setting to **Expert Mode**.
3. Expand **Components Under Test** and select Test Boundaries.
4. Under **Additional Files or Directories**, select the value field and click the '**...**' button to enter a list of files or directories
5. Enter any other advanced settings and continue with the Component Testing wizard.

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.

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 appear in the Component Testing for C++ test node.

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.

Error Handling

An error may be generated by either native code or any of the following instructions in a test script:

- CHECK
- CHECK PROPERTY
- CHECK EXCEPTION
- CHECK STUB
- CHECK METHOD
- REQUIRE
- ENSURE
- Native statement

Refer to each of these keywords to see when the instructions generate an error.

Error handling behavior is specified with the keyword ON ERROR. According to the choice specified by ON ERROR, the script may continue normal execution, skip the current block, or exit.

Test Results

When no errors occur during execution of a C++ Test Script Language script, the script receives *Passed* status. Otherwise, it is considered *Failed*.

When the test is completed, the errors appear in the Report Viewer or in the UML/SD Viewer as red notes.

Template Classes

Component Testing for C++ supports assertions only for fully generic and fully specialized template classes. Partial specializations are not supported.

A contract referring to a generic template class is applied to every instance of this template class, unless a specific contract has been defined for an instance of this template class.

There may be a state machine description associated with the template class, and another with a template specialization. In such a case, the latter applies to the specific template instance, and the first applies to any other instance.

Same mechanism for invariant definition (There may be invariants associated with the template class, and other invariants with a template specialization. In such a case, the latter ones apply to the specific template instance, while the first one apply to any other instance.)

A wrap defined within a generic template class contract does not apply to specialization of the associated method. If you want to test a method specialization, you must define a WRAP into the contract associated to the class instance the method specialization belongs to.

It is not possible to define WRAPs for template methods within a non-template class.

Specialization

Specialized templates are templates for which some of the parameters are real. Full-specialization of a template is an instance of the template (all parameters are real).

Example

```
template <class T,int N> class C; // generic template, not a
specialization
template <class T> class C<T,2>; // partial specialization (not
supported by Component Testing for C++)
```

```
template <> class C<char *,2>; // full-specialization
```

Note When using full-specializations, latest ISO/IEC C++ standards suggest using the template prefix **template<>**.

Testing shared libraries

In order to test a shared library, you must create a test node containing the **.otd** component test script that uses the library, and a reference link to the library.

After the execution of the test node, the runtime analysis and component test results are located in the application node.

To test a shared library:

1. Add the library to your project:
2. Right-click a group or project node and select **Add Child** and **Library** from the popup menu.
3. Enter the name of the Library node
4. Right-click the Library node and select **Add Child** and **Files** from the popup menu.
5. Select the source files of the shared library.
6. Run the Component Testing wizard as usual on the source file of your library. This creates a test node containing the **.otc** and **.otd** test scripts and the source file.
7. Delete the source file from the test node.
8. Create a reference to the shared library in the test node:
9. Right-click the application or test node that will use the shared library and select **Add Child** and **Reference** from the popup menu.
10. Select the library node and click **OK**.
11. Build and execute the test node.

Example

An example demonstrating how to test shared libraries is provided in the **Shared Library** example project. See Example projects for more information.

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.

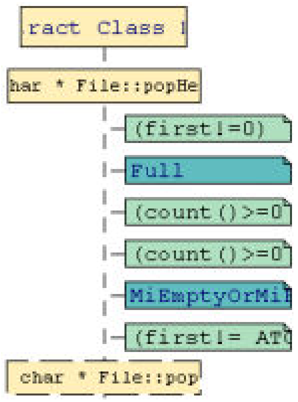
During the test, the object is continuously tested to comply to the user-defined **STATE**s and **TRANSITION**s. There are three possible cases.

- 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.

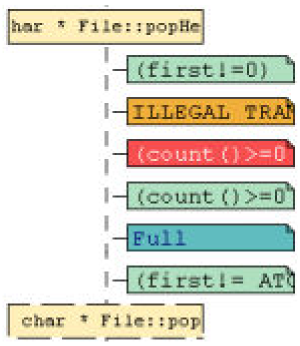
Contract-Checks

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.

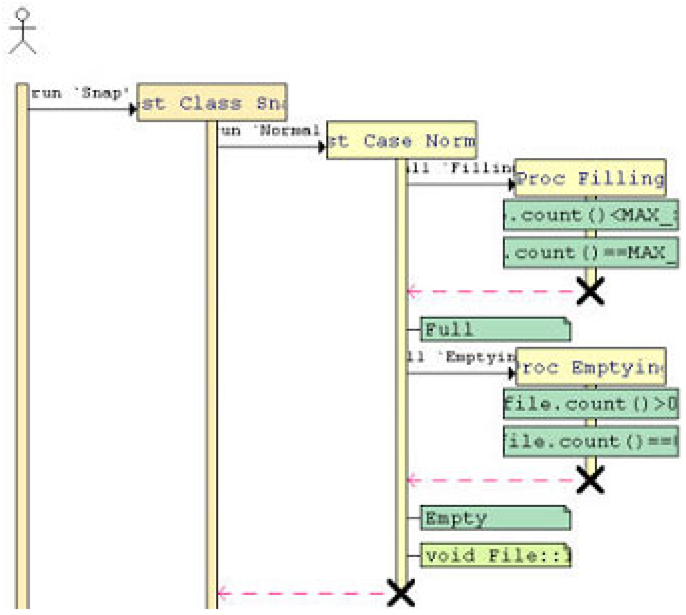
Illegal and Multiple Transitions

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 Passed ("✓") or Failed ("✗") glyphs attached to the instances.

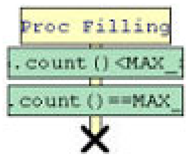
You can click any of these glyphs to highlight the corresponding statement in the Text Editor window.

- 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

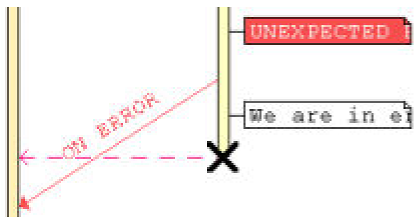
Exceptions

Component Testing for C++ generates **UNEXPECTED EXCEPTION** Notes whenever an unexpected exception is encountered. These notes will be followed by the **ON ERROR** condition.

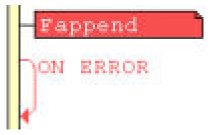
Error Handling

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:

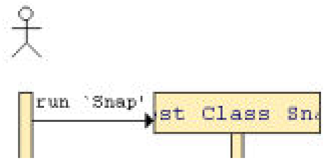


Comments and Prints

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 Ada

The Component Testing feature of Test RealTime provides a unique, fully automated, and proven solution for the Ada language, dramatically increasing test productivity.

Component Testing for Ada overview

Basically, Component Testing for Ada interacts with your source code through the Ada Test Script Language. The **Rational Test RealTime Reference Manual** contains full reference information on the Test Script Languages.

Testing with Component Testing for Ada is as simple as following these steps:

- Set up your test project in the GUI
- Write a **.ptu** test script
- Run your tests
- View the results.

How Component Testing for Ada Works

When a test node is executed, the Component Testing Test Compiler (**attolpreproADA**) compiles both the test scripts and the source under test. This preprocessing creates a **.tdc** file. The resulting source code generates a test driver.

If any Runtime Analysis tools are associated with the test node, then the source code is also instrumented with the Instrumentor (**attolada**) tool.

The test driver, TDP, stubs and dependency files all make up the test harness.

The test harness interacts with the source code under test and produces test results. Test execution creates a **.rio** file.

The **.tdc** and **.rio** files are processed together the Component Testing Report Generator (**attolpostpro**). The output is the **.xrd** report file, which can be viewed and controlled in the Test RealTime GUI.

Of course, these steps are mostly transparent to the user when the test node is executed in the Test RealTime GUI.

Integrated, simulated and additional Files

When creating a Component Testing test node for 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 creates a default test which is added to an environment named after the file in the **.ptu** 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 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.

You can toggle instrumentation of a source file by using the Properties Window dialog box.

Component Testing test selection

The Test Selection dialog box allows you to configure the Component Testing test driver.

To open the Test Selection dialog box:

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

Service/Test Tab

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

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.

Initial and expected value settings

The Initial and Expected Value settings are part of the Component Testing Settings for Ada dialog box and describe 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.

Writing a test script

When you first create Component Testing for Ada test node with the Component Testing Wizard, Test RealTime produces a **.ptu** test script template based on the source under test.

To write the test script, you can use the Text Editor provided with Test RealTime.

Component Testing for Ada uses the Ada Test Script Language. Full reference information for this language is provided in the **Rational Test RealTime Reference Manual.**

Test Script Structure

The Ada 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.

All instructions in a test script have the following characteristics:

- All statements begin with a keyword.
- Statements are not case sensitive.
- 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 2048 characters, although this limit may be lower on some platforms.

The basic structure of a Component Testing **.ptu** test script for Ada looks like this:

```
HEADER add, 1, 1
<variable declarations for the test script>
BEGIN
SERVICE add
  <local variable declarations for the service>
  TEST 1
  FAMILY nominal
  ELEMENT
    VAR variable1, INIT=0, EV=0
    VAR variable2, INIT=0, EV=0
    #<call to the procedure under test>
  END ELEMENT
  END TEST
END SERVICE
```

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 related 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**.
- **FAMILY:** Qualifies the test case to which it is attached. The qualification is free (in this case **nominal**). A list of qualifications can be specified (for example: **family**, **nominal**, **structure**) in the Tester Configuration dialog box.
- **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.

Using native Ada statements

In some cases, it can be necessary to include portions of Ada native code inside a **.ptu** test script for one the following reasons:

- To declare native variables that participate in the flow of a scenario. Such statements must be analyzed by the Component Testing Parser.
- To insert native code into a scenario. In this case, the code is ignored by the parser, but carried on into the generated code.

Analyzed native code

Lines prefixed with a **#** character are analyzed by Component Testing Parser.

Prefix statements with a **#** character to include native Ada variable declarations as well as any code that can be analyzed by the parser.

```
#int i;
#char *foo;
```

Variable declarations must be placed outside of Component Testing Language blocks or preferably at the beginning of scenarios and procedures.

Ignored native code

Lines prefixed with a @ character are ignored by the parser, but copied into the generated code.

To use native Ada code in the test script, start instructions with a @ character:

```
@for(i=0; i++; i<100) func(i);
@foo(a,&b,c);
```

You can add native code either inside or outside of Ada Test Script Language blocks.

Testing variables

One of the main features of Component Testing for Ada is its ability to compare initial values, expected values and actual values of variables during test execution. In the Ada Test Script Language, this is done with the **VAR** statement.

The **VAR** statement specifies both the test start-up procedure and the post-execution test for simple variables. This instruction uses three parameters:

- **Name of the variable under test:** this can be a simple variable, an array element, or a field of a record. It is also possible to test an entire array, part of an array or all the fields of a record.
- Initial value of the variable: identified by the keyword INIT.
- Expected value of the variable after the procedure has been executed: identified by the keyword EV.

Declare variables under test with the **VAR** statement, followed by the declaration keywords:

- **INIT =** for an assignment
- **INIT ==** for no initialization
- **EV =** for a simple test.

Component Testing for Ada allows you to define initial and expected values with standard Ada expressions.

All literal values, variable types, functions and most operators available in the Ada language are accepted by Component Testing for Ada.

It does not matter where the **VAR** instructions are located with respect to the test procedure call since the Ada code generator separates **VAR** instructions into two parts :

- The variable test is initialized with the **ELEMENT** instruction
- The actual test against the expected value is done with the **END ELEMENT** instruction

Many other forms are available that enable you to create more complex test scenarios.

Example

The following example demonstrates typical use of the **VAR** statement

```
HEADER add, 1, 1
#with add;
BEGIN
SERVICE add
 # a, b, c : integer;
 TEST 1
  FAMILY nominal
  ELEMENT
   VAR a, init = 1, ev = init
   VAR b, init = 3, ev = init
   VAR c, init = 0, ev = 4
   #c := add(a,b);
  END ELEMENT
 END TEST
END SERVICE
```

Testing Intervals

You can test an expected value within a given interval by replacing **EV** with the keywords **MIN** and **MAX**.

You can also use this form on alphanumeric variables, where character strings are considered in alphabetical order ("**A**"<="**B**"<="**C**").

Example

The following example demonstrates how to test a value within an interval:

```
TEST 4
 FAMILY nominal
 ELEMENT
  VAR a, init in (1,2,3), ev = init
  VAR b, init = 3, ev = init
  VAR c, init = 0, min = 4, max = 6
  #c = add(a,b);
 END ELEMENT
END TEST
```

Testing Tolerances

You can associate a tolerance with an expected value for numerical variables. To do this, use the keyword **DELTA** with the expected value **EV**.

This tolerance can either be an absolute value (the default option) or relative (in the form of a percentage <*value*>%).

Example

```
 TEST 5
 FAMILY nominal
```

```
ELEMENT
VAR a, INIT in (1,2,3), EV = INIT
VAR b, INIT = 3, EV = INIT
VAR c, INIT = 0, EV = 5, DELTA = 1
#c = add(a,b);
END ELEMENT
END TEST
```

or

```
TEST 6
FAMILY nominal
ELEMENT
VAR a, INIT in (1,2,3), EV = INIT
VAR b, INIT = 3, EV = INIT
VAR c, INIT = 0, EV = 5, DELTA = 20%
#c = add(a,b);
END ELEMENT
END TEST
```

Testing expressions

To test the return value of an expression, rather than declaring a local variable to memorize the value under test, you can directly test the return value with the VAR instruction.

In some cases, you must leave out the initialization part of the instruction.

Example

The following example places the call of the ***add*** function in a **VAR** statement:

```
TEST 12
 FAMILY nominal
 ELEMENT
  VAR a, init = 1, ev = init
  VAR b, init = 3, ev = init
  VAR add(a,b), ev = 4
 END ELEMENT
FIN TEST
```

In this example, you no longer need the variable ***c***. The resulting test report an *Unknown* ?status indicating that it has not been tested.

All syntax examples of expected values are still applicable, even in this particular case.

Initializing without testing

It is sometimes difficult to predict the expected result for a variable; such as if a variable holds the current date or time. In this case, you might want to avoid specifying an expected output but still have the value of the variable initialized in the test script. To do this, use the **EV ==** syntax.

Example

In the following script **a**, **b**, and **c** are initialized, but only **a** and **b** are tested.

```
TEST 7
 FAMILY nominal
 ELEMENT
  VAR a, init in (1,2,3), ev = init
  VAR b, init = 3, ev = init
  VAR c, init = 0, ev ==
  #c = add(a,b);
 END ELEMENT
END TEST
```

Declaring global variables for testing

The Target Deployment Ports for Ada do not provide any variables that can be used freely by the tester.

To avoid having to modify the code under test, it is easier to add an extra C 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.

Testing arrays

With Component Testing for Ada, you can test arrays in quite the same way as you test variables. In the Ada Test Script Language, this is done with the **ARRAY** statement.

The **ARRAY** statement specifies both the test start-up procedure and the post-execution test for simple variables. This instruction uses three parameters:

- **Name of the variable under test:** species the name of the array in any of the following ways:
- To test one array element, conform to the Ada syntax: **histo(0)**.
- To test the entire array without specifying its bounds, the size of the array is deduced by analyzing its declaration. This can only be done for well-defined arrays.
- To test a part of the array, specify the lower and upper bounds within which the test will be run, separated with two periods (**..**), as in: `histo(1..SIZE_HISTO)`
- Initial value of the array: identified by the keyword INIT.
- Expected value of the array after the procedure has been executed: identified by the keyword EV.

Declare variables under test with the **ARRAY** statement, followed by the declaration keywords:

- **INIT =** for an assignment
- **INIT ==** for no initialization
- **EV =** for a simple test.

It does not matter where the **ARRAY** instructions are located with respect to the test procedure call since the Ada code generator separates **ARRAY** instructions into two parts :

- The array test is initialized with the **ELEMENT** instruction
- The actual test against the expected value is done with the **END ELEMENT** instruction

Testing an Array with Ada Expressions

To initialize and test an array, specify the same value for all the array elements. The following two examples illustrate this simple form.

```
ARRAY image, init = 0, ev = init
ARRAY histo[1..SIZE_HISTO-1], init = 0, ev = 0
```

You can use the same expressions for initial and expected values as those used for simple variables (literal values, constants, variables, functions, and Ada operators).

Example

```
HEADER histo, 1, 1
#with histo; use histo;
BEGIN
SERVICE COMPUTE_HISTO
  # x1, x2, y1, y2 : integer;
  # histo : T_HISTO;
  TEST 1
    FAMILY nominal
    ELEMENT
      VAR x1, init = 0, ev = init
      VAR x2, init = SIZE_IMAGE•1, ev = init
      VAR y1, init = 0, ev = init
      VAR y2, init = SIZE_IMAGE•1, ev = init
      ARRAY image(1..200,1..200), init = 0, ev = init
      VAR histo(1), init = 0, ev = SIZE_IMAGE*SIZE_IMAGE
      ARRAY histo(1..SIZE_HISTO), init = 0, ev = 0
      #compute_histo(x1, y1, x2, y2, histo);
    END ELEMENT
  END TEST
END SERVICE
```

Testing arrays with pseudo-variables

Another form of initialization consists of using one or more pseudo-variables, as the following example illustrates:

```
TEST 3
 FAMILY nominal
 ELEMENT
  VAR x1, init = 0, ev = init
  VAR x2, init = SIZE_IMAGE-1, ev = init
  VAR y1, init = 0, ev = init
  VAR y2, init = SIZE_IMAGE-1, ev = init
  ARRAY image, init=(int)(100*(1+sin((float)(I1+I2)))), ev = init
  ARRAY histo[0..200], init = 0, ev ==
  ARRAY histo[201..SIZE_HISTO-1], init = 0, ev = 0
  VAR status, init ==, ev = 0
  #status = compute_histo(x1, y1, x2, y2, histo);
 END ELEMENT
END TEST
```

I1 and **I2** are two pseudo-variables which take as their value the current values of the array indices (for image, from **0** to **199** for **I1** and **I2**). You can use these pseudo-variables like a standard variable in any Ada expression.

This allows you to create more complex test scripts when using large arrays when the use of enumerated expressions is limited.

For multidimensional arrays, you can combine these different types of initialization and test expressions, as demonstrated in the following example:

```
ARRAY image, init = (0 => I2, 1 => ( 0 => 100, others => 0 ),
& others => (I1 + I2) % 255 )
```

Testing Character Arrays

Character arrays are a special case. Variables of this type are processed as character strings delimited by quotes.

You therefore need to initialize and test character arrays using character strings, as the following list example illustrates.

If you want to test character arrays like other arrays, you must use a format modification declaration (FORMAT instruction) to change them to arrays of integers.

Example

The following list example illustrates this type of modification:

```
TEST 2
 FAMILY nominal
 ELEMENT
  VAR l, init = NIL, ev = NONIL
  VAR s, init = "foo", ev = init
  VAR l.str(1..5), init = "foo" , ev = ('f','o','o')
  #l := stack(s, l);
 END ELEMENT
END TEST
```

Testing large arrays

The maximum number of array elements that can be processed is 100. If you need to test an array that contains more than 100 elements, then you must split the initialization of the array over two or more initializations, as shown in the following example.

Example

The following initiatialization produces a **Too many INIT or VA values** error:

```
ARRAY a, init=
(1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,20,1,2,3,4,5,6,7,8,9,30,1,2,3,
4,
5,6,7,8,9,40,1,2,3,4,5,6,7,8,9,50,1,2,3,4,5,6,7,8,9,60,1,2,3,4,5,6,7,8
,9,
70,1,2,3,4,5,6,7,8,9,80,1,2,3,4,5,6,7,8,9,90,1,2,3,4,5,6,7,8,9,100,1,2
,3,
4,5,6,7,8,9,110,1,2,3,4,5,6,7,8,9,120,1,2,3,4,5,6,7,8,9,130,1,2,3,4,5,
6,
7,8,9,140,1,2,3,4,5,6,7,8,9,150,1,2,3,4,5,6,7,8,9,160,1,2,3,4,5,6,7,8,
9,
170,1,2,3,4,5,6,7,8,9,180,1,2,3,4,5,6,7,8,9,190,1,2,3,4,5,6,7,8,9,200)
, ev=init
```

Instead, use the following expression:

```
ARRAY z [0..99],
init=(1,2,3,4,5,6,7,8,9,10,1,2,3,4,5,6,7,8,9,20,1,2,3,4,5,6,7,8,9,30,1
,2
,3,4,5,6,7,8,9,40,1,2,3,4,5,6,7,8,9,50,1,2,3,4,5,6,7,8,9,60,1,2,3,4,5,
6,
7,8,9,70,1,2,3,4,5,6,7,8,9,80,1,2,3,4,5,6,7,8,9,90,1,2,3,4,5,6,7,8,9,1
00)
, ev=init
ARRAY z [100..199],
init={1,2,3,4,5,6,7,8,9,110,1,2,3,4,5,6,7,8,9,120,1,2,3,4,5,6,7,8,9,13
0,
1,2,3,4,5,6,7,8,9,140,1,2,3,4,5,6,7,8,9,150,1,2,3,4,5,6,7,8,9,160,1,2,
3,
4,5,6,7,8,9,170,1,2,3,4,5,6,7,8,9,180,1,2,3,4,5,6,7,8,9,190,1,2,3,4,5,
6,
7,8,9,200}
, ev=init
```

Testing arrays with lists

While an expression initializes all the array elements in the same way, you can also initialize each element by using an enumerated list of expressions between brackets "**()**". In this case, you must specify a value for each array element.

Furthermore, you can precede every element in this list of initial or expected values with the array index of the element concerned followed by the characters "**=>**". The following example illustrates this form:

```
ARRAY histo[0..3], init = (0 => 0, 1 => 10, 2 => 100, 3 => 10) ...
```

This form of writing the **ARRAY** statement has several advantages:

- Improved readability of the list
- Ability to mix values without worrying about the order

You can also use this form together with the simple form if you follow this rule: once one element has been defined with its array index, you must do the same with all the following elements.

If several elements in an array are to take the same value, specify the range of elements taking this value as follows:

```
ARRAY histo[0..3], init = ( 0 .. 2 => 10, 3 => 10 ) ...
```

You can also initialize and test multidimensional arrays with a list of expressions, as follows. In this case, the previously mentioned rules apply to each dimension.

```
ARRAY image, init = (0, 1=>4, others=>(1, 2, others=>100)) ...
```

Example

You can specify a value for all the as yet undefined elements by using the keyword others, as the following example illustrates:

```
TEST 2
 FAMILY nominal
 ELEMENT
  VAR x1, init = 0, ev = init
  VAR x2, init = SIZE_IMAGE-1, ev = init
  VAR y1, init = 0, ev = init
  VAR y2, init = SIZE_IMAGE-1, ev = init
  ARRAY image, init = (others=>(others=>100)), ev = init
  ARRAY histo, init = 0,
  & ev = (100=>SIZE_IMAGE*SIZE_IMAGE, others=>0)
  VAR status, init ==, ev = 0
  #status = compute_histo(x1, y1, x2, y2, histo);
 END ELEMENT
END TEST
```

Testing arrays with other arrays

Component Testing for Ada is flexible enough to allow complex array comparisons. You can initialize or compare an array with another array that shares the same declaration.

You can use this form of initialization and testing with one or more array dimensions.

Example

The following example tests the two arrays ***read_image*** and ***extern_image***, which have been declared in the same way. Every element from the ***extern_image*** array is assigned to the corresponding ***read_image*** array element.

```
TEST 4
  FAMILY nominal
  #read_image(extern_image,"image.bmp");
  ELEMENT
    VAR x1, init = 0, ev = init
    VAR x2, init = SIZE_IMAGE-1, ev = init
    VAR y1, init = 0, ev = init
    VAR y2, init = SIZE_IMAGE-1, ev = init
    ARRAY image, init = extern_image, ev = init
    ARRAY histo, init = 0, ev ==
    VAR status, init ==, ev = 0
    #status = compute_histo(x1, y1, x2, y2, histo);
  END ELEMENT
END TEST
```

Testing records

To test all the fields of a structured variable or record, use a single **STR** instruction to define the initializations and expected values of the structure.

The **STR** statement specifies both the test start-up procedure and the post-execution test for simple variables. This instruction uses three parameters:

- **Name of the variable under test:** this can be a simple variable, an array element, or a field of a record. It is also possible to test an entire array, part of an array or all the fields of a record.
- Initial value of the variable: identified by the keyword INIT.
- Expected value of the variable after the procedure has been executed: identified by the keyword EV.

Declare variables under test with the **STR** statement, followed by the declaration keywords:

- **INIT =** for an assignment
- **INIT ==** for no initialization
- **EV =** for a simple test.

It does not matter where the **STR** instructions are located with respect to the test procedure call since the Ada code generator separates **STR** instructions into two parts :

- The variable test is initialized with the **ELEMENT** instruction
- The actual test against the expected value is done with the **END ELEMENT** instruction

Many other forms are available that enable you to create more complex test scenarios.

Example

The following example demonstrates typical use of the **STR** statement:

```
--procedure push(l: in out list; s:string);
TEST 2
  FAMILY nominal
  ELEMENT
    VAR l, init = NIL, ev = NONIL
    STR l.all, init == , ev = ("myfoo",NIL,NIL)
    VAR s, init = "myfoo", ev = init
    #push(l,s);
  END ELEMENT
END TEST
```

Testing a Record with Ada Expressions

To initialize and test a structured variable or record, you must initialize or test all the fields using a list of native language expressions (one per field). The following example illustrates this form:

```
STR l.all, init == , ev = ("myfoo",NIL,NIL)
```

Each element in the list must correspond to the structured variable field as it was declared.

Every expression in the list must obey the rules described so far, according to the type of field being initialized and tested:

- An expression for simple fields or arrays of simple variables initialized using an expression
- In Ada, an aggregate for fields of type record or array

Using Field Names in Native Expressions

As with arrays, you can specify field names in native expressions by following the field name of the structure with the characters =>, as follows:

```
TEST 3
FAMILY nominal
ELEMENT
VAR l, init = NIL, ev = NONIL
```

```
 VAR l.all, init == , ev = (str=>"myfoo",next=>NIL,prev=>NIL)
 VAR s, init = "myfoo", ev = init
 #l = push(l,s);
 END ELEMENT
 END TEST
```

When using this form, you do not have to respect the order of expressions in the list.

Testing a Record with Another Record

As with arrays, you can initialize and test a record using another record of the same type. The following example illustrates this form:

```
STR l.all, init == , ev = l1.all
```

Each field of the structured variable will be initialized or tested using the associated fields of the variable used for initialization or testing.

Testing Records with Discriminants

You can use record types with discriminants, with the following Ada restrictions:

- The initialization part must be complete.
- The evaluation can omit every field except discriminant fields.

Initialization and expected value expressions are Ada aggregates beginning with the value of the discriminant.

Example

Ada example:

```
type rec (discr:boolean:=TRUE)
 case discr is
 when TRUE =>
 ch2:float;
 when FALSE =>
 ch3:integer;
 end case;
end record;
```

Test Script Sample:

```
#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
```

Testing Tagged Records

Component Testing for Ada supports tagged record types. As with other classic records, you can omit a field in the initialization or evaluation part. You can also define tagged types with a discriminant part. In such cases, the only limitation is that of the discriminant.

Example

The following example illustrates tagged records. First, the source code:

```
Package Items Is
 Type Item Is Tagged Record
 X_Coord : Float;
 Y_Coord : Float;
 End Record;
Procedure foo_test;
End Items;
With Items; Use Items;
Package Forms Is
 Type Point Is New Item With Null Record;
 Type Circle Is New Item With Record
 Radius : Float;
 End Record;
 Type Triangle Is New Item With Record
 A,B,C : Float;
 End Record;
 Type Cylinder Is New Circle With Record
 Height : Float;
 End Record;
End Forms;
```

Following is the associated test script:

```
HEADER Items, ,
#With Items; Use Items;
#With Forms; Use Forms;
BEGIN Items
#I : Item := (1.0,0.5);
#C : Circle := (0.0,1.0,13.5);
#T : Triangle;
#P : Point;
#Cyl : Cylinder;
SERVICE Compute_Items
 SERVICE_TYPE extern
 TEST 1
  FAMILY Nominal
  ELEMENT
   Var T, Init = (0.0,1.5,4.5,5.0,6.5), Ev = (I with A=>4.0, B=>5.0,
C=>6.0)
   Var P, Init = I, Ev = (Y_coord => 1.0, X_coord => 0.0)
   Var I, Init = (0.0,1.0), Ev = Item(C)
   Var P, Init = (I with NULL record), Ev = (Y_coord => 1.0, X_coord
=> 0.0)
  End Element
 END TEST -- Test 1
 TEST 2
  FAMILY Nominal
  ELEMENT
   Var I, Init = (2.0,3.0), Ev ==
   Var T, Init = (2.0,3.0,4.0,5.0,6.0), Ev = (I with A=>4.0, B=>5.0,
C=>6.0)
   Var Cyl, Init = (2.0, 3.0, 4.0, 5.0), Ev ==
   Var I, Init ==, Ev = Item(Cyl)
  END ELEMENT
```

```
     END TEST -- Test 2
    END SERVICE -- Compute_Items
```

No Test

You can only initialize and test records with the following forms:

- INIT =
- INIT ==
- EV =
- EV ==

If a field of a structured variable needs to be initialized or tested in a different way, you can omit its initial and expected values from the global test of the structured variable, and run a separate test on this field.

The following example illustrates this:

```
TEST 4
  FAMILY nominal
  ELEMENT
    VAR l, init = NIL, ev = NONIL
    VAR l.all, init == , ev = (next=>NIL,prev=>NIL)
    VAR s, init in ("foo","bar"), ev = init
    VAR l.str, init ==, ev(s) in ("foo","bar")
    #push(l,s);
  END ELEMENT
END TEST
```

Stub simulation

Stub simulation is based on the idea that subroutines to be simulated are replaced with other subroutines generated in the test driver. These simulated subroutines are often referred to as *stubs*.

Stubs use the same interface as the simulated subroutines, only the body of the subroutine is replaced.

Stubs have the following roles:

- Check **in** and **in out** parameters against the simulated subroutine. If there is a mismatch, the values are stored.
- Assign **out** and **in out** parameters from the simulated procedure
- Return a value for a simulated function

To generate stubs, the Test Script Compiler needs to know the specification of the compilation units that are to be simulated.

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 define a stub, you must specify the following information:

- The data type in the calling function
- The method of passing the data

Example

An example project called Stub Ada is available from the Examples section of the Start page. This example demonstrates the use of stubs in Component Testing for Ada. See Example projects for more information.

Defining stubs

The following example 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.

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_INTO is ....;
   procedure PROC(x:TYPE_PARAM,y:in out TYPE_INTO,Z:out integer);
   function FUNC return TYPE_INTO;
end GEN;
```

Use the following stub definition:

```
DEFINE STUB GEN
#   procedure PROC(x: _NO TYPE_PARAM,y: _NO TYPE_INTO,Z:out integer);
#   function FUNC return _NO TYPE_INTO;
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

It 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.

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 example:

```
STUB F 1..10 => (1<->5)30,
& 11..19 => (1<->5)0,
& 20..30 => (<->) 1,
& others =>(<->)-1
```

To check that a **STUB** is never called, even if an **ENVIRONMENT** containing the **STUB** is used, use the the following syntax:

```
STUB F 0=>(<->)
```

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:

- Simulated unit (specification)
- <*testname*>_var_simule.ada
- <*testname*>_var_simule_B.ada
- Test program
- <*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.

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.

Multiple stub calls

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
```

Advanced stubs

This section covers some of the more complex notions when dealing with stub simulations in Component Testing for Ada.

Creating complex stubs

If necessary, you can make stub operation more complex by inserting native Ada code into the body of the simulated function. You can do this easily by adding the lines of native code after the prototype.

Example

The following stub definition makes extensive use of native Ada code.

```
DEFINE STUB file
 #function open_file(f:string) return file_t is
 #begin
 # raise file_error;
 #end;
END DEFINE
```

Excluding a parameter from a stub

You can specify in the stub definition that a particular parameter is not to be tested or given a value. This is done using a modifier of type **no** instead of **in**, **out** or **in out**.

> **Note** You must be careful when using **_no** on an output parameter, as no value will be assigned. It will then be difficult to predict the behavior of the function under test on returning from the stub.

Example

In this example, the *f* parameters to ***read_file*** and ***write_file*** are never tested.

```
DEFINE STUB file
```

```
#procedure read_file(f: _no file_t; l:out string; res:out BOOLEAN);
#procedure write_file(f: _no file_t, l : string);
END DEFINE
```

Stubbing separate compilation units

It is possible to create stubs for separate compilation units, such as procedures or packages, even for protected packages.

For the stubbing of a protected object to work, you must either:

- Stub the package containing the protected object, or
- A body exists for the package in which the protected body is declared as separate.

To stub a protected object you must use the following syntax:

```
DEFINE STUB SEPARATE(<package>) <compilation unit>
...
END DEFINE
```

If the compilation unit does contain an **entry** statement, the **entry** itself cannot be stubbed. In this case you must define the **entry** body within the DEFINE STUB block as in the following example:

```
DEFINE STUB SEPARATE(<package>) <compilation unit>
# entry body E1 ... is ...
END DEFINE
```

Example

The following example is a **.ptu** test script implementing a stub of a separate compilation unit. It is available in the **StubAda** example project provided with the product.

```
HEADER PARENT, ,
#With PARENT;
BEGIN
  DEFINE STUB package
  END DEFINE
  DEFINE STUB SEPARATE(package) MY_VALUE
  END DEFINE
  SERVICE SOMETHING
  SERVICE_TYPE extern
    -- Declaration of service's parameters
    #X : INTEGER;
    #Ret : INTEGER;
    TEST 1
    FAMILY nominal
      ELEMENT
   -- stub of the protected object "get"
        STUB My_Value.Get()2
        Var X, Init = 0, ev = Init
        Var Ret, Init = 0, ev = 2
        #Ret := PARENT.SOMETHING(X);
      END ELEMENT
    END TEST -- TEST 1
  END SERVICE -- SOMETHING
```

Stubbing generic units

You can stub generic units just as ordinary units by using the following syntax:

```
DEFINE STUB STUB_NAME < dimension>
```

```
# optional declarations
END DEFINE
```

The Unit Testing Ada Test Compiler generates a stub body for this unit to perform the desired simulations.

Simulating functions with _inout mode arrays

To stub a function that takes 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>})
```

Stubbing functions with varying parameters

In some cases, functions may be designed to 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
```

Stubbing a body separately

Under certain circumstances, it may be useful to define the body stub separately, with a proprietary behavior.

To do this, declare the stub separately and then define a body for it.

Example

In the following example, Component Testing for Ada will not generate a body for the service My_Procedure, but will expect you to do so:

```
DEFINE STUB <STUB NAME>
# procedure My_Procedure(...) is separate ;
END DEFINE
```

Advanced Ada testing

Advanced Ada testing

This section covers some of the more complex notions behind Component Testing for Ada.

Testing Internal Procedures and Internal and Private Variables

Black box testing is not sufficient as soon as you want to test the following:

- Internal procedures of packages
- Internal variables of packages
- Private type variables

For packages, you can test internal procedures via external procedures. However, it is sometimes easier to test them directly.

You cannot modify or test internal variables with a black box approach. Internal variables are generally tested via external procedures, but it is sometimes easier to modify and test them directly also.

Private type variables are also a problem because their structure is not visible from outside the package.

Testing Generic Compilation Units

Types and objects in a generic unit depend on generic formal parameters that are not known by the Test Script Compiler. Therefore, Component Testing for Ada cannot directly test a generic package.

To test a generic package, you must first instanciate the package and then call the instance. Such instances must appear in compilation units or at the beginning of the test script (in any case before the BEGIN statement), as follows:

```
WITH <generic>;
PACKAGE <instance> IS NEW <generic> (...);
```

Depending on the nature of the source code under test, there are two ways to test an instanciation of a generic package:

- If the code cannot contain a specific procedure for testing purposes and the test does not need access to internal variables, then the test body can be generated as an external package. The test body can view the instance under test through the use of a **WITH** instruction.

  In the **.ptu** test script, after the generic instanciation, add the **WITH *<instance>*;** statement before the **BEGIN** keyword. For example:

```
      WITH <Generic_Package>;
      PACKAGE <Instance> IS NEW <Generic_Package> (...);
      WITH <Instance>;
      BEGIN
```

  where <*Generic_Package*> is the name of the generic unit under test, and <*Instance*> is the name of the instanciated unit from the generic.

- If you need to test private types within the generic package and the test needs access to all internal variables, then the test body must be part of the generic package as a specific test procedure.

  In the **.ptu** test script, specify the generic package, the instance package and the test procedure on the **BEGIN** line. For example:

```
      WITH <Generic_Package>;
      PACKAGE <Instance> IS NEW <Generic_Package> (...);
      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 instanciated unit from the generic. The <*Procedure_Name*> parameter is not mandatory. Component Testing uses **Attol_Test** by default.

  This instruction generates the test body into <*Procedure_Name*> as a separate unit of the Generic package as well as the **WITH** to this instance, as requested by the test body.

  If specified, <*Procedure_Name*> must be part of the generic package as separate procedure.

Example

Consider the following Ada compilation unit:

```
Generic
Type t is private ;
Procedure swap(x,y :in out t) ;
Procedure swap(x,y :in out t) is
Z :t ;
Begin
  Z := x ;
  X := y;
  Y := z;
End swap ;
With swap ;
Procedure swap_integer is new swap(integer) ;
```

You can test the ***swap_integer*** procedure just like any other procedure:

```
HEADER swap_integer,,
#with swap_integer;
BEGIN
SERVICE swap_integer
  #x,y:integer;
  TEST 1
    FAMILY nominal
```

```
        ELEMENT
          VAR x , init = 1 , ev = 4
          Var Y , init=4 ,ev = 1
          #swap_integer(x,y) ;
        END ELEMENT
      END TEST
    END SERVICE
```

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 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 to the string "l11c01pA00",
--and then string comparisons are done with the expected values using
str_comp.
```

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_Under_Tst, 0.3, 0.0
#with Pck_Stub;
BEGIN Prg_Under_Tst
DEFINE STUB Pck_Stub
#with Text_IO;
#procedure Proc_Stubbed is
#begin
```

```
# Text_IO.Put_Line("Stub called.");
#end;
END DEFINE
SERVICE S1
SERVICE_TYPE extern
    #Param_1 : duration;
    #task1 : Prioritaire;
    TEST 1
    FAMILY nominal
        ELEMENT
       VAR   Param_1, init = duration(0), ev = init
   STUB  Pck_Stub.Proc_Stubbed 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_1, init = duration(2), ev = init
   STUB  Pck_Stub.Proc_Stubbed 1..1 => ()
    #declare
          #   Task2 : T_Prio := new Prioritaire;
    #begin
    #  Task2.Do_Something_Useful(Param_1);
    #  Task2.Unit_Testing_Exit_Loop;
    #  Task2.Unit_Testing_Wait_Termination;
    #end;
        END ELEMENT
    END TEST -- TEST 2
END SERVICE --S1
```

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.

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.

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 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.

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.

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 Iterations

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.

Test cases introduced by the instruction **NEXT_TEST** can be dissociated after the tests are run. With the ELEMENT structure, the different phases of the same test case can be dissociated.

Test phases introduced by the instruction **ELEMENT** can be included in the loops created by the **LOOP** instruction**.**

Viewing reports

After test execution, depending on the options selected, a series of Component Testing for Ada test reports are produced.

Understanding Component Testing reports

Test reports for Component Testing are displayed in the 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.

Comparing Ada Test Reports

The Component Testing comparison capability allows you to compare the results of the last two consecutive tests.

To activate the comparison mode, select **Compare two test runs** in the Component Testing for C and Ada Settings 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.

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 the speed of execution because checks are performed on each array as a whole.

Component Testing for Java

The Component Testing for Java feature of Test RealTime provides a unique, fully automated, and proven solution for Java unit 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.

How Component Testing for Java works

When a test node is executed, Test RealTime compiles both the **.java** test scripts and the source under test. This preprocessing creates a **.tsf** file. The resulting source code generates a test driver.

If any Runtime Analysis tools are associated with the test node, then the source code is also instrumented with the Java Instrumentor (**javi**) tool.

The test driver, TDP, stubs and dependency files all make up the test harness.

The test harness interacts with the source code under test and produces test results. Test execution creates a **.tdf** file.

The **.tsf** and **.tdf** files are processed together the Component Testing Report Generator (**javapostpro**). the output is the **.xrd** report file, which can be viewed and controlled in the Test RealTime GUI.

Of course, these steps are mostly transparent to the user when the test node is executed in the Test RealTime GUI.

Java testing overview

About JUnit

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

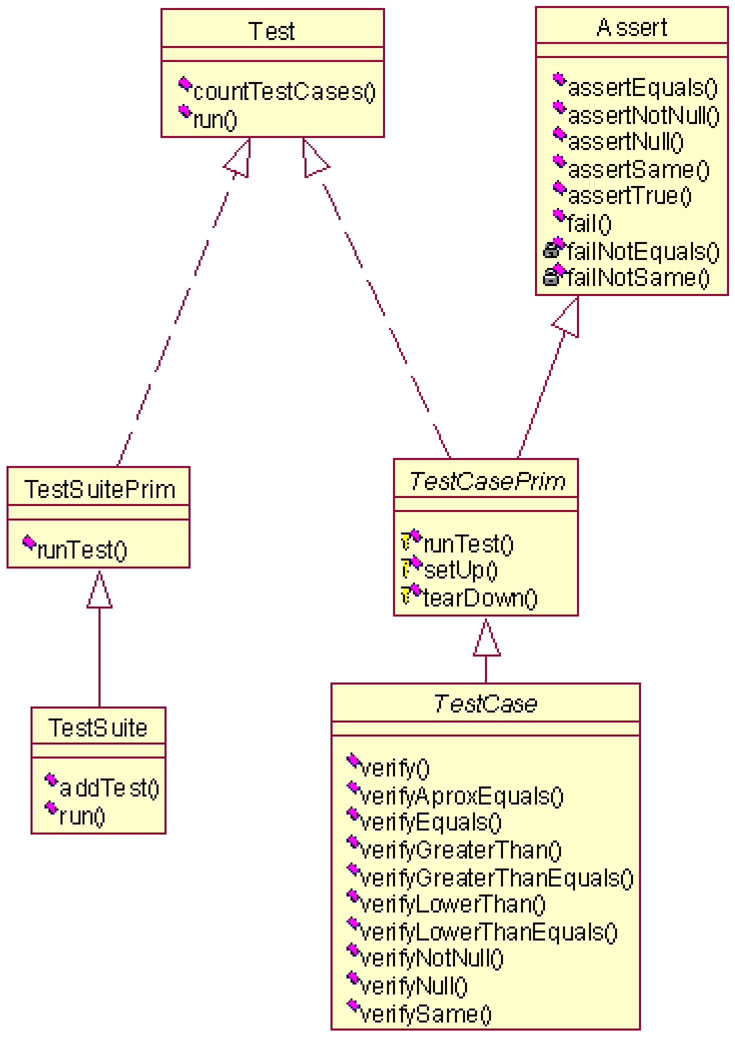
JUnit for J2ME

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 Reference section

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 Reference section.

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 ***TestCase***s 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= m12EUR.add(m14EUR);
    assertTrue(expected.equals(result));
}
```

Java stub harness

Component Testing for Java supports the following verification methods for stubbed classes:

- Stub failure detection
- Stub sequence mechanism

Stub mechanism overview

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.

For each test, the ***enter***, ***exit*** and ***fail*** methods of the stubbed objects are logged by the object **TestSynchroStub**.

You can then query the **TestSynchroStub** object to:

- Compare the actual stub ***enter*** and ***exit*** sequence against an expected sequence as defined by the **StubSequence** object
- Check for the presence of at least one ***fail*** method

Use the test harness ***assert*** or ***verify*** primitives to add the stub sequence or failure verification results into a formal test report.

Stub sequence verification

Use the **StubSequence** object to test the stub sequence. This object can be loaded with the sequence under test. The actual comparison is performed with the corresponding method of the object **TestSynchroStub**.

The following example demonstrates how to verify the entry into a method

```
public void teststub3()
{
  NoStub another = new NoStub();
  another.call1();
  StubSequence testof = new StubSequence(this);
  verifyLogMessage("verify one enter method");
  testof.addEltToSequence( new StubbedOne().getClass() , "methodone",
StubInfo.ENTER) ;
  verifyEquals("Test single
sequence",TestSynchroStub.isSeqRespected(testof),true);
}
```

This example shows how to verify the entry into ***methodone*** of the class ***StubbedOne*** and that the method m1 of StubbedTwo has been successively entered and exited. This is part of the call stack of the methods of stubbed objects.

```
public void teststub4()
{
  NoStub another = new NoStub();
  another.call1();
  StubSequence testof = new StubSequence(this);
  testof = new StubSequence(this);
  testof.addEltToSequence( new StubbedOne().getClass() ,
"methodone",StubInfo.ENTER) ;
  testof.addEltToSequence( new StubbedTwo().getClass() ,
"m1",StubInfo.ENTER) ;
  testof.addEltToSequence( new StubbedTwo().getClass() ,
"m1",StubInfo.EXIT) ;
  verifyLogMessage("Check true for stub calls");
  verifyEquals("Test single
sequence",TestSynchroStub.isSeqRespected(testof),true);
}
```

Stub failure detection

In the Component Testing for Java test harness, stubs can declare an error by use of the ***fail()*** method.

To check for the existence of a stub error, use the following global call type:

```
  verify("Test single
sequence",TestSynchroStub.areStubfail(this),true);
```

The following example demonstrates the use of **TestSynchroStub** to test if a stub has been declared failed.

```
public void testStubFail()
{
  StubbedThree st = new StubbedThree();
  st.call();
  verifyLogMessage("Check fail call from stub");
  verifyEquals("Test single
sequence",TestSynchroStub.areStubfail(this),true);
}
```

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

- Create a derived class of **TestCase**:
- 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.

- 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 thow 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

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.

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:

9. Create a test suite class **Suite()** that transforms a test class into a J2ME test suite.
10. 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* ✗ 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 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:

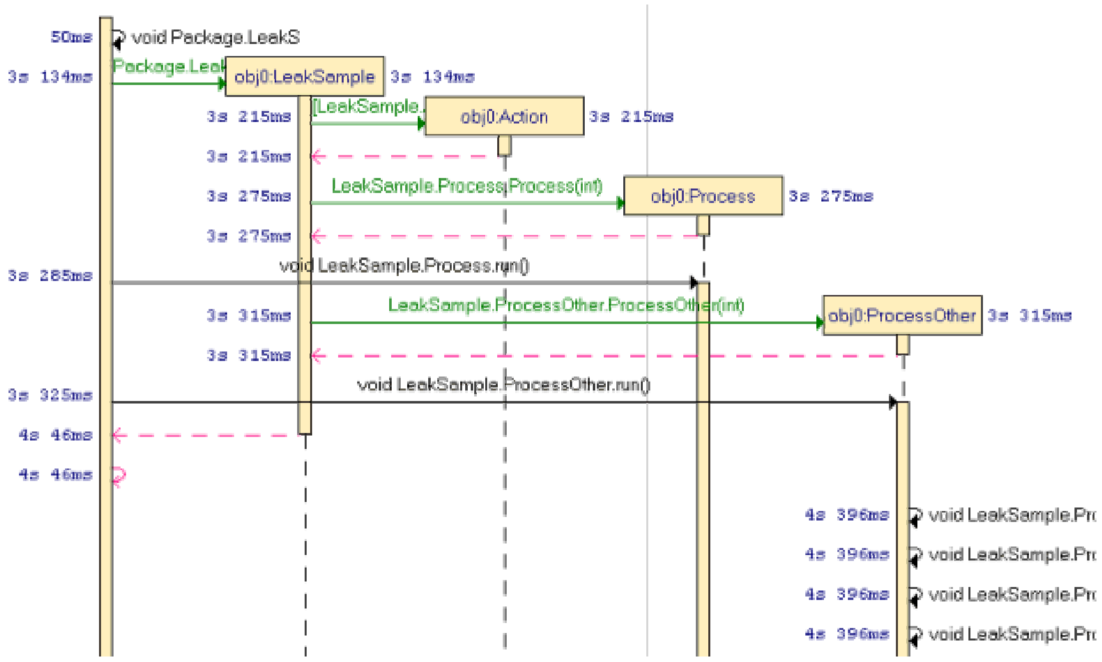
Condition	Result	Status
A verification fails	✗	Failed
All verifications pass on each execution	✓	Passed

Understanding Java component testing UML sequence siagrams

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

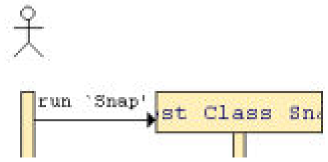
Component Testing for Java generates **UNEXPECTED EXCEPTION** Notes whenever an unexpected exception is encountered.

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

System Testing for C 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 for C 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 the System Testing tool, 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 for C is recommended for testing:

- Telecommunication and networking equipment using standard protocols
- Aerospace equipment using standard or proprietary operating systems and a communication bus
- Automotive Electronic Control Units (ECUs) or appliance systems
- Distributed applications based on message-oriented middleware
- Applications developed using Rational Rose RealTime

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 on the Test RealTime host computer, 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**

On Windows platforms, the **ATS_DIR** environment variable must be set to indicate the directory where the **atsagtd.exe** and **atsagtd.ini** files are located. If the file cannot be found, only the current user on the current computer will be authorized.

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**.

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.

In some cases, you might need to update the file **atsagtd.sh** shell script to add some environment variables to the target machine.

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. The System Testing Agent should answer with the following message:

```
210 hello, please to meet you.
```

After the connection succeeds, press **Enter** to close the connection or type the following command to check that <*username*> is set up as a user:

```
Jef <username>
```

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.

To see the current working directory, type the following command:

```
PWD
```

To close the connection, type:

```
QUIT
```

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:

```
<computer name> <username> [#<comment>]
```

On Windows platforms, the System Testing Agent uses the **ATS_DIR** environment variable to locate the **atsagt.ini** file.

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. Blank lines are not allowed.

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**. The name of the virtual tester must be a standard C identifier.
- **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. Make sure that **Execution** is selected in your Build options.
2. In the **Project Explorer**, right-click a System Testing node.
3. From the pop-up menu, select **Deployment Configuration**.
4. 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. Leave this field blank if you want to use the IP address specified in the Host Configuration section of the General Settings.

  Note If the IP address line in the Host Configuration settings is blank, then the Virtual Tester Deployment Table retrieves the IP address of the local machine when generating the deployment script.

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.

- **RIO filename:** This specifies the name of the .rio file containing the Virtual Tester output, for use in multi-threaded or RTOS environments.

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.

For the **.spv** supervisor script to be generated, the **Execution** option must be selected in the Build options.

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. Make sure that **Execution** is selected in your Build options.
2. In the **Project Explorer**, right-click a System Testing node.
3. From the pop-up menu, select **Generate Deployment Script**.
4. 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 Reference section.

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 t**est 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.
5. 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.
6. 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.

Launching virtual tester threads

In a multithreaded environment, there are two methods of starting the virtual tester threads:

- From a specially designed thread launcher program that you must write to include in your project.
- From a TDP thread launcher if available.

TDP thread launcher from TDP

Some TDPs can launch the virtual tester threads without needing a special program. If your TDP supports this method, the only requirement is to specify this in the Configuration settings of the System Testing test node.

To use the TDP thread launcher:

1. In the **Project Explorer**, click the **Settings** button.
2. Select a System Testing test node in the **Project Explorer** pane.
3. In the Configuration Settings list, expand **System Testing** and select **Test Compiler**.
4. Set the **Use thread launcher from TDP** setting to **Yes**.
5. When you have finished, click **OK** to validate the changes.

TDP thread launcher program

If the TDP does not contain a TDP thread launcher, the only way to start 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

The following example is a sample program for launching virtual tester threads.

```
#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);
}
```

An example demonstrating how to use System Testing for C on multithreaded applications is provided in the **Broadcast Server** example project. See Example projects for more information.

System Testing for C Test Scripts

Flow control

Several execution flow instructions let you develop algorithms with multiple branches.

System Testing **.pts** test script flow control instructions include:

- Function calls
- Conditions
- Iterations
- Multiple Conditions

Function calls

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.
```

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
```

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
```

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
```

Adaptation layer

The adaptation Layer 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 **COMMTYPE** instruction cannot handle basic types. Therefore, you must previously define the type with a **typedef** statement.

For example, on UNIX systems, the data type for the BSD sockets is an integer. The **COMMTYPE** instruction is therefore used as follows:

```
#typedef int bsd_socket_id_t;
COMMTYPE ux_bsd_socket IS bsd_socket_id_t
```

In the ***stack*** example provided with the product, the following line 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*** example 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)
```

In this example, the function call to **mbx_init** opens a connection between the Virtual Tester and the system under test. 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 by the **SEND** instruction to sends a message to the system under test (SUT).

In the following example, **msg** is a **message_t** typed input formal parameter specifying the message to send. The input formal parameter id is used to know where to send the 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.

The script must have one **PROCSEND** for each message type and channel type pair used by any of the **SEND** instructions.

The name of each **PROCSEND** in the generated C code is made up with the signature of the *message* type and *channel* type for each **PROCSEND** found in the test script, as follows.

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.

To send the message on the appropriate channel, the generated code calls the **PROCSEND** named with the signature of the *message* type to be sent (first parameter) and the *channel* type to be used (second parameter).

The message type is provided by the **MESSAGE** instruction. The channel type is provided by the **CHANNEL** instruction.

Therefore, in the generated code, the **SEND** instruction calls the following function:

```
PROCSEND_message_t_appl_comm(message, appl_ch)
```

which corresponds to the following line in the test script:

```
PROCSEND message_t ... ON appl_comm
```

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;
PROCSEND message_t: msg ON appl_comm: id
   CALL mbx_send_message ( &id, &msg) @ err_ok
END PROCSEND
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 procedre 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 system under test could be:

```
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**.

When multiple connections are used, 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.

The script must have one CALLBACK for each *message* type - *channel* type pair used by any WAITTIL instructions.

The name of each **CALLBACK** in the generated C code is made up with the signature of the *message* type and *channel* type for each **CALLBACK** found in the test script.

DEF_MESSAGE Instruction

The **DEF_MESSAGE** instruction defines the values of a reference message declared with the **MESSAGE** instruction. A reference message is a set of field values as expected by the virtual tester from the system under test. Any undefined fields are not compared to the receive message.

```
DEF_MESSAGE ack, EV= { type => ACK }
DEF_MESSAGE data, EV= {
& type => DATA,
& applname => "SATURN",
& userdata => "hello world !" }
```

To learn more about the **DEF_MESSAGE** Instruction, see the Messages and Data Management chapter.

WAITTIL Instruction

The **WAITTIL** instruction allows the test script to wait for events or conditions. **WAITTIL** is made of two Boolean expressions: an expected condition, and a failure condition. The script execution pauses 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 3000ms (300 x 10ms, the default time unit), the **WAITTIL** exits with an error. The scenario execution is interrupted.

```
WAITTIL ( MATCHING(ack, appl_ch), WTIME == 300)
```

The time unit is configurable in the Target Deployment Port depending on the target platform.

To receive a message on the appropriate channel, the generated code calls a CALLBACK named with the signature of the expected message type (first parameter) and the channel type (second parameter).

The message type is provided by the MESSAGE instruction. The channel type is provided by the CHANNEL instruction.

Therefore, in the generated code, the **SEND** instruction calls the following function:

```
CALLBACK_message_t_appl_comm(message, appl_ch)
```

which corresponds to the following line in the test script:

```
CALLBACK message_t ... ON appl_comm
```

If the channel parameter is omitted in the **WAITTIL** instruction, the generated code calls all CALLBACK instructions that read the corresponding message type on all known channel types.

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

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 = <expec_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 *<expec_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 (**=>**) 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 (**=>**) 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 (**..**).

```
#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. Virtual Tester messages can be either an *integer* or a *text string*.

For information about the **INTERSEND** and **INTERRECV** statements, please refer to the System Testing Script Language section in the **Rational Test RealTime Reference Manual**.

For these statements to be active, you must enable On-the-fly Runtime Tracing in the Configuration Settings.

To enable Virtual Tester communication:

1. In the Project Explorer, select the System Testing test node, and click **Settings**.
2. In the **Configuration Settings** dialog box, select **System Testing** and **Target Deployment Port for System Testing**.
3. Set **Enable On-the-fly Runtime Tracing** to **Yes** and click **OK**.

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.

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.

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.

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
```

Using native C statements

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 declare native variables that participate in the flow of a scenario. Such statements must be analyzed by the System Testing Parser.
- To insert native code into a scenario. In this case, the code is ignored by the parser, but carried on into the generated code.

Analyzed native code

Lines prefixed with a # character are analyzed by Component Testing Parser.

Only prefix statements with a # character to include native C variable declarations that must be analyzed by the parser.

```
#int i;
#char *foo;
```

Variable declarations must be placed outside of System Testing Language blocks or preferably at the beginning of scenarios and procedures.

Ignored native code

Lines prefixed with a @ character are ignored by the parser, but copied into the generated code.

To use native C code in the test script, start instructions with a @ character:

```
@for(i=0; i++; i<100) func(i);
@foo(a,&b,c);
```

You can add native code either inside or outside of System Testing Language blocks.

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.

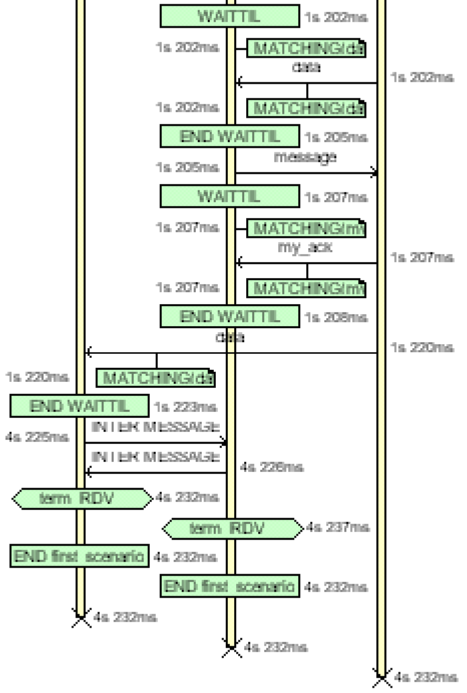
- Elements marked as *Failed* ✗ are either a failed test, or an element that contains at least one failed test.
- Elements marked as *Passed* ✓ are either passed tests or elements that contain only passed tests.

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 Repo

sts the details of test script execution, with time

values for each field

t compares initial values, expected values and

Understan ***e Diagrams***

g generates trace data this is used by the UML/SD

. uses standard UML notation to represent both

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

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.

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 Reference section for detailed information on **.pts** test script instructions.

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 Reference section for detailed information on **.pts** test script instructions.

Note The **TRACE_ON**, **TRACE_OFF** and **FLUSH_TRACE** instructions only apply when the Circular Trace Buffer option is selected.

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:

1. select **On-the-fly tracing** in the System Testing Advanced Settings for the System Testing test node
2. ensure that the **Allow remote connections** option in selected in the General Preferences.

Chapter 5. Using the graphical user interface

The graphical user interface (GUI) of Test RealTime provides an integrated 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 source code, test, analysis reports, UML sequence diagrams. 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 an 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.

The Test RealTime 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.

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 **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.

   Note The Enter and Esc keys do not work in the Configuration Settings window. Use the **OK**, **Apply**, and **Cancel** buttons.

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.

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:

1. 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 create a new Configuration for a Project:

1. From the **Project** menu, select **Configurations**.
2. In the **Configurations** dialog box, click the **New...** button.
3. Enter a **Name** for the Configuration.
4. Select the **Target Deployment Port** to be used to create the Configuration.
5. Enter the **Hostname**, **Address** and **Port** of the machine on which the Target Deployment Port is to be compiled.
6. Click **OK**.
7. 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. From the **Project** menu, select **Configurations**.
2. In the **Configurations** dialog box, select the Configuration to be removed.
3. Click the **Remove** button.
4. 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. From the **Project** menu, select **Configurations**.

2. In the **Configurations** dialog box, select an existing Configuration.
3. Click the **Copy To...** button
4. Enter a **Name** for the new Configuration.
5. Click **OK**.

Creating tests and applications

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.

Creating a new project

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.

Creating a runtime analysis application node

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.

Once you have created your application node, you simply add the options required to run any of the 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. The **Application Files** page opens. Use the **Add** and **Remove** buttons to build a list of source files and header files (for C and C++) to add to your project.

   The **Configuration Settings** button allows you to override the default Configuration Settings. 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.
7. 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.
8. 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 tools to the application under analysis.

Creating a component test

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 unit 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 .

There are two methods of creating a test node with the Component Testing wizard:

- **From the Start page:** this method allows you to specify a set of files or components to test.
- **From the Asset Browser:** this method rapidly creates a test from a single file or source code component selected in the Asset Browser.

Once the test node has been generated, you can complete your Component Testing test scripts in the Text Editor. Refer to the **Test RealTime Reference Manual** for information about the actual language semantics.

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. 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.
   On the **Start Page**, select **Activities** and choose the **Component Testing** activity.

2. The **Application Files** page opens. Use the Add and Remove buttons to build a list of source files and header files (for C and C++) to add to your project.

   The Configuration Settings button allows you to override the default Configuration Settings. Select **Compute Static Metrics** to run the analysis of static testability metrics.

   Click **Next>** to continue.

Note If the static metrics analysis takes too much time, you can clear the **Compute Static Metrics** option. In this case, the calculation and display of static metrics in any further steps are disabled.

Note With Component Testing for Ada, it is not possible to submit only an Ada procedure file. Instead, you must include the single procedure in a package.

3. The **Components Under Test** page allows you to select the units or files for the selected source files.

   In order to help you choose which components you want to test, this page displays the metrics for each file or unit (packages, classes or functions depending on the language).
   Select **File Selection** to choose files under test or **Unit Selection** to choose the source code units that require testing. The selection mode toggles the static metrics displayed between file metrics or unit metrics.

Note If the Unit Selection view seems incomplete, cancel the wizard, from the **Project** menu, select **Refresh File Information** and restart the wizard. See Refreshing the Asset Browser.

   Click **Metrics Diagram** to select the units under test from a graph representation.
   Click **Next>** to continue or **Generate** to skip any further configuration and to use default settings.

4. The **Test Script Generation Settings** page allows you to specify the test node generation options. The **General** settings specify how the wizard creates the test node.

   - **Test Name:** Enter a name for the test node.
   - **Test Mode:** Disables or enables the test boundaries.
   - **Typical Mode:** No test boundaries are specified. This is the default setting. For Java, all dependency classes are stubbed in the node.

- **Expert Mode:** This mode allowing you to manually drive generation of the test harness. This provides more flexibility in sophisticated software architectures.
- **Node Creation Mode:** Selects how the test node is created:
- **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 components.
- **Multiple Mode:** This creates a single test node for each selected source code unit.

The **Components Under Test** settings specify advanced settings for each component of the test node. These settings depend on the language and Configuration.
Click **Next>** to continue.

5. 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.

6. The **Test Generation Result** page displays progression of the test node creation process. Click **Settings** to set the Configuration Settings. You can always modify the test node Configuration Settings later if necessary, from the Project Explorer.

   Note If you apply new settings after the test generation, the wizard reruns the test generation. This allows you to fine-tune any settings that may cause the test generation to fail.

Once a test node has been successfully generated, click **Finish** to quit the Component Testing Wizard and update the project.

To run the Component Testing Wizard from the Asset Browser:

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 **Asset Browser** tab.
4. Right click an object, package or source file under test. From the popup menu, select **Test**.
5. The **Test Script Generation Settings** allows you to specify the test node generation options. The **General** settings specify how the wizard creates the test node.
   - **Test Name:** Enter a name for the test node.
   - **Test Mode:** Disables or enables the test boundaries.
   - **Typical Mode:** No test boundaries are specified. This is the default setting. For Java, all dependency classes are stubbed in the node.
   - **Expert Mode:** This mode allowing you to manually drive generation of the test harness. This provides more flexibility in sophisticated software architectures.

   The **Components Under Test** settings specify advanced settings for the component of the test node. These settings depend on the language and Configuration.

   Click **Next>** to continue.
6. 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.
7. The **Test Generation Result** page displays progression of the test node creation process. Click **Settings** to set the Configuration Settings. You can always modify the test node Configuration Settings later if necessary, from the Project Explorer.

Note If you apply new settings after the test generation, the wizard reruns the test generation. This allows you to fine-tune any settings that may cause the test generation to fail.

Once a test node has been successfully generated, click **Finish** to quit the Component Testing Wizard and update the project.

Creating a system test

The System Testing Wizard helps you create a new System Testing test node in your project.

Basically, a System Testing node contains a **.pts** test script as well as a set of Virtual Testers for message-based testing.

Note System Testing for C does not support paths or filenames which contain spaces. When naming files or directories, make sure that these do not contain any spaces.

To create a System Testing node:

1. Enter the name of the new System Testing test node.
2. On the **Test Script Selection (1/7)** page, you select the source files that are used to build your application among the source files that are currently in your workspace.

   Select whether you want to create a new **.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.

   Next, use the **Add** and **Remove** buttons to build a list of interface files. The **Interface Files List** must contain **.h** header files that define the message structures used by your application.

   Click **Next** to continue.
3. On the **Include Directories List (2/7)** page, specify the directories that contain include files that can be required by the interface files and the messaging API.

   Use the **Add** and **Remove** buttons to build a list of include directories. These are the directories that contain files that are included by your application's source code. If necessary, you can use the **Up** and **Down** buttons to indicate the order in which they are searched.

   Click Next to continue. If you chose to use an existing **.pts** test script, this brings you straight **Virtual Tester Driver Creation (5/7)** page to step 6.
4. On the **Generate New Test Script (3/7)** page, specify the message type to be used by the test.
   - **Message type:** Select the type definition that will be used for messages.
   - **Base filename:** Specify the name of the generated API files. The wizard generates **.c**, **.h** and **.hts** files based on this filename.
   - **Directory:** Specify the location where the API files will be generated.
   - **Generate with INSTANCE blocks:** Select this option if you want INSTANCE statements to be created in the **.pts** test for a multi-process or multi-threaded test driver.

   Click **Next** to continue.
5. On the **Generate New Test Script (4/7)** page, change the configuration settings of the test node or click **Next**.
6. On the **Virtual Tester Driver Creation (5/7)** page, you can create a set of virtual testers. Use the **New** button to create and name a new virtual tester. You can create and duplicate several virtual testers. You can also skip this page and decide to create your virtual testers later on.

   When a virtual tester is selected, the **General** tab allows you to specify an instance and target deployment port for the virtual tester.

    - **VT Name:** This is the name of the selected virtual tester. This must be a standard C identifier.
    - **Implemented INSTANCE:** Use this box to assign an **INSTANCE** statement, defined in the **.pts** test script, to the selected virtual tester. This information is used to deploy the virtual tester. Select **Default** to manually specify the instance during deployment.
    - **Target:** Select the Target Deployment Port that will be used for the selected Virtual Tester.

   The **Scenario** tab lets you select one or several scenarios as defined in the **.pts** test script. During execution, the Virtual Tester plays the selected scenarios.

   The **Family** tab lets you select one or several families as defined in the **.pts** test script. During execution, the Virtual Tester plays the selected families.

   If necessary, click the **Configure Settings** button to change the configuration settings for the selected virtual tester.

   The **API source files** list displays the generated messaging API source files. Use the **Add** or **Remove** buttons to modify this list if your messaging API requires more files.

   Click **Next** to continue.

7. On the **Deploy Configuration (6/7)** page, you specify how to deploy the virtual testers onto host and target computers. Use the **Add**, **Remove** buttons to modify the list. Each line represents one or several parallel 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 testers. If an instance was specifically assigned in the Virtual Tester Configuration box, this cannot be changed. Select **<default>** 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. Leave this field blank if you want to use the IP address specified in the Host Configuration section of the General Settings.

     Note If the IP address line in the Host Configuration settings is blank, then the Virtual Tester Deployment Table retrieves the IP address of the local machine when generating the deployment script.

     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.
   - **RIO filename:** This specifies the name of the .rio file containing the Virtual Tester output, for use in multi-threaded or RTOS environments.

     If necessary, click the **Rendezvous Configuration** button to set up any rendezvous members.

   Click **Next** to continue.

8. Review the options in the **Test Generation Summary (7/7)** 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 testers that are to be deployed by the test.
9. 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 complete the generated System Testing test script in the Text Editor and then configure and deploy your virtual testers.

Refer to the System Testing language reference for information about the System Testing script language.

Viewing a static 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.

Specifying advanced component test 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

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 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 script:** 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 files in the list. The include file list used by the Component Testing wizard is 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.
- **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.

Working with projects

The project is your main work area in Test RealTime , as displayed in the **Project Explorer** window.

A project is a tree representation that contains nodes. Projects can contain one or more sub-projects which are actually links to other projects.

> **Note** Previous versions of the product used Workspaces instead of sub-projects. Workspaces are automatically converted to sub-projects when loaded into the current version of the product.

Within the project tree, each node has its own individual Configuration Settings —inherited from its parent node— and can be individually executed.

Project overview

A project is a tree representation that contains nodes.

Within the project tree, each node has its own individual Configuration Settings —inherited from its parent node— and can be individually executed.

Project Nodes

The project is your main work area in Test RealTime .

A project is materialized as a directory in your file system, which contains everything you need to test and analyze your code:

- Source code
- Test scripts
- Analysis and test result files

In the Test RealTime GUI, a project is organized as follows

- **Project node:** this node contains any of the following nodes:
- **Group node:** Allows you to group together several application or test nodes.
- **Application node:** contains a complete application.
- **Results node:** contains your runtime analysis result files, once the application has been executed. Use this node to control the result files in Rational ClearCase or any other configuration management system.
- **Source node:** these are the actual source files under test. They can be instrumented or not instrumented .

- **Test node:** represents a complete test harness, for Component Testing for C and Ada , C++ , Java or System Testing . A test node containing.
- **Results node:** contains your test result files, once the test has been executed. Use this node to control the result files in Rational ClearCase or any other configuration management system.
- **Test Script node:** contains the test driver script for the current test.
- **Source node:** these are the actual source files under test. They can be instrumented or not instrumented .
- **External Command node:** this node allows you to execute a command line anywhere in the project. Use this to launch applications or to communicate with the application under test.

Application and test nodes can be moved around the project to change the order in which they are executed. The order of files inside a Test node cannot be changed; for example the test script must be executed before the source under test.

Projects and sub-projects

Projects can contain one or more sub-projects which are actually links to other project directories. The behavior of a sub-project is the same as a project. In fact, a sub-project can be opened separately as a stand-alone project.

> **Note** Previous versions of the product used Workspaces instead of sub-projects. Workspaces are automatically converted to sub-projects when loaded into the current version of the product.

Here are several examples of the use of super-projects and sub-projects:

- In a team, users work on their own projects to develop and test portions of a larger development project. For testing the whole project, a single master project can be created to integrate, build, and test multiple sub-projects in one go.
- A single project may contain different sub-projects for different target platforms.

Results Node

By default, each application and test node contains a Results node.

Once the test or runtime analysis results have been generated, this node contains the report files. Right-click the result node or the report files to bring up the **Source Control** popup menu.

If you are not controlling result files in a configuration management system, you can hide the Results node by setting the appropriate option in the Project Preferences.

Troubleshooting a project

When executing a node for the first time in Test RealTime, it is not uncommon to experience compilation issues. Most common problems are due to some common oversights pertaining to library or include paths or Target Deployment Port settings.

To help debug such problems during execution, you can prompt the GUI to report more detailed information in the Output window by selecting the verbose output option.

To set the verbose output option from the GUI:

1. From the **Edit** menu, select **Preferences**.
2. Select the **Project** preferences.
3. Select **Verbose output** and click **OK**.

To set the verbose output option from the command line:

1. Set the environment variable **$ATTOLSTUDIO_VERBOSE**.
2. Rerun the command line tools.

Refreshing the asset browser

The **Asset Browser** view of the **Project Explorer** window analyzes source files and extracts information about source code components (classes, methods, functions, etc...) as well as any dependency files. This capability, known as *file tagging*, allows you to navigate through your source files more easily and provides direct access to the source code components through the Text Editor.

When the automatic file tagging option is selected, Test RealTime 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.

You can change the way files are tagged by Test RealTime by changing the **Source File Information** Configuration Settings for the current 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:

1. From the **Project** menu, select **Refresh File Information**.

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**.

To edit the Source File Information settings for the project:

1. In the **Project Explorer**, click the **Settings** button.
2. Select the project node in the **Project Explorer** pane.
3. In the Configuration Settings list, expand **General**.
4. Select **Source File Information**.
5. When you have finished, click **OK** to validate the changes.

Manually creating an application or test node

Application nodes and test nodes 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 filed requires for the test.

The preferred method to create an application or test 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.

The GUI allows you to freely create and modify test or application nodes. However, you must follow the logical rules regarding the order of execution of the items contained in the node. When using Component Testing for C++, **.otc** scripts must be placed before **.otd** scripts.

To manually add components to the application node.

1. In the Project Explorer, right-click a Project 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 **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.

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**.

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.

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.

Note Some reports require opening several files. For example, 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.

To open a report from the Project Explorer:

1. Execute your test with the **Build** command.
2. Right-click the application or test node.
3. From the pop-up menu, select **View Report** and then the appropriate report.

   Note Reports cannot be viewed before the application or test has been executed.

To manually open a report made of several files:

1. From the **File** menu, select **Browse Reports**. Use the **Browse Reports** window to create a list of files to be opened in a single report. For example, a **.tdf** dynamic trace file with the corresponding **.tsf** static trace files.
2. Click the **Add** button. In the **Type** box of the File Selector, select the appropriate file type. For example, select **.tdf**.
3. Locate and select the report files that you want to open. Click **Open**.
4. Click the **Add** button. In the **Type** box of the File Selector, select the appropriate file type. For example, select **.tsf**.
5. Locate and select the report files that you want to open. Click **Open**.
6. In the **Browse Reports** window, click **Open**.

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 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 and C++ and Performance Profiling results.

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 File** 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.

Using assembler source files

Test RealTime, provides support for using assembler source code in your projects. Due to their nature, you cannot use Component Testing or Runtime Analysis tools directly on assembler files.

Because assembler file extensions are not standard and depend on your development environment, it is necessary to configure Test RealTime to recognize the file extension used for assembler files. You must specify the assembler file extension:

- In the Project Preferences in order for the GUI to recognize the file type.
- In the TDP Editor for the TDP to recognize assembler files.

To specify the file type preferences:

1. From the **Edit** menu, select **Preferences** and select the **Project** > **Source File Types** page.
2. Click **Add** to create a new line.
3. In the **Extension** column, enter the file extension. For example: ***.asm**.
4. In the **Description** column, enter the description of the file type. For example: **Assembler source files**.
5. Click OK.

To change ASMEXT in the TDP Editor:

1. Open the TDP Editor: from the Tools menu, select Target Deployment Port Editor > Start.
2. In the TDP Editor select **Basic Settings** and the native language of the TDP.
3. Double-click the **ASMEXT** customization point, and add the assembler file extension. For example: **asm**.
4. Save the TDP and quit the TDP Editor.

To add the assembler files to your project.

1. In the Project explorer, right click an test or application node and select **Add Child > Files**.

2. Select the corresponding file type; and locate and select the assembler files that you want to use in your project.
3. Click OK.

Unloadable libraries

In some cases, the architecture of an application requires that shared libraries are loaded and unloaded dynamically during the execution in order to optimizing memory usage.

Test RealTime supports this behavior by allowing you to specify this in the Configuration settings of the project. There are two steps to this:

- Define a shared library as unloadable
- Specify an application as using unloadable libraries

To use unloadable libraries in a project:

1. In the **Project Explorer**, click the **Settings** button.
2. Select an application or test node in the **Project Explorer** pane.
3. In the **Configuration Settings** list, expand **Build > Build Target Deployment Port**.
4. On **Use unloadable library**, select **Yes**.
5. Select the library node of your unloadable shared library in the **Project Explorer** pane.
6. In the **Configuration Settings** list, expand **Build > Build Target Deployment Port**.
7. On **Build as unloadable library**, select **Yes**.
8. When you have finished, click **OK** to validate the changes.

Using shared libraries

Test RealTime, provides support for using, testing and profiling shared libraries with any C or C++ test or application node.

Shared libraries must be stored inside library nodes within the project in order for them to be accessed by test or application nodes. The library node is a container for the source files of the shared library.

Once the library has been included in the project, you must create link the library to the test or application by creating a reference node in the test or application node.

There are three steps that you must follow in order to use a shared library in your project:

- Create a library node in the project.
- Specify how the library is to be linked (statically or dynamically).
- Create a reference to the library in the test or application node.

To add a shared library to your project:

1. Right-click a group or project node and select **Add Child** and **Library** from the popup menu.
2. Enter the name of the Library node
3. Right-click the Library node and select **Add Child** and **Files** from the popup menu.
4. Select the source files of the shared library and click **OK**.

To specify link settings for a library node:

1. Select a library node in the **Project Explorer** pane.

2. In the **Project Explorer**, click the **Settings** button.
3. Select the Build > Linker page and select the **Generation Format**:

- Static library (.lib, .a)
- Dynamic library (.dll, .so)
- Executable file (.exe)

4. When you have finished, click **OK** to validate the changes.

To link a library node to a test or application node:

1. Right-click the test or application node that will use the shared library and select **Add Child** and **Reference** from the popup menu.
2. Select the library that you want to reference and click **OK**.

Example

An example demonstrating how to test and profile shared libraries is provided in the **Shared Library** example project. See Example projects for more information.

Viewing node 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.

Renaming a node

Renaming a node in the Project Explorer involves modifying the properties of the node.

To change the name of a node:

- In the **Project Explorer**, right-click the node that you want to modify.
- Select **Properties** in the pop-up menu.
- Change the **Name** of the node.
- Click **OK**.

Adding files to a 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.

Importing files

Importing files from a Microsoft Visual Studio project

The Test RealTime GUI offers the ability to create a project by importing source files from an existing Microsoft Visual Studio 6.0 or .NET project.

> **Note** The Import feature merely imports a list of files as referenced in the Visual Studio project. It does not import everything you need to immediately build a project in Test RealTime.

The makefile import feature creates a new project, reads the **.dsp** or **.vcproj** project file and adds the source files found in the Visual Studio project to the Test RealTime project. The project is created with the default Configuration Settings of the current Target Deployment Port (TDP).

Any other information contained in the Visual Studio project, such as compilation options, must be entered manually in the **Configuration Settings** dialog box.

Alternatively, you can import the files as a sub-project of the Test RealTime current project. In this case, the sub-project inherits the Configuration Settings of the master project.

To import files from a Microsoft Visual Studio project as a new project:

1. Close any open projects.
2. From the **File** menu, select **Import > Import from Visual Studio 6.0 Project** or **Import from Visual Studio .NET Project**.
3. Use the file selector to locate a valid **.dsp** or **.vcproj** project file 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 and click **OK**.

To import files from a Microsoft Visual Studio project as a sub-project:

1. With a project open, select the project node.
2. Right-click the project node and select **Add Child > Import**.
3. Use the file selector to locate a valid **.dsp** or **.vcproj** project file and click **Open**.
4. In the **Project Explorer**, click **Settings** .
5. Enter any specific compilation options in the **Build** settings and click **OK**.

Importing files from a makefile

The Test RealTime GUI offers the ability to create a project by importing source files from an existing makefile.

> **Note** The Import Makefile feature merely imports a list of files as referenced in the makefile. It does not import everything you need to immediately build a project in Test RealTime.

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 created 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. The following limitations apply:

- Source files must be referenced in the build line
- The makefile cannot be recursive
- Any external commands such as Unix Shell commands are not imported
- Complex operations with variables cannot be imported

Any environment variables used within the makefile must be valid.

You can also use Import Makefile feature to import any list of files contained in a plain text file.

Alternatively, you can import the project as a sub-project of the Test RealTime current project. In this case, the sub-project inherits the Configuration Settings of the master project.

To import files from a makefile as a new project:

1. Close any open projects.
2. From the **File** menu, select **Import > Import from Makefile**. Use the file selector to locate a valid makefile and click **Open**.
3. Enter a name for the new project and click **OK**.
4. Select the correct Configuration in the Configuration toolbar.
5. In the **Project Explorer**, click **Settings** .
6. Enter any specific compilation options in the **Build** settings and click **OK**.

To import files from a makefile as a sub-project:

1. With a project open, select the project node.
2. Right-click the project node and select **Add Child > Import**.
3. Use the file selector to locate a valid makefile and click **Open**.
4. In the **Project Explorer**, click **Settings** .
5. Enter any specific compilation options in the **Build** settings and click **OK**.

Importing sub-projects

Sub-projects are projects that are grouped together within a master project. Projects can contain one or more sub-projects which are actually links to other project directories. The behaviour of a sub-project is the same as a project.

There are two ways of setting up a master project:

- Add the projects manually to a new or existing project. Use this method to import projects one by one from different locations or to add sub-projects to an existing project.
- Imports all the projects contained in a specific directory into a master project. Use this method to automatically import many sub-projects when they are all located in the same directory.

To add an existing sub-project:

1. Create a new project or open an existing project.
2. Select **File** > **Add Project** > **Existing Project**. This opens the file selector.

3. Locate and select an **.rtp** project file and click **OK**.

To create a new sub-project:

1. Create a new project or open an existing project.
2. Select **File** > **Add Project** > **New Project**. This opens the Add New Project wizard.
3. Enter a name and location for the new project, and click **Finish**. The new sub-project is created with the configuration settings of the super-project.

To create a master project containing all sub-projects from a directory:

1. Close any open projects
2. Select **File** > **Import** > **Import multiple Test RealTime projects**.
3. Enter the name of the new master project and the location of the existing projects and click **OK**. The new project is created in the selected directory and imports all the projects found in all sub-directories of that location. When browsing many directories, the import can take a long time.

Importing a data table (.csv file)

IBM Rational Test RealTime Component Testing for C and C++ provide the ability to import **.csv** table files and to turn these into standard **.h** header files. The resulting header file uses the same filename with a **.h** extension. 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 most spreadsheet programs or a text editor.

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. The **.csv** table file must be located before the **.ptu** test script in the test node.
5. Edit the **.ptu** test script to manually add an include statement of the resulting **.h** header file.

   Note The **.csv** data table file must be located before the **.ptu** test script in the test node. If not, then you must manually build the **.csv** data table file before building the test node.

CSV File Format

The formatting rules for the **.csv** file are as follow:

- The first line contains the names of the variable arrays separated by the default CSV separator specified in the preferences or the Configuration settings.
- 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 value 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.

The separator options for the **.csv** file are defined in two locations:

- Data tables preferences: These specify the default behavior for Test RealTime.
- Data tables section in the General Configuration settings: These allow you to override the default settings for a particular project of test node.

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};
```

Editing code and test scripts

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 with Test RealTime is its tight integration with the rest of the GUI. 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

3. From the **File** menu, select **New...** and then open the **Text File** option
4. 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.

You can also 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, Ada and C++ 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 from the filename extension.

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.

Commenting code in the text editor

The text editor allows you simply to comment and uncomment blocks of source code or test script. The same principle also applies to declaring native C code in a C test script by prefixing each with a dash (#) character.

To comment a block of source code

1. In the text editor, select a block of code.
2. Click the **Comment** (**--** or **//** depending on the language) button in the toolbar.

To uncomment a block of commented source code

1. In the text editor, select a block of commented code.
2. Click the **Uncomment** (**--** or **//** depending on the language) button in the toolbar.

To declare native code in a .ptu test script

1. In the **.ptu** test script, select a block of native C code.
2. Click the **Native #** button in the toolbar.

Running tests and applications

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 each particular node.
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:

1. 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.

Selecting Build Options for a Node

The Test RealTime GUI allows you to specify the actions that will be performed during a build for each node in the test project.

Build options contain two sections:

- **Stages** contains the compilation options. In most cases, you will need to select the **All** option to ensure the test is up to date.
- **Runtime Analysis** allows you to enable debugging and Runtime Analysis tools.

Build options are linked to each node through the Configuration Settings mechanism. For example, you can decide to only apply Code Coverage to one node in the project. If you want you changes to apply to the entire project, set the build options on the project node.

By default, the build options 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.

To set the build options of a node:

1. In the **Project Explorer**, click the **Settings** button.
2. Select a node in the **Project Explorer** pane.
3. Select the **Build** node.
4. Click the **Value** column the **...** button.
5. Select the Runtime Analysis features (Memory Profiling, Performance Profiling, Code Coverage and Runtime Tracing) and build options to use them on the current node.
6. Click **OK** and **Apply**.

Excluding a Node from a Build

In some cases, you might want to exclude one or several nodes from the build process. This can be done by changing the Build state of the node 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, there are three possible build states:

Build state	Symbol	Description
Build	⚙	The node is normally built and executed.
Report only	R	The node is not built, but is used to produce the report.
Exclude from Build	✕	The node is not built and ignored.

The **Report only** option means that only static result files (**.tsf** and **.fdc**) are used to generate the report, but the node is not built and does not produce any dynamic results.

To change the Build state of a node:

1. In the **Project Explorer**, click the Build state symbol to toggle the three different states.
2. In the **Properties** window set the Build property to No.

Excluding a Node from Instrumentation

In some cases, you might want to exclude one or several source files from the instrumentation process. This can be done directly in the Project Explorer, as described below, or through the Properties window.

- Instrumented files are displayed with a blue icon
- Non-instrumented files are displayed with a white icon

You can combine both of the following methods to exclude or include a large number of files from the instrumentation process.

To exclude entire directories from instrumentation:

1. In the **Project Explorer**, click the **Settings** button.
2. Select Runtime Analysis, General Runtime Analysis and Selective Instrumentation.
3. In **Directories excluded from Instrumentation**, add the directories to be excluded.
4. Click **Ok**.

To turn off instrumentation for an individual node:

1. In the **Project Explorer**, select the node that you want to exclude from the build.
2. In the **Properties** window set the **Instrumented** property to **No**.

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.

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. See the **Rational Target Deployment Guide** for further information.

> **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.

Setting environment variables

Solaris, Linux or HP-UX platforms

To set an environment variable with a csh shell:

1. Open a shell window
2. Type the following command:
```
setenv <variable> <value>
```

To set an environment variable with a sh, ksh, or Bourne shell:

1. Open a shell window
2. Type the following commands:

```
<variable>=<value>
export <variable>
```

Windows platforms

To set an environment variable under Windows NT, 2000 or XP:

1. From the **Start** menu, select **Parameters**, **Control Panel**, and double-click **System**.
2. Select the **Advanced** tab and click **Environment variables**.
3. Click the **New...** button to add the new environment variable.
4. Click **OK**.

Viewing reports

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 tools

Most reports are produced as **.xrd** files, which are generated during the execution of the test or application node.

To navigate through the report:

1. 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**.
2. 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.

1. From the **Report Viewer** menu, select **Failed Tests Only** or click the **Failed Tests Only** button in the Report Viewer toolbar.
2. 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 filter out certain elements of a report.

1. From the **Report Viewer** menu, select the elements that you want to hide or show.

Exporting reports to HTML

You can export the following Test and Runtime Analysis reports to HTML.

- Memory Profiling
- Performance Profiling

- Code Coverage
- Static Metrics
- Component Testing for C and Ada
- Component Testing for C++
- System Testing for C

There are two methods of exporting to HTML, depending on whether you are viewing the report in a loaded project or you are viewing the report as a standalone document.

To export to HTML from a project:

1. Select a report in the Project Browser.
2. Select **File** > **Export** project report in HTML file format.
3. Choose between exporting the entire project (all the report files contained in the project) or only the selected report.
4. Select the type of report to export (only if you have selected the entire project) and the directory where you want the HTML files to be generated.
5. Click **Export**.

To export to HTML when no project is open:

1. Open the report.
2. In the Report Viewer menu (labelled as the type of report), select **Generate HTML**.
3. Select the directory where you want the HTML files to be generated.
4. Click OK.

Understanding Reports

Test RealTime generates Test and Runtime Analysis reports based on the execution of your application.

Runtime analysis reports

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

Static analysis reports

- Static metrics
- Code review

Test verdict reports

- Component Testing for C and Ada
- Component Testing for C++
- System Testing for C

Setting the zoom level

UML sequence diagrams and other reports can be viewed with different zoom levels.

To set the zoom level:

1. You can directly change the zoom level in the View Toolbar by using the **Zoom In** and **Zoom Out** buttons

   Or

2. Select one of the pre-defined or custom levels from the **Choose Zoom Level** box of the View Toolbar.

Displaying a report summary header

In some cases, test reports can be quite large and complicated when all you want is a quick summary. The report viewer can display a short summary header at the top of a Component Testing test report.

The summary header contains:

- The name of the report
- The number of failed and passed tests
- The total number of tests

To display the summary header for the current test:

1. Open a test report
2. From the **Test Report** menu, select **Show Header**.

To display a full summary for the entire project:

1. Right-click the main project node
2. Select **View Report** and **Test**.
3. From the **Test Report** menu, select **Show Header**.

Monitoring the test process

About the 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 Only the Code Coverage tool provides data for the test process monitor. You can, however, build your own files with the Test Process Monitor tool (**tpmadd**).

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**.

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**.

Customizing tools

Custom tools overview

The **Tools** menu is a user-configurable menu that allows you to access personal tools from the Test RealTime 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.

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 the type of tool:
- 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.
- Select **External Editor** if the tool is an editor. When **External Editor** is selected, you can select **Automatic Launch** if you want this editor to replace the Test RealTime editor for file extensions specified in the **Files Filter** list. (for example: "***.c;*.cpp;*.txt**").
- Select **Other** if the tool is neither a configuration management tool nor an editor.
- 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** if you want to view the tool's text output to be sent to a specific tab 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 GUI macro variables and functions.

A series of macro variables is available to pass parameters on to your tool's command line. See **GUI Macro Variables** in the **Reference** section for detailed information about using the macro command language.

Click **OK** to validate any changes made to the Tool Edit dialog box.

Examples

IBM Rational ClearCase is preconfigured in the Tools menu as the default configuration management tool. If you are using another tool you can simply add it to the Tools menu.

To add CVS to the Tools menu:

1. Select **Tools** > **Configure Tools** and click **Add**.
2. On the **Identification** page, enter **CVS** in the **Name** field, and select **Change Management System**.
3. On the **Actions** page, enter the following command lines:
   - Add to Source Control: **cvs -add $$VCSITEMS**
   - Check Out: cvs -co $$VCSITEMS
   - Check In: cvs -ci $$VCSITEMS
4. Click **OK**.

To add the Windows Notepad editor to the Tools menu:

1. Select **Tools** > **Configure Tools** and click **Add**.
2. On the **Identification** page, enter **Notepad** in the name field, and select **External Editor**.
3. If you want Notepad to replace the default editor for **.java** and **.c** files for example, then select Automatic launch and enter:

   ***.java;*.c**
4. On the **Actions** page, enter:

   notepad.exe $$NODEPATH
5. Click **OK**.

Customizing the Tools menu

The **Tools** menu is a user-configurable menu that allows you to access personal tools from the Test RealTime graphical user interface (GUI). You can customize the Tools menu to meet your own requirements.

In the **Tools** menu, each tool appears as a submenu item, or **Name**, with one or several associated actions or **Captions**.

The **Tool Configuration** dialog allows you to configure a new or existing tool.

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. Select **Tools** > **Configure Tools**.
2. To add a new tool, click **Add...** If you want to create and modify a copy of an existing tool, select the existing tool, click **Copy** and click **Edit...**
3. Edit the tool in the **Tool Edit** box. See Custom tools overview.
4. Click **OK** and **Close**.

To edit a user-defined tool:

1. Select **Tools** > **Configure Tools**.
2. Select the tool that you want to modify and click **Edit...**
3. Edit the tool in the **Tool Edit** box. See Custom tools overview.
4. Click **OK** and **Close**.

To remove a tool from the Tools menu:

1. Select **Tools** > **Configure Tools**.
2. Select an existing tool from the tool list.
3. Click **Remove** and **Close**.

Chapter 6. Using the command line interface

IBM 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 **studio** command line to launch the GUI and run the corresponding project node.

When not using the GUI to execute a node, you must create source files that can execute Test RealTime tests or acquire runtime analysis data without conflicting with the your native compiler and linker. In both cases – that is, regardless of whether you are attempting to execute a Test or Application node – the native compiler and linker do the true work.

For Test nodes, the following commands convert Test RealTime test scripts into source files supported by your native compiler and linker:

- **attolpreproC** for the C language
- **atoprepro** for the C++ language
- **attolpreproADA** for the Ada language

Java does not require a special command because the test scripts are already **.java** files.

For Runtime Analysis, the primary choice is whether or not you wish to perform source code insertion (SCI) as an independent activity or as part of the compilation and linkage process. Of course, if no runtime analysis is required, source code insertion is unnecessary and should not be performed. To simply perform source code insertion, use the binaries:

- **attolcc1** for the C language
- **attolccp** for the C++ language
- **attolada** for the Ada language
- **javi** for the Java language

However, if the user would like compilation and linkage to immediately follow source code insertion, use the binaries:

- **attolcc** for the C and C++ language
- **javic** for the Java language for standard compilation
- inclusion of the **javic.jar** library, and calls to **javic.jar** classes, as part of an ant-facilitated build process

The following sections provide details about the most common use cases.

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.

An HTML output option produces a set of HTML reports in a specified directory. The output is the same as exporting to HTML from the original reports. With this option, it is not necessary to open the GUI to view the reports.

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> [-html <directory>]
   ```
   where:
   - *<node>* is the node to be executed.
   - *<project>* is the **.rtp** project file.
   - *<directory>* is the output directory for the optional HTML output.

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 -html project/output
```

Command Line Runtime Analysis for C and C++

The runtime analysis tools 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
   ```
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, launch the **Graphical User Interface** to view the test reports. See Opening Reports from the Command Line.

Command Line Runtime Analysis for Java

The runtime analysis tools 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.

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 section.

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 section for information on the options and settings.
3. After execution, to obtain the final test results, as well as any trace dump information, you need to separate the output file into separate result files:

- For Memory Profiling for Java, the **studio** command splits the .jpt JVMPI trace dump file.
- For all other features, the **atlsplit** command splits the .spt SCI trace dump file.

  See Splitting the trace dump file for details about this task.

4. Finally, launch the **Graphical User Interface** to view the test reports. See Opening reports from the command line.

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 section.
3. Use the generated **.ptu**, **.otc** or **.otd** templates to write a test script. See the Reference section 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 tools, 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 Reference section.
13. Finally, launch the **Graphical User Interface** to view the test reports. See Opening Reports from the Command Line.

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 **.java** templates to write a test script. See the Java Test Primitives reference section.
4. Compile the generated test harness source file. See Compiling the Test Harness.
5. If you are using any of the runtime analysis tools, 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 section.
12. Finally, launch the **Graphical User Interface** to view the test reports. See Opening Reports from the Command Line.

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 System Testing language reference.
3. Write a System Testing **.spv** supervisor script. See System Testing language reference.

   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 tools, 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 System Testing Report Generator - atsmerge.
12. Finally, launch the **Graphical User Interface** to view the test reports. See Opening Reports from the Command Line.

Command line examples

This section describes an example of using Test RealTime Runtime Analysis tools through the Command Line Interface:

Note This example is for UNIX platforms only.

This example demonstrates using Runtime Analysis tools through the **attolcc** Instrumentation Launcher. The example application is the Apache Web Server, which is widely available for most platforms.

Additionally, the Apache Web Server is a multi-process, multi-tasking application written in C where particular attention must be paid to tracking memory leaks.

To prepare for the example:

1. Download the **apache_1.3.27.tar.gz** archive of the Apache web server source code from:

   http://www.apache.org/dist/httpd/

2. Copy the archive file to the directory where you will perform the tests (for example, **/projects/Apache_Test**) and untar the archive:
   ```
   cp /projects/download/apache_1.3.27.tar.gz .
   tar zxvf apache_1.3.27.tar.gz
   cd apache_1.3.27
   ```

3. To set up the Test RealTime environment, type the following command:
   ```
   . <install_dir>/TestRealTime.2003.06.00/testrtinit.sh
   ```
   where <*install_dir*> is the installation directory of the product.

   Refer to the **Rational Test RealTime Installation Guide** for information about setting up and launching the product.

To compile the application with the Runtime Analysis features

1. To configure the Apache release, type the following command:
   ```
   ./configure --prefix=`pwd`
   ```

2. To compile the Apache server with instrumentation, type the following commands:
   ```
   REP=`pwd`
   make -C src/main gen_test_char
   make CC="attolcc -mempro -perfpro -trace -proc=ret -block=l -keep --
   atl_multi_process=1 --atl_traces_file=$REP/atlout.spt -- gcc"
   ```
   This compiles the application with the following options:

   - Memory Profiling instrumentation enabled
   - Performance Profiling instrumentation enabled
   - Runtime Tracing instrumentation enabled
   - Instrumentation of procedure inputs, outputs, and terminal instructions
   - Instrumentation of simple, implicit and logical blocks (loops)
   - Keep instrumented files
   - Multi-process support

3. To install the Apache server, type the following command:
   ```
   make install
   ```
   This should display a message indicating that you have successfully built and installed the Apache 1.3 HTTP server.

To run the application and view runtime analysis results

1. Optionally, edit the configuration file **apache_1.3.27/conf/httpd.conf**.
2. To start the Apache server, type the following command:

```
/projects/Apache_Test/apache_1.3.27/bin/apachectl start
```

3. To stimulate the application, start a web browser on port 8080 (see the **httpd.conf** file), type the following command:

   ```
   netscape <IP Address>:8080
   ```

   where <IP Address> is the IP address of the machine hosting the Apache server.

4. To stop the Apache server, type the following command:

   ```
   /projects/Apache_Test/apache_1.3.27/bin/apachectl stop
   ```

5. To split the results, type the following command:

   ```
   atlsplit *.spt
   ```

6. To start the Test RealTime GUI to view the results, type the following command:

   ```
   studio  `find . -name "*.fdc"` `find . -name "*.tsf"` *.tio *.tpf
   *.tqf *.tdf
   ```

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.

Note Environment variables concerning Java on Windows must not contain spaces. Use 8.3 naming conventions, for example: **TESTRTDIR=C:\PROGRA~1\Rational\TESTRE~1**.

Most of these environment variables are set during installation of the product. Under Linux, use the testrtinit.sh script to set these variables. See the Reference section for more information about these scripts.

Automated Testing

If you are using Component Testing or System Testing features, 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 **$TESTRTDIR/lib/***<platform>***/***<os>*
- On HP-UX platforms: **SHLIB_PATH** points to **$TESTRTDIR/lib/***<platform>***/*<os>***
- **On AIX platforms: LIB_PATH** points to **$TESTRTDIR/lib/***<platform>***/***<os>*

*w*here *<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**.

   The directory of your choice may be the directory where the generated source files or instrumented source files are located.

2. 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**.

3. 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
```

4. 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 tools (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, Ada, C++ or Java 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 reference section:

- C and C++ Instrumentation Launcher - attolcc
- Java Instrumentation Launcher - javic

Instrumentation and Compilation

Alternatively, you can use the actual Instrumentor command line tools to instrument the source files.

See the reference section:

- C and C++ Instrumentor - attolcc1 and attolccp
- Ada Instrumentor - attolada
- Java Instrumentor - javi

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 section.

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 tools, 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 section:

- C and C++ Instrumentor - attolcc1 and attolccp
- Ada Instrumentor - attolada
- Java Instrumentor - javi

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.

You just have to add the Target Deployment Port library object to the object files linked together.

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.

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 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.

TDP compilation fails	When using the -I- linker option, the TDP fails to compile. This is because the following line is added to the instrumented file:
	`#include "<path to target directory>/TP.h"`
	where **TP.h** includes other files using the **#include** syntax, such as:
	`#include "clock.h"`
	where **clock.h** is in the same directory as "TP.h". If you use the -I- flag, the compiler no longer searches the same directory as the current file (**TP.h**) and therefore cannot find **clock.h**. If you cannot remove the **-I-** flag, you

	must add a **-I** flag for the compiler to find the include files required by the TDP.
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.

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.

The method for splitting the trace dump file is different, depending on the output file produced.

- Memory Profiling for Java uses a **.jpt** JVMPI trace dump file.
- All other features use an **atlout.spt** SCI dump file.

Splitting an SCI dump file

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:

1. Use the **atlsplit** tool supplied in the **/bin** directory of Test RealTime :

```
atlsplit atlout.spt
```

After the split, depending on the selected runtime analysis tools, 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 Profiling Viewer

Splitting a JVMPI trace dump file (Memory Profiling for Java only)

The Memory Profiling for Java features produces a **.jpt** file. When opening this file for the first time, the **.jpt** must be split into a **.tpf** file and a **.txf** file by studio.

To split and view the Java trace file:

1. Run the **studio** GUI with the following parameters:

```
studio <jpt file> <txf file> <tpf file>
```

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

- **.tpf Memory Profiling report files:** view with Report Viewer
- .txf Java Dynamic report files: view with Report Viewer

Opening Reports from the Command Line

Once the test harness or application has been successfully run, you will want to view the test result files in the Test RealTime. To do this, simply invoke the **studio** binary with the corresponding result files. Some reports require at least two files to be opened simultaneously.

- Code Coverage: **.fdc** and **.tio** files
- Memory Profiling for C and C++: **.tsf** and **.tpf** files
- Memory Profiling for Java: **.tpf** and **.txf** files
- Performance Profiling: **.tsf** and **.tqf** files
- Runtime Tracing: **.tsf** and **.tdf** files
- Test Reports: **.xrd** files

Alternatively, you can launch the GUI (studio) and use the Browse Reports feature to open the report files. See Opening a Report.

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 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 and C++ and Performance Profiling results.

Examples

To open the Runtime Tracing UML sequence diagram:

```
studio MyApp.tsf MyApp.tdf
```

To open a Java Memory Profile report for the first time:

```
studio MyApp.jpt MyApp.txf MyApp.tpf
```

To open a Java Memory Profile report once the **.txf** an **.tpf** files have been generated:

```
studio MyApp.txf MyApp.tpf
```

To open a Test Report file

```
studio MyTest.xrd
```

Chapter 7. Using source code insertion

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.

IBM Rational Test RealTime makes extensive use of source code insertion 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.

Source code insertion 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 selected, each covered branch is referenced by an array that uses

- 1 byte in Default mode
- 1 bit in Compact mode
- 4 bytes in Hit Count mode

The actual 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 and C++.

See Information Modes for more information.

Other Specifics:

- **Loops, switch and case statements:** a 1-byte local variable is declared for each instance
- **Modified/multiple conditions:** one *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).

As a summary, **Hit Count** mode and modified/multiple conditions produce the greatest data and execution time overhead. In most cases you can select each coverage type independently and use

Pass mode by default in order to reduce this overhead. The source code can also be partially instrumented.

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

This applies to Memory Profiling for C and C++. 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**.

Optimizing the Information Mode

When using Code Coverage, you can specify the Information Mode which defines how much coverage data is produced, and therefore stored in memory.

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(1)** instrumentation pragma (the parameter 1 is ignored).

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.

You can automatically add an explicit dump your C and C++ source code by clicking the **Add Dump** button in the text editor. This inserts the **_ATCPQ_DUMP** instrumentation pragma into your source code.

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 number of the signal that is to trigger the trace dump.

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.

Chapter 8. Working with other development tools

Rational Test RealTime is a versatile tool that is designed to integrate with 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.

Note Before using any configuration management tool, you must first configure the CMS Preferences dialog box. See Customizing Configuration Management.

You can also set up the GUI to use a CM system of your choice.

Working with IBM Rational ClearCase

IBM Rational ClearCase is a configuration management system (CMS) 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, Test RealTime offers configuration management support for ClearCase. You can however customize the product to support different configuration management software. When using 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.

Source Control Commands.

For any file in the Test RealTime project, ClearCase, or any other CMS tool, can be accessed through a set of source control commands.

Source control can be applied to all files and nodes in the Project Browser or Asset Browser. When a source control command is applied to a project, group, application, test or results node, it affects all the files contained in that node.

The following source control commands are included for use with ClearCase:

- Add to Source Control
- Check Out
- Check In
- Undo Check Out
- Compare to Previous Version
- Show History

- Show Properties

Please refer to the documentation provided with ClearCase for more information about these commands.

Source control commands are fully configurable from the **Tools** menu.

To control files from the Tools menu:

1. Select one or several files in the **Project Explorer** window.
2. From the **Tools** menu, select **Rational ClearCase** and the source control command that you want to apply.

To control files from the Source Control popup menu:

1. Right-click one or several files in the **Project Explorer** window.
2. From the popup menu, select Source Control and the source control command that you want to apply.

Working with IBM Rational ClearQuest

IBM Rational ClearQuest is a defect and change tracking 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 with Test RealTime you can directly submit a report from a test or runtime analysis report.

To submit a ClearQuest report from Test RealTime:

1. In the **Report Explorer**, right-click a test.
2. From the pop-up menu, select **Submit ClearQuest 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.

Customizing source control tools

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 Eclipse C/C++ Development Tools

The Test RealTime plug-in for Eclipse CDT allows you to use the Runtime Analysis tools with the Eclipse C/C++ Development Tools (CDT) without leaving the Eclipse environment.

If you installed Eclipse and Eclipse CDT after Test RealTime, you must manually install the Test RealTime plug-in for Eclipse. Installation is described in the **IBM Rational Test RealTime Installation Guide**.

Test RealTime plugin for Eclipse overview

The Test RealTime plugin for Eclipse allows you to use the Runtime Analysis tools with the Eclipse C/C++ Development Tools (CDT) on Windows platforms without leaving the Eclipse environment.

The Test RealTime plugin for Eclipse CDT relies on the following third party product versions:

- Eclipse version 3.1.0
- Eclipse CDT version 3.0.0
- EMF/XSD version 2.1.0
- Java 2 Platform, Standard Edition, version 1.4.2
- Cygwin 1.5 or later

It is important that these exact versions are used. Any later or earlier versions of these products might cause the Test RealTime integration to fail.

Please refer to the **IBM Rational Test RealTime Installation Guide** for details on installing these products and the Test RealTime plugin.

How the Test RealTime plugin for Eclipse works

The plugin supports managed and unmanaged C and C++ projects in the Eclipse development platform.

In Eclipse, you engage the runtime analysis tools on a managed or unmanaged C or C++ project. You set the configuration settings for the project or for each source file, run the source code, and view the runtime analysis results in Eclipse.

Enabling runtime analysis tools on an Eclipse project

To use the runtime analysis tools on your Eclipse project files, you must enable the project for Test RealTime. This creates the Test RealTime configuration for your project.

Before you enable the project, you must ensure that the correct Target Deployment Port is selected in the Test RealTime preferences.

> **Note** By default, the selected TDP is for C. If you are using C++, you must select a C++ TDP.

When you first enable Test RealTime on a project, a default configuration is created based on the default TDP as specified in the Test RealTime preferences. You can change the active configurations later in the Set Active Configurations.

To enable an Eclipse C or C++ project for Test RealTime:

1. Select **Window** > **Preferences** and **Test RealTime** and ensure the correct Target Deployment Port is selected in **Default TDP**. Click **OK**.
2. In Eclipse, right click the C or C++ project and select **Test RealTime > Enable**. This creates a Test RealTime configuration based on the default TDP.

Managing configurations in Eclipse

Test RealTime *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 project element overridden settings.

Configuration settings are a main characteristic of the project and can be individually customized for any single project element in the C/C++ project navigator.

To copy an existing configuration:

1. In the C or C++ Eclipse project, expand the **Test RealTime** folder
2. Right click the folder of an existing configuration and click **Copy**.
3. Right click the **Test RealTime** folder and click **Paste**.
4. Rename the new folder and the settings file as required.

   Note This can be useful if you want several configurations, with different custom settings, based on a unique Target Deployment Port, or if you want to create a new configuration.

To remove a configuration from an Eclipse project:

1. In the C or C++ Eclipse project, expand the **Test RealTime** folder.
2. Right click the folder of an existing configuration and click **Delete**.

   Note If you choose to remove a configuration, all custom settings for that Configuration will be lost.

To change the active Test RealTime configurations in a project:

1. In Eclipse, right click the C or C++ project and select **Test RealTime > Set Active Configuration**.
2. In the **Set Active Configurations** window, select the configurations that will apply to the Eclipse project.
3. Click **OK.**

To change the Target Deployment Port in a configuration:

1. In the C or C++ Eclipse project, expand the **Test RealTime** folder.
2. Open the **.settings** configuration.
3. Expand **Configuration properties** > **Build** > **Build options**.
4. Select **Target Deployment Port** and change the setting if necessary.
5. Close the configuration editor to save your changes.

Managing configuration settings in Eclipse

The Eclipse CDT integration adds a Configuration Settings editor to Eclipse. The Test RealTime settings are added to the C or C++ project when the project is enabled for Test RealTime.

For the usage of configuration settings, see Configuration settings reference.

To edit the configuration settings for an Eclipse project element:

1. In the C or C++ Eclipse project, expand **Test RealTime** and the folder for the current configuration.
2. Double-click the settings. This opens the Configuration Settings editor.

3. In the **Configuration properties** list on the left, locate the settings that you want to change.
4. On the right, select the setting and change the value.
5. When you have finished, click **Save** to validate the changes.

Running a project with Test RealTime in Eclipse

Before you can run a project in Eclipse, you must create a run configuration.

To create a run configuration:

1. From the **Run** menu, select **Run**.
2. In the **Configurations** list select **C/C++ Application under Test RealTime** and click **New**.
3. Enter a name for the new run configuration.
4. Click **Select** to select the project that will use this run configuration.
5. Click **Select** or **Browse** to specify the name of the C/C++ application.
6. Click **Apply** and **Close** to create the configuration and quit, or click **Run** to create and run the configuration.

To run an existing configuration:

1. From the **Run** menu, select **Run**.
2. Select the run configuration and click **Run**.

or

3. Click the **Run** toolbar button.

Viewing runtime analysis reports in Eclipse

After execution, you can view the results of the Test RealTime runtime analysis in the Test RealTime report view.

To open Test RealTime results in Eclipse:

1. In the **C/C++ Projects** view, expand **Test RealTime** and double-click **results.xtp**. This opens the Test RealTime report view in Eclipse.
2. At the bottom of the Test RealTime report, click the tabs corresponding to runtime analysis tool that you want to see. There is a page for each tool that was selected in the Test RealTime **Build** settings.

Each page of the report is identical to the corresponding report viewer in Test RealTime. See Viewing reports for more information about each viewer.

Test RealTime preferences in Eclipse

The Test RealTime preferences in the Eclipse workbench allow you to configure settings for Test RealTime in Eclipse.

Test RealTime preferences

The Test RealTime preferences allow you to change the following settings:

- **Binary Directory:** Specifies the directory where Test RealTime binaries are located.
- **Default TDP:** Specifies the default TDP that will be used in the **Default.settings** configuration when you enable Test RealTime in a C or C++ project.

- **Verbose Mode:** Enables detailed information of Test RealTime components in the console during execution.

Results Editor preferences

The **Results Editor** preferences allow you to change the appearance of your Test and Runtime Analysis reports in Eclipse.

These preferences are identical to the corresponding preferences in the Test RealTime user interface.

- **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.

To access the Test RealTime preferences in Eclipse:

1. In Eclipse, select **Window** > **Preferences**.
2. In the **Preferences** window, expand **Test RealTime**.

Working with IBM Rational Rose RealTime

IBM 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.

If you installed Rose RealTime after Test RealTime, you must manually install the plug-in. Installation of the Rose RealTime plug-in is covered in the **Test RealTime Installation Guide**.

Using Test RealTime with Rose RealTime

Before using IBM Rational Test RealTime as a Rose RealTime plug-in, you must first open or create a model within Rose RealTime.

Test RealTime can perform Source Code Insertion (SCI) instrumentation on several components.

Note If you installed Rose RealTime after Test RealTime, you must manually install the plug-in. Please refer to the **IBM Rational Test RealTime Installation Guide** for further information.

To activate Runtime Analysis tools:

1. From Rose RealTime, open the Component Specification of the components that you want to observe and select the **C++ TestRT** 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 tools 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 operation whenever you change any of the options described above.

To run a build with the runtime analysis tools:

1. 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 dump data in Rose RealTime

The Source Code Insertion (SCI) technology used for Test RealTime 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 and .NET, such an object file is already provided in: <*install dir*>**\bin\intel\RoseRT\cvisual6** or **cvisual7** 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

Once the application has run and dump data has been collected, you can view the execution results in Rose RealTime. See Using Test RealTime with Rose RealTime for the main steps to using Test RealTime with Rose RealTime.

To view the results with Test RealTime report viewers:

1. In Rose RealTime, from the **Tools** menu, select **Rational Test RealTime**, **View all Results** 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:

1. In Rose RealTime, from the **Tools** menu, select **Rational Test RealTime**, **Model Coverage** and either:
   - Show Onto Selected State Diagram
   - Hide From Selected State Diagram
   - Show Onto all State Diagram
   - Hide from all State Diagram
2. This displays State Diagram colored with Code Coverage. Default colors are:
   - Green: covered code
   - Red: non-covered code

Advanced Rose RealTime integration

This section covers the more advanced information about using Test RealTime with IBM Rational Rose RealTime.

Using a Cross Compiler with Rose RealTime

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.

To use a cross compiler:

1. Locate the corresponding Target Deployment Ports. These TDPs must contain an **attolcc** Instrumentation Launcher binary.
2. In the **TDP.txt** file located in the Rose RealTime installation directory, write a line for each Target Deployment Ports based on the following syntax:

```
<rosert_targetRTS_name>  ,  <testrt_tdp_name> [, [<path>], [$], [/|\]]
```

where:

- *<rosert_targetRTS_name>* is the name of the Rose RealTime TDP.
- *<testrt_tdp name>* is the name of the Test RealTime TDP.
- *<path>* is the location of the **CmdCommand.o** file in the Rose RealTime targetRTS
- The **$** option indicates to use environment variable names instead of their values.
- Use the **/** or **** option to specify the use of the '/' or '\'directory separator if these are not the platform default.

For example:

```
NT40T.x86-VisualC++-6.0  ,  cvisual6 , C:/temp ,$ ,/
```

Using 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:

To compile with a makefile:

1. 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.

Splitting the Result File

The instrumented application produces the **atlout.spt** file at the end of the execution.

To display the report

1. 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

Problem	Solution
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

Problem	Solution
An application cannot 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 cannot 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

Problem	Solution
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** section for further information about Instrumentation Launcher command line options.

Results

Problem	Solution
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 IBM Rational TestManager

IBM Rational TestManager is used to manage all aspects of testing and all sources of information related to the testing effort throughout all phases of the 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.
- 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

  Note If you installed TestManager after Test RealTime, you must manually install the plug-in. Please refer to the **IBM Rational Test RealTime Installation Guide** for further information.

Installing and configuring the TestManager integration

Test RealTime (on Windows installations only) includes a integration with IBM Rational TestManager. In order for the integration to work, TestManager and Test RealTime must be installed on the same workstation.

Before you are able to use the two products together, the plug-in must be installed by doing the following:

To install the plug-in:

1. Ensure that both TestManager and Test RealTime are not running.
2. From the Windows **Start** menu select **Programs > Rational Software > Test RealTime > Tools > Enable Rational Project TestManager Integration**.

   Note Test Manager must be installed before Test RealTime is installed otherwise the **Enable Rational Project TestManager Integration** menu item will not appear in the Start menu.

Associating test nodes to test cases

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. See the chapter Installing TestManager Integration.

The Rational project and Test RealTime project can be located on any network-accessible drive - and not necessarily the same drive - but test execution must occur locally, on the Windows machine upon which TestManager and Test RealTime are co-installed. TestManager does not support remote target execution of Test RealTime tests.

TestManager associates test cases with nodes found within the Test RealTime Project Browser. You can only associate TestManager test cases with Test RealTime test nodes and group nodes. Test RealTime 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 an individual test node is preserved.)

To associate a TestManager test case with a test node or group 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** window, 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 Test Node or Group Node** window now displays a list of all top-level group and test nodes.
4. 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, a group node containing one or more test nodes or child nodes of group nodes.

   The text box in the **Automated Implementation** section of the test case **Properties** window 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 and the test or group node names.

Accessing Test RealTime test nodes and group nodes

You can open and execute Test RealTime test nodes and group nodes from within TestManager. This implies that you have previously built a working project in Test RealTime.

When a Test RealTime test node or group node is executed from within Test Manager, the Test RealTime GUI does not appear and tests are run silently. To obtain full feedback of test execution, run the node directly from within Test RealTime.

To open a test or group node from within TestManager:

1. In TestManager, from the **File** menu, select **Open Test Script** and **Rational Test RealTime**.
2. Browse to a Test RealTime project in the Rational Test RealTime Test Selection window and then press the **Open button.**
3. The window **Select a Test Node or Group Node** lists all the test nodes and group nodes within the project. Left-click the node representing the test you wish to view and then click **OK**.
   The corresponding project opens in Test RealTime.

To execute a test node or group node from within TestManager:

1. In the **Test Plan** window of TestManager, right-click the test case and select **Run**.
2. In the **Run Test Cases** window, select the execution options and click **OK**.

   Note Test RealTime test nodes and group nodes can also be associated with TestManager test cases as test implementations. This means that Test RealTime test and group nodes can also be executed as part of a TestManager suite.

Viewing results in TestManager

Once a Test RealTime test has been executed from within a TestManager suite, test results are accessible from the TestManager Test Log window.

For each TestManager test case, the Test Log window displays a single User Defined line for each 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 (accessed via a right-click of the User Defined line and then selecting Properties), 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.

Submitting a ClearQuest defect from TestManager

In TestManager you can submit a defect relating to a Test RealTime test node as part of the TestManager - 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.

Troubleshooting the TestManager integration

If you are experiencing problems related to the TestManager integration, the following troubleshooting guide might help you to find a solution.

If a problem persists, do not hesitate to contact IBM Customer support for help.

Problem	Solution
Once the test has finished executing, when trying to view the reports through the **Detail** tab, Test RealTime launches but no graphical results are shown.	When creating the test asset you must select both a project and also a test node. If no test node is selected, the test will pass but no results will be generated and no reports will be displayed.
When configuring the **Test Case Properties**, when I select the **Automated Implementation** on the **Implementation** tab, a **General Data Store error** message is displayed.	There are two possible causes for the generation of this error: • The **TestRTConsoleAdapter.dll** has been placed in **Rational\Rational Test\tsea** instead of **\Rational\Rational Test** • The version of the TestRTConsoleAdapter.dll file which has been installed is wrong. For example, it may occur that a file from a previous version was not updated when upgrading to Test RealTime 2003.06.13.
When selecting an **Automated Implementation**, I click **Select**, but Rational Test RealTime is not shown as an option.	This means that the integration has not been enabled for the project. See Installing and configuring the TestManager integration. This problem can also occur if you tried to install the plug-in while TestManager was running. You must not be running TestManager when you install the plug-in

Working with Microsoft Visual Studio

Configuring Microsoft Visual Studio integration

Test RealTime provides a special setup tool to configure runtime analysis tools 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:** Enable or Disable the runtime analysis tools
- **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.

Integration with Microsoft Visual Studio is only available for the Windows versions of Test RealTime .

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.

To disable

Configuring Microsoft Visual Studio integration

Test RealTime provides a special setup tool to configure runtime analysis tools 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:** Enable or Disable the runtime analysis tools
- **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.

Notices

Glossary

A

ABR: Array Bounds Read

ABW: Array Bounds Write

ABWL: Late Detect Array Bound Write on the Heap

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.

B

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.

BSR: Beyond Stack Read

BSW: Beyond Stack Write

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.

C

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.

COM: Com API/Interface Failure

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.

COR: Core Dump

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.

D

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.

E

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.

EXC: Continued Exception

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.

EXH: Handled Exception

EXI: Ignored Exception

EXU: Unhandled Exception

F

FFM: Freeing Freed Memory

FIM: Freeing Invalid Memory

FIU: File In Use

FMM: Freeing Mismatched Memory

FMR: Freeing Memory Read

FMW: Free Memory Write

FMWL: Late Detect Free Memory Write On the Heap

FUM: Freeing Unallocated Memory

G

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.

H

HAN: Invalid Handle Use

HIU: Handle In Use

I

ILK: Com Interface Leak

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.

IPR: Invalid Pointer Read

IPW: Invalid Pointer Write

J

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.

M

MAF: Memory Allocation Failure

MC/DC: Modified Condition/Decision Coverage.

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.

MIU: Memory In Use

MLK: Memory Leak

Model: A representation intended to explain the behavior of some aspects of [an artifact or activity]. A model is considered an abstraction of reality.

N

Node: Any item that appears in the Project Explorer. This includes test nodes, application nodes, source files or test scripts.

NPR: Null Pointer Read

NPW: Null Pointer Write

O

ODS: Output Debug String

P

Package (ADA): Program units that allow the specification of groups of logically related entities.

Package (Java): A group of types (classes and interfaces).

PAR: Bad System Api Parameter

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.

PLK: Potential Memory Leak

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.

R

Requirement: A desired feature, property, or behavior of a system.

Runtime Tracing: The Test RealTime feature whose function is to monitor code s 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.

S

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.

SIG: Signal Received

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.

T

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.

U

UMC: Uninitialized Memory Copy

UML: Unified Modeling Language. A general-purpose notational language for specifying and visualizing complex software, especially large, object-oriented projects.

UMR: Uninitialized Memory Read

Unit: Generic term referring to language specific code elements such as procedures, classes, functions, methods, packages.

Unit Testing: See Component Testing.

W

White box testing: See Clear box testing.

Index

Printed in USA