Rational® PurifyPlus for Linux

User Guide

VERSION: 2003.06.00

UNIX

Rational®
the **software development** company

Legal Notices

©2001-2003, Rational Software Corporation. All rights reserved.

Any reproduction or distribution of this work is expressly prohibited without the prior written consent of Rational.

Version Number: 2003.06.00

Rational, Rational Software Corporation, the Rational logo, Rational Developer Network, AnalystStudio, , ClearCase, ClearCase Attache, ClearCase MultiSite, ClearDDTS, ClearGuide, ClearQuest, ClearTrack, Connexis, e-Development Accelerators, DDTS, Object Testing, Object-Oriented Recording, ObjecTime, ObjecTime Design Logo, Objectory, PerformanceStudio, PureCoverage, PureDDTS, PureLink, Purify, Quantify, Rational Apex, Rational CRC, Rational Process Workbench, Rational Rose, Rational Suite, Rational Suite ContentStudio, , Rational Summit, Rational Visual Test, Rational Unified Process, RUP, RequisitePro, ScriptAssure, SiteCheck,SiteLoad, SoDA, TestFactory, TestFoundation, TestStudio, TestMate, VADS, and XDE, among others, are trademarks or registered trademarks of Rational Software Corporation in the United States and/or in other countries. All other names are used for identification purposes only, and are trademarks or registered trademarks of their respective companies.

Portions covered by U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329 and 5,574,898 and 5,649,200 and 5,675,802 and 5,754,760 and 5,835,701 and 6,049,666 and 6,126,329 and 6,167,534 and 6,206,584. Additional U.S. Patents and International Patents pending.

U.S. GOVERNMENT RIGHTS. All Rational software products provided to the U.S. Government are provided and licensed as commercial software, subject to the applicable license agreement. All such products provided to the U.S. Government pursuant to solicitations issued prior to December 1, 1995 are provided with "Restricted Rights" as provided for in FAR, 48 CFR 52.227-14 (JUNE 1987) or DFARS, 48 CFR 252.227-7013 (OCT 1988), as applicable.

WARRANTY DISCLAIMER. This document and its associated software may be used as stated in the underlying license agreement. Except as explicitly stated otherwise in such license agreement, and except to the extent prohibited or limited by law from jurisdiction to jurisdiction, Rational Software Corporation expressly disclaims all other warranties, express or implied, with respect to the media and software product and its documentation, including without limitation, the warranties of merchantability, non-infringement, title or fitness for a particular purpose or arising from a course of dealing, usage or trade practice, and any warranty against interference with Licensee's quiet enjoyment of the product.

Third Party Notices, Code, Licenses, and Acknowledgements

Portions Copyright ©1992-1999, Summit Software Company. All rights reserved.

User Guide Contents

Preface

Welcome to Rational PurifyPlus for Linux.

This User Guide contains extensive information on a broad range of subjects that will help you enhance your software development experience with Rational PurifyPlus RealTime.

PurifyPlus for Linux is a complete solution for runtime analysis on Linux platforms: it delivers memory leak detection, memory and performance profiling as well as code coverage and runtime tracing.

Advanced usage of the product is described in the *PurifyPlus for Linux Reference Manual*.

If you are using the product for the first time, please take the time to go through the *PurifyPlus for Linux Online Tutorial*.

Audience

This guide is intended for Rational software users who are using PurifyPlus for Linux, such as application developers, quality assurance managers, and quality assurance testers.

You should be familiar with the selected Linux platform as well as your C, C++ or Java development environment.

Contacting Rational Technical Publications

To send feedback about documentation for Rational products, please send e-mail to our technical publications department at techpubs@rational.com.

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

Other Resources

All manuals are available online, either in HTML or PDF format. The online manuals are on the CD and are installed with the product.

For the most recent updates to the product, including documentation, please visit the Product Support section of the Web site at:

> http://www.rational.com/products/pqc/pplus_lx.jsp

Documentation updates and printable PDF versions of Rational documentation can also be downloaded from:

> http://www.rational.com/support/documentation/index.jsp

For more information about Rational Software technical publications, see:

> http://www.rational.com/documentation.

For more information on training opportunities, see the Rational University Web site:

> http://www.rational.com/university.

Customer Support

Before contacting Rational Customer Support, make sure you have a look at the tips, advice and answers to frequently asked questions in Rational's Solution database:

http://solutions.rational.com/solutions

Choose the product from the list and enter a keyword that most represents your problem. For example, to obtain all the documents that talk about stubs taking parameters of type "char", enter "stub char". This database is updated with more than 20 documents each month.

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

- **About you:**
  Name, title, e-mail address, telephone number

- **About your company:**
  Company name and company address

- **About the product:**
  Product name and version number (from the **Help** menu, select **About**).
  What components of the product you are using

- **About your development environment:**
  Operating system and version number (for example, Linux RedHat 8.0), target compiler, operating system and microprocessor.

- **About your problem:**
  Your service request number (if you are calling about a previously reported problem)
  A summary description of the problem, related errors, and how it was made to occur
  Please state how critical your problem is
  Any files that can be helpful for the technical support to reproduce the problem (project, workspace, test scripts, source files). Formats accepted are **.zip** and compressed tar (**.tar.Z** or **.tar.gz**)

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

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

> **Note** When sending e-mail concerning a previously-reported problem, please include in the subject field: "**[SR#**_<number>_**]**", where _<number>_ is the service request number of the issue. For example:
> Re:[SR#12176528] New data on PurifyPlus for Linux install issue

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

Location	Contact
North America	Rational Software, 18880 Homestead Road, Cupertino, CA 95014
	voice: (800) 433-5444 fax: (408) 863-4001
	e-mail: support@rational.com
Europe, Middle East, and Africa	Rational Software, Beechavenue 30, 1119 PV Schiphol-Rijk, The Netherlands
	voice: +31 20 454 6200

	fax: +31 20 454 6201
	e-mail: support@europe.rational.com
Asia Pacific	Rational Software Corporation Pty Ltd, Level 13, Tower A, Zenith Centre, 821 Pacific Highway, Chatswood NSW 2067, Australia
	voice: +61 2-9419-0111 fax: +61 2-9419-0123
	e-mail: support@apac.rational.com

Product Overview

PurifyPlus for Linux is a complete solution for runtime analysis on Linux platforms: it delivers memory leak detection, memory and performance profiling as well as code coverage and runtime tracing.

With one mouse-click, your application is equipped with the robust functionality of these features:

- **Memory Profiling**: Providing memory leak detection for all of your C, C++ and Java code

- **Performance Profiling**: Providing function-level performance profiling for all of your C, C++ and Java code

- **Code Coverage**: Providing full code coverage and metrics for C, C++ and Java in a color-coded source-code graphical user interface (GUI)

- **Runtime Tracing**: Providing runtime-generated UML sequence diagrams of interactions between C, C++ or Java modules, classes, and instances

Discover the power of runtime analysis capabilities of Rational PurifyPlus for Linux. Know your code inside and out without having to change the way you work.

Source Code Insertion

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

Rational PurifyPlus for Linux makes extensive use of SCI technology to transparently produce analysis reports on both native and embedded target platforms.

Estimating Instrumentation Overhead

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

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

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

Code Coverage Overhead

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

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

Depending on the information mode 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)** macro definition (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.

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.

Target Deployment Ports

Rational's Target Deployment Technology is a versatile, low-overhead technology that allows Rational PurifyPlus for Linux to accommodate your compiler, linker and debugger. Your project is independent of the TDP, so you don't need to reconfigure it when you change your development environment.

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 PurifyPlus for Linux 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 PurifyPlus for Linux :

1.   From the **Project** menu, select **Configurations**.

2.   Select the TDP and click **Remove**.

3.   Click **New**, select the TDP and click **OK**.

Reconfiguring a TDP for a Compiler or JDK

During installation of Rational PurifyPlus for Linux, 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 PurifyPlus for Linux.

If you would like to make a different GNU compiler or JDK accessible in PurifyPlus for Linux, you need to rerun these dialogs. From a command prompt, execute the applicable shell script:

- **ConfigureGcc.sh** to locate a GCC compiler

- ConfigureJavac.sh to locate a JDK

The two shell scripts are located in the following folders:

- *<install dir>*/releases/PurifyPlusForLinux.v2002R2/bin/intel/linux_redhat

- *<install dir>*/releases/PurifyPlusForLinux.v2002R2/bin/intel/linux_suse

These shell scripts depend on a properly configured environment. If, for some reason, your environment is not properly configured (indicated by the interactive dialogs):

- Set your current directory to the applicable shell script folder mentioned above

- Execute the environment configuration shell script:
  `. ppluslinuxinit.sh`

Once finished, for any existing project within which you would like to use the newly supported compiler or JVM, you must update the project.

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

1. Open the project in PurifyPlus for Linux.

2. From the **Project** menu, select **Configuration**.

3. In the **Configurations** window, click **New**.

4. In the **New Configuration** window, select the newly supported compiler or JDK in the dropdown list and click **OK**.

5. In the **Configurations** window, click **Close**.

Unified Modeling Language

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

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

Model Elements and Relationships in Sequence Diagrams

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

Activations

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

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

Shape

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

Activation   Nested Activations
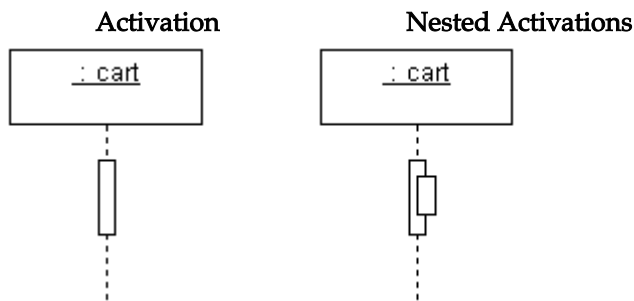
Using Activations

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

- On lifelines depicting instances (actors, classifier roles, or objects), an activation typically appears as the result of a message to indicate the time during which an instance is active.

- On lifelines involved in complex interactions, nested activations (also known as stacked activations or nested focuses of control) are displayed to indicate nested stack frames in a calling sequence, such as those that happen during recursive calls.

- On lifelines depicting concurrent operations, the entire lifeline may appear as an activation (thin rectangles) instead of dashed lines.

Naming Conventions

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

Classifier Roles

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

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

Shape

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

Classifier Role   Classifier Role with Lifeline

/ Observer   / Observer
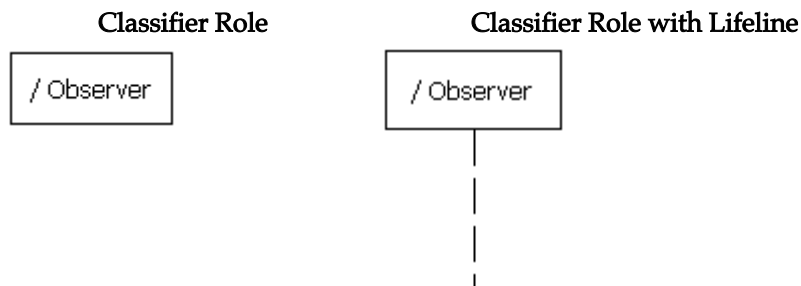
Using Classifier Roles

Classifier roles can appear on a model to represent the following:

- In models depicting role-based interactions, a classifier role represents an instance in an interaction. Using classifier roles instead of objects can provide two advantages: First, a class can serve as the base classifier for multiple classifier roles. Second, instances of a class can realize multiple classifier roles in one or more collaborations.

- In models depicting patterns, a classifier role specifies the kind of object that must ultimately fulfill a role in the pattern. The classifier role shows how the object will participate in the pattern, and its reference to a base class defines the attributes and operations that are required for participation in the pattern. When the pattern is used in the model, classes are bound to the collaboration to identify the type of objects that realize the classifier roles.

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

Naming Conventions

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

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

Destruction Markers

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

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

Shape

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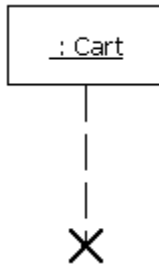
Naming Conventions

Destruction markers do not have names.

Lifelines

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

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

Shape

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

Lifeline for an Object
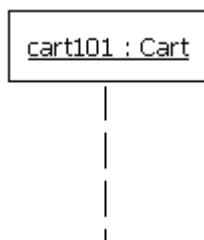
Lifeline for a Classifier Role
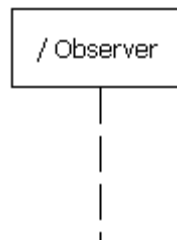
Using Lifelines

When a classifier role or object appears in a sequence diagram, it will automatically have a lifeline. Lifelines indicate the following:

- Creation – If an instance is created during the interaction, its lifeline starts at the level of the message or stimulus that creates it; otherwise, its lifeline starts at the top of the diagram to indicate that it existed prior to the interaction.

- Communication – Messages or stimuli between instances are illustrated with arrows. A message or stimulus is drawn with its end on the lifeline of the instance that sends it and its arrowhead on the lifeline of the instance that receives it.

- Activity – The time during which an instance is active (either executing an operation directly or through a subordinate operation) can be shown with activations.

- Destruction – If an instance is destroyed during the interaction, its lifeline ends at the level of the message or stimulus that destroys it, and a destruction marker appears; otherwise, its lifeline extends beyond the final message or stimulus to indicate that it exists during the entire interaction.

Naming Conventions

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

Messages

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

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

Shape

A message appears as a line with an arrow. The direction of the arrow indicates the direction in which the message is sent. In a sequence diagram, messages usually connect two classifier role lifelines.
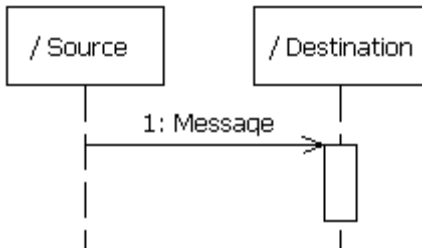
Message shapes can be adorned with names and sequence numbers.

Types of Messages

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

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

Using Messages

Messages can appear in a sequence diagram to represent the communications exchanged between classifier roles during dynamic interactions.

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

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

Naming Conventions

Messages can be identified by a name or operation signature.

Type	Example	Description
Name	// Get the Password	A name identifies only the name of the message. Simple

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

Objects

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

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

Shape

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

Object   Object with Lifeline

cart101 : Cart

cart101 : Cart

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

Active Object

cart101 : Cart

Multiobject

: Item

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

Types of Objects

The following table identifies three types of objects.

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

Using Objects

Objects can appear in a sequence diagram to represent concrete and prototypical instances. A concrete instance represents an actual person or thing in the real world. For example, a concrete instances of a "Customer" class would represent an actual customer. A prototypical instance represents an example person or thing. For

example, a prototypical instance of a "Customer" class would contain the data that a typical customer would provide.

Naming Conventions

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

Syntax	Example	Description
object/role:class	cart100/storage:cart	Named instance (cart100) of the cart class that is playing the storage role during an interaction.
object:class	cart100:cart	Named instance (cart100) of the cart class.
/role:class	/storage:cart	Anonymous instance of the cart class playing the storage role in an interaction.
object/role	cart/storage	An object named cart playing the storage role. This object is either an object that is hiding the name of the class or an instance that is not associated with a class.
object	cart100	An object named cart100. This object is either an instance that is hiding the name of the class or an instance that is not associated with a class.
/role	/storage	An anonymous instance playing the storage role. This object is either an instance that is hiding the name of the object and class or an instance that is not associated with an object or class.
:class	:cart	Anonymous instance of the customer class.

Stimuli

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

Shape

A stimulus appears as a line with an arrow. The direction of the arrow indicates the direction in which the stimulus is sent. In a sequence diagram, a stimulus usually connects two object lifelines.
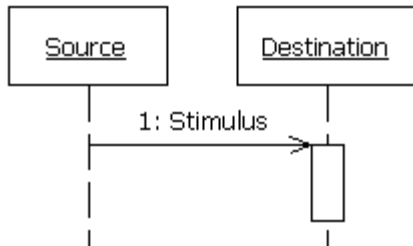
Stimulus shapes can be adorned with names and sequence numbers.

Types of Stimuli

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

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

Naming Conventions

Stimuli can have either names or signatures.

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

> name of the operation and its parameters.
> Signatures are often used in diagrams developed
> during design because the provide the detail that
> developers need when they code the design.

Actions

An action is represented as shown below:

```
Action/Ins#1
```

The action box displays the name of the action.

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

Exceptions

When tracing C++ exceptions, Runtime Tracing locates the throw point of the exception (the throw keyword in C++) as well as its catch point.

Exceptions are displayed as a slanted red line, as shown in the example below, generated by Runtime Tracing.
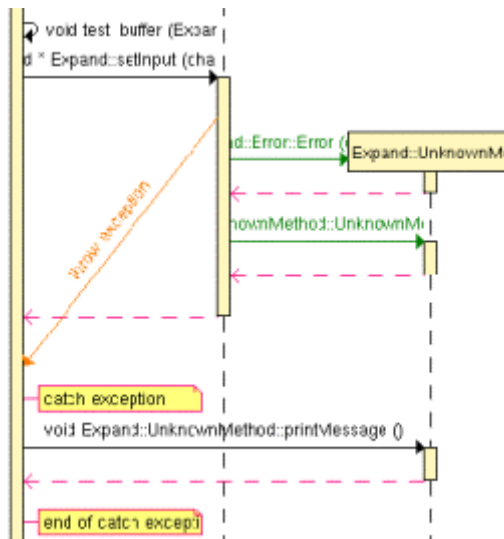
To jump to the corresponding portion of source code:

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

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

To filter an instance out of the UML sequence diagram:

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

Actors

An actor is a model element that describes a role that a user plays when interacting with the system being modeled. Actors, by definition, are external to the system. Although an actor typically represents a human user, it can also represent an organization, system, or machine that interacts with the system. An actor can correspond to multiple real users, and a single user may play the role of multiple actors.

Shape

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

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

Naming Conventions

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

Loops

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

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

A loop is represented as shown below:
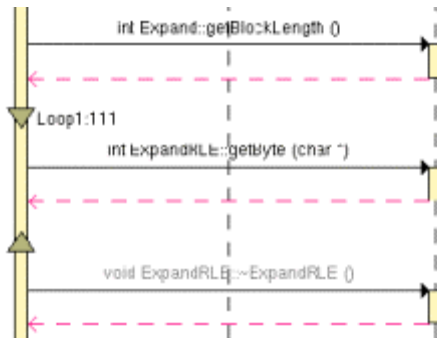
A tag displays the name of the loop and the number of executions.

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

To configure Runtime Tracing to detect loops:

1. From the Project Explorer, select the highest level node to which you want to apply the option, such as the Workspace.

2. Right-click the node, and select **Settings...** from the pop-up menu.

3. In the **Configuration Settings** dialog, select the **Runtime Tracing** node, and **Trace Control**.

4. From the options box, set the **Automatic Loop Detection** to **Yes**.

5. Click **OK**.

Synchronizations

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

Shape

A synchronization is represented as shown below:

The synchronization box displays the name of the synchronization.

The synchronization is linked to its source file. In the UML/SD Viewer, click a synchronization to open the Text Editor at the corresponding line in the source code.
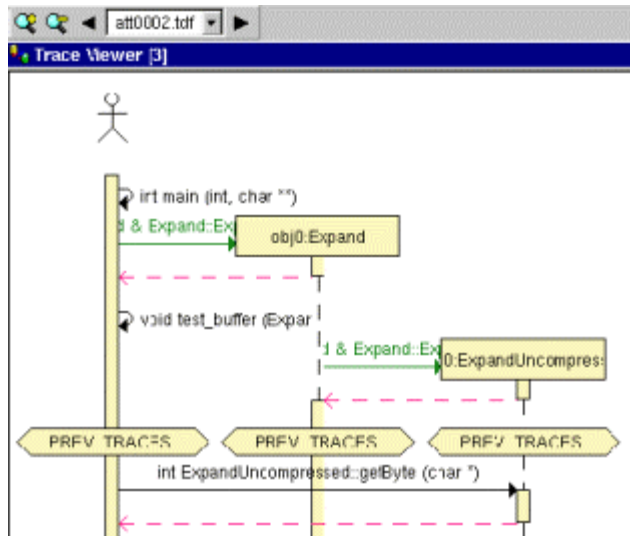
When the Split Trace capability is enabled, the UML/SD Viewer displays the list of TDF files generated in the UML/SD Viewer toolbar.

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

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

Notes

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

sequence diagram notes can be associated to messages and instances.

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

Runtime Analysis

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

Using Runtime Analysis Features

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

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

- Memory Profiling analyzes memory usage and detects memory leaks

- Performance Profiling provides performance load monitoring

- Code Coverage performs code coverage analysis

- Runtime Tracing draws a real-time UML Sequence Diagram of your application

  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.

How to use the runtime 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 features to be applied to the application in the Build options.

5.   Use the Project Explorer to set up the test campaign and add any additional runtime analysis or test nodes.

6.   Run the application node to build and execute the instrumented application.

7.   View and analyze the generated analysis and profiling reports.

Code Coverage

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

The Code Coverage feature brings efficient, easy-to-use robust coverage technologies to real-time embedded systems. Code Coverage provides a completely automated and proven solution for C, C++ and 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** 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 PurifyPlus for Linux 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 application node is executed in the PurifyPlus for Linux GUI.

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	C++	Java
Call Coverage	C		
Condition Coverage	C		
Function, Unit or Method Coverage	C	C++	Java
Link Files			
Templates		C++	
Additional statements	C	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**.

C Coverage

Block Coverage

When running the Code Coverage feature on C source code, PurifyPlus for Linux 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, PurifyPlus for Linux can provide the following condition coverage:

- Basic Coverage

- Forced Coverage

Basic Conditions

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

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

```
/* Power_of_10 function */

/* -cond */
int power_of_10 ( int value, int max )
{
  int result = value, i;
  if ( value == 0 ) return 0;
  for ( i = 0; i < 10; i++ )
  {
    result = max > 0 && ( max / value ) < result ?
               result * value :
               max;
  }
  return result ;
} /* There are 4*2 basic conditions in this function */
/* Near_color function */
ColorType near_color ( ColorType color )
{
  switch ( color )
  {
    case WHITE :
    case LIGHT_GRAY :
      return WHITE;
    case RED :
    case PINK :
    case BURGUNDY :
      return RED;
    /* etc ... */
  }
} /* There are at least 5 basic conditions here */
```

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

Forced Conditions

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

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

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

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

  {
    A->value -= 1;
  }
}
```

Modified Conditions

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

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

```
/* state_control function */
int state_control ( void )
{
  if ( ( ( flag & 0x01 ) &&
         ( instances_number > 10 ) ) ||
       ( flag & 0x04 ) )
    return VALID_STATE;
  else
    return INVALID_STATE;
} /* There are 3 basic conditions, 5 compound conditions
    and 3 modified conditions :
```

```
          flag & 0x01 : TTX=T and FXF=F
          nb_instances > 10 : TTX=T and TFF=F
          flag & 0x04 : TFT=T and TFF=F, or FXT=T and FXF=F
       4 test cases are enough to cover all those modified
       conditions :
          TTX=T
          FXF=F
          TFF=F
          TFT=T or FXT=T
    */
```

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

Code Coverage calculates matching cases for each modified condition.

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

Multiple Conditions

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

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

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

Code Coverage calculates every available case for each composition.

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

Function Coverage

When analyzing C source code, PurifyPlus for Linux 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**.

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 Code Coverage

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

- Statement Blocks

- Statement Blocks and Decisions

- Statement Blocks, Decisions, and Loops

Statement Blocks

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

- THEN and ELSE FOR IF, WHILE and DO ... WHILE blocks

- non-empty blocks introduced by **SWITCH CASE** or **DEFAULT** statements

- true and false outcomes of ternary expressions (*<expr>* ? *<expr>* : *<expr>*)

- **TRY** blocks and any associated catch handler

- blocks following a potentially terminal statement.

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

Inputs to Procedures

Inputs identify the C++ methods executed.

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

One branch per C++ method is defined.

Procedure Inputs, Outputs and Returns, and Terminal Instructions

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

terminate(), as well as the input.

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

```
          }
```

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

Potentially Terminal Statements

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

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

- SWITCH, FOR

- WHILE or DO...WHILE

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);
  }
```

51

```
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.

Code Coverage Viewer

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

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

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

You can jump directly to the next or previous 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 application node

2.   If a Code Coverage report was generated during execution of the node, select **View Report** and then **Code Coverage**.

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 C.

•   **Block code coverage:** select the desired block coverage method.

•   **Condition code coverage:** select condition coverage for C.

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

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

To filter coverage types from the report:

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

2.   Toggle each coverage type in the menu.

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

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**.

Exporting a Report to HTML

Code Coverage results can be exported to an HTML file.

To export results to an HTML file:

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

Source Report

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

Hypertext Links

The Source report provides hypertext navigation throughout the source code:

*   Click a plain underlined function call to jump to the definition of the function.

*   Click a dashed underlined text to view additional coverage information in a pop-up window.

*   Right-click any line of code and select **Edit Source** to open the source file in the **Text Editor** at the selected line of code.

Macro Expansion

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

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

Hit Count

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

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

To activate the Hit Count tool-tip:

1. In the **Code Coverage Viewer** window, select the **Source** tab.

2. From the **Code Coverage** menu select **Hit**. The mouse cursor changes shape.

3. In the **Code Coverage Viewer** window, click a portion of covered source code to display the Hit Count tool-tip.

Cross Reference

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

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

To activate the Cross Reference tool-tip:

1. In the **Code Coverage Viewer** window, select the **Source** tab.

2. From the **Code Coverage** menu select **Cross Reference**. The mouse cursor changes shape.

3. In the **Code Coverage Viewer** window, click a portion of covered source code to display the **Cross Reference** tool-tip.

Comment

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

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

Rates Report

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

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

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

Code Coverage Toolbar

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

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

- The **Previous Link** and **Next Link** buttons allow you to quickly navigate through the Failed items.

- The **Previous Uncovered Line** and **Next Uncovered Line** buttons allow you to quickly navigate through the Failed items.

- The **Failed Tests Only** or **All Tests** button toggles between the two display modes.

- The **F** button allows you to hide or show functions

- The **E** button allows you to hide or show function exits

- The **B** button allows you to hide or show statement blocks

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

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

Code Coverage Viewer Preferences

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

To choose Code Coverage report colors and attributes:

1. Select the **Code Coverage Viewer** node:

   - **Background color:** This allows you to choose a background color for the **Code Coverage Viewer** window.

   - **Stroud Number:** This parameter modifies the results of Halstead Metrics.

2. Expand the **Code Coverage Viewer** node, and select **Styles**:

   - **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.

   - **Font:** This allows you to change the font type and size for the selected style.

   - **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.

   - **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

3. Click **OK** to apply your changes.

Code Coverage Dump Driver

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

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

Static Metrics

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

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

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

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

Halstead Metric – Vocabulary
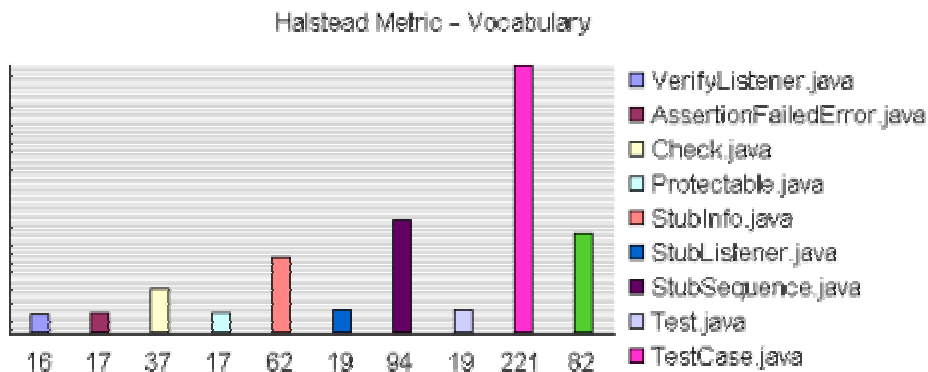
The following display modes are available for the Halstead graph:

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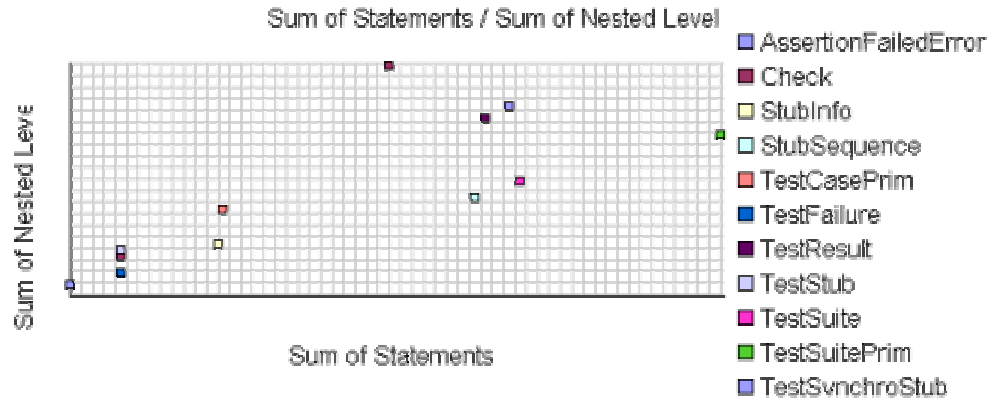
File View is the only available view with C source code. When viewing metrics for C++ and Java, an Object View is also available.

Two modes are available for the data graph:

- Vocabulary

- Size

- Volume

- Difficulty

- Testing Effort

- Testing Errors

- Testing Time

See the Halstead Metrics section for more information.

Metrics Summary

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

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

- **Statements:** shows the total number of statements within the object

- **Nested Levels:** shows the highest statement nesting level reached in the object

- **Ext Comp Calls:** measures the number of calls to components defined outside of the object

- **Ext Var Use:** measures the number of uses of variables defined outside of the object

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

To select the Object View:

1.   Select the **Root** node in the Report Explorer to open the Root page.

2.   Select **Object View** in the **View** box of the Report Explorer.

To switch the object graph mode:

1.   From the **Metrics** menu, select **Object Graph for Root Page**.

2.   Select **ExtVarUse by ExtCompCall** or **Nested Level by Statement**.

Halstead Metrics

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

Halstead provides various indicators of the module's complexity

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

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

	instances	

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

Metric	Meaning	Formula
n	Vocabulary	$n_1 + n_2$
N	Size	$N_2 + N_2$
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 formulas, k is the *Stroud* number, and has a default value of **18**. You can change the value of k in the Metrics Viewer Preferences. Adjustment of the Stroud number allows you to adapt the calculation of T to the testing conditions: team background, criticity level, and so on.

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

V(g) or Cyclomatic Number

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

A value of **1** means the code has no branching.

A function's cyclomatic complexity should not exceed **10**.

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

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

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

Metrics Viewer Preferences

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

To choose Metrics Viewer report colors and attributes:

1. Select the **Metrics Viewer** node:

    *   **Background color:** This allows you to choose a background color for the **Metrics Viewer** window.

    *   **Stroud number:** This parameter modifies the results of Halstead Metrics.

2. Expand the **Metrics Viewer** node, and select **Styles**:

    *   **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.

    *   **Font:** This allows you to change the font type and size for the selected style.

    *   **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.

    *   **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

3. Click **OK** to apply your changes.

Memory Profiling for C and C++

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

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

*   You associate Memory Profiling with an 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 PurifyPlus for Linux 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 application node is executed in the PurifyPlus for Linux GUI.

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.
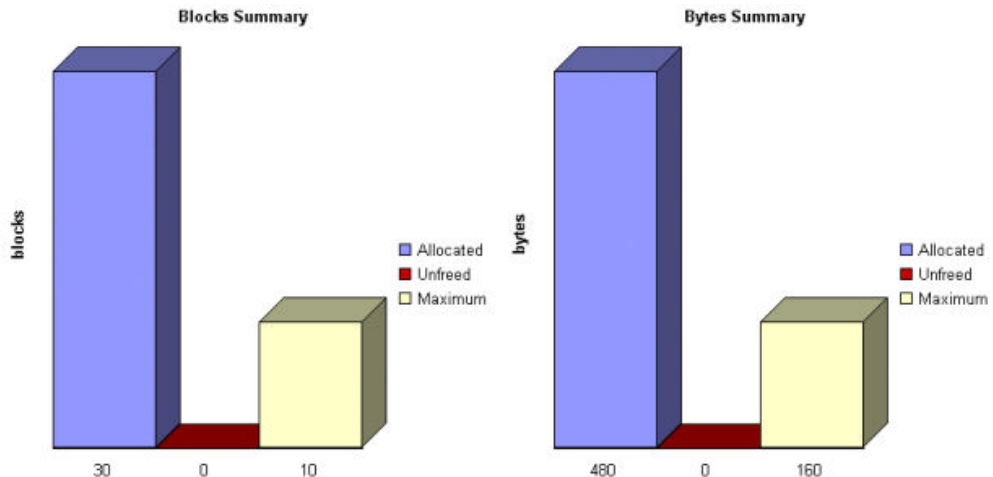
Where:

- **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)

- Late Detect Array Bounds Write (ABWL)

- Late Detect Free Memory Write (FMWL)

- Memory Allocation Failure (MAF)

- Core Dump (COR)

Freeing Freed Memory (FFM)

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

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

Memory Profiling maintains a *free queue*, whose role is to actually delay memory free calls in order to compare with upcoming free calls. The length of the delay depends on the **Free queue length** and **Free queue threshold** Memory Profiling Settings. A large deferred free queue length and threshold increases the chances of catching FFM errors long after the block has been freed. A smaller deferred free queue length and threshold limits the amount of memory on the deferred free queue, taking up less memory at run time but providing a lower level of error detection.

Freeing Unallocated Memory (FUM)

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

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

- Memory on the stack

- Program code and data sections

Late Detect Array Bounds Write (ABWL)

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

This message can occur when you:

- Make an array too small. For example, you fail to account for the terminating NULL in a string.

- Forget to multiply by sizeof(type) when you allocate an array of objects.

- Use an array index that is too large or is negative.

- Fail to NULL terminate a string.

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

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

Increasing the size of the Red Zone helps PurifyPlus for Linux catch bounds errors before or beyond the block.

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

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

Late Detect Free Memory Write (FMWL)

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

This message can occur when you:

- Have a dangling pointer to a block of memory that has already been freed (caused by retaining the pointer too long or freeing the memory too soon)

- Index far off the end of a valid block

- Use a completely random pointer which happens to fall within a freed block of memory

Memory Profiling maintains a *free queue*, whose role is to actually delay memory free calls in order to compare with upcoming free calls. The length of the delay depends on the **Free queue length** and **Free queue threshold** Memory Profiling Settings. A large deferred free queue length and threshold increases the chances of catching FMWL errors. A smaller deferred free queue length and threshold limits the amount of memory on the deferred free queue, taking up less memory at run time but providing a lower level of error detection.

Memory Allocation Failure (MAF)

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

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

Core Dump (COR)

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

Memory Profiling 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.

Exporting a Report to HTML

Memory Profiling results can be exported to an HTML file.

To export results to an HTML file:

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

Memory Profiling Viewer Preferences

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

To choose Memory Profiling report colors and attributes:

1. Select the **Memory Profiling Viewer** node:

    - **Background color:** This allows you to choose a background color for the **Memory Profiling Viewer** window.

2. Expand the **Memory Profiling Viewer** node, and select **Styles**:

    - **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.

    - **Font:** This allows you to change the font type and size for the selected style.

    - **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.

    - **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

3. Click **OK** to apply your changes.

Memory Profiling for Java

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

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

- You associate Memory Profiling with an existing test node or Application code.

- You compile and run your application.

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

The Java 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 instrumented by the Java Instrumentor (**javi**). The resulting source code is then executed and the Memory Profiling for Java feature outputs a static **.tsf** file for each instrumented source file and a dynamic **.txf** file.

These files can be viewed and controlled from the PurifyPlus for Linux GUI. Both the **.tsf** and **.txf** files need to be opened simultaneously to view the report.

Of course, these steps are mostly transparent to the user when the application node is executed in the PurifyPlus for Linux 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. 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 PurifyPlus for Linux after the execution of the application.

- You associate Performance Profiling with an existing application code.

- You build and execute your code in PurifyPlus for Linux .

- 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.

Top 2 Functions

60,05%

16,23%

23,72%

- ■ int ExpandUncompressed::getByte (char
- ■ void test_buffer (Expand *, char *, int, cha
- □ Others (< 5%)
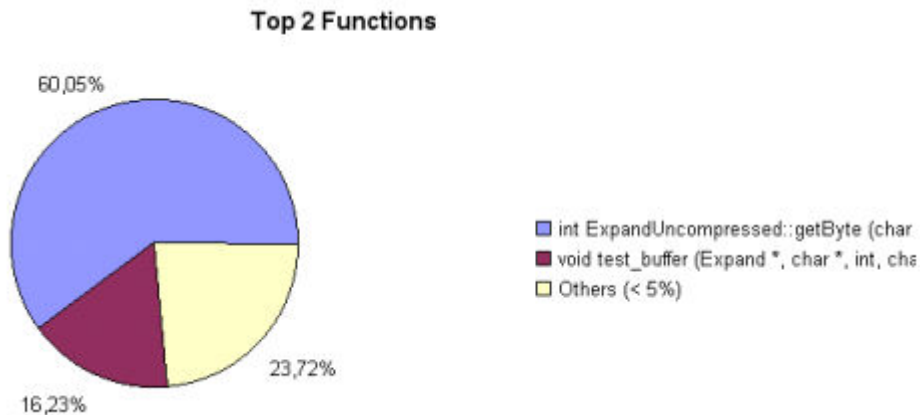
Performance Summary

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

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

- **Function (F) time:** The time required to execute the code in a function exclusive of any calls to its descendants

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

Note that since each of the descendants may have been called by other functions, it is not sufficient to simply add the descendants' F+D to the caller function's F. In fact, it is possible for the descendants' F+D to be larger than the calling function's F+D. The

following example demonstrates three functions *a*, *b* and *c*, where both *a* and *b* each call *c* once:

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

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

- **F Time (% of root)** and **F+D Time (% of root):** Same as above, expressed in percentage of total execution time

- **Average F Time:** The average time spent executing the function each time it was called

Performance Profiling SCI Dump Driver

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

To customize the Performance Profiling Dump Driver, open the Target Deployment Port directory and edit the **atqapi.h**. Follow the instructions and comments included in the source code.

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 **.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**.

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**.

Performance Profiling Viewer Preferences

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

To choose Performance Profiling report colors and attributes:

1. Select the **Performance Profiling Viewer** node:

   - **Background color:** This allows you to choose a background color for the **Performance Profiling Viewer** window.

2. Expand the **Performance Profiling Viewer** node, and select **Styles**:

   - **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.

   - **Font:** This allows you to change the font type and size for the selected style.

   - **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.

   - **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

3. Click **OK** to apply your changes.

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 PurifyPlus for Linux GUI.

- You associate Performance Profiling with an existing test or application code.

- You build and execute your code in PurifyPlus for Linux .

- 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 PurifyPlus for Linux 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 application node is executed in the PurifyPlus for Linux GUI.

Understanding Runtime Tracing UML Sequence Diagrams

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 UML Sequence Diagrams.

Example

Below is an example of object lifelines generated by Runtime Tracing from a C++ application.
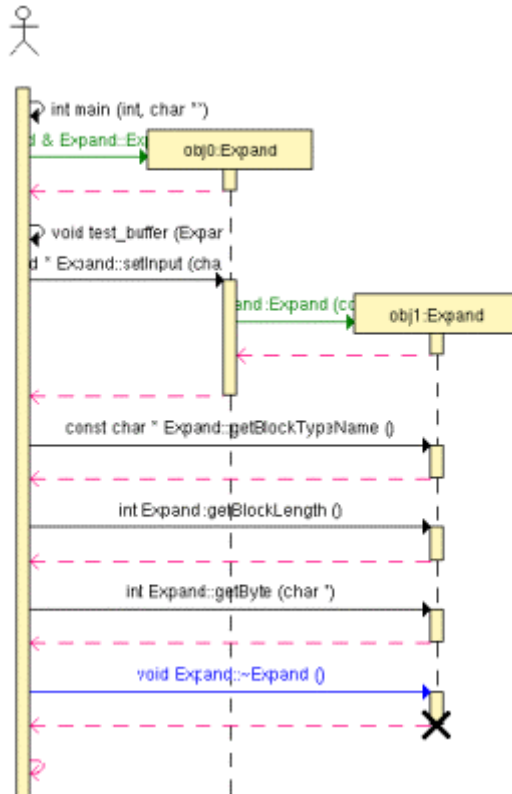
In this C++ example, functions and static methods are attached to the World instance.

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

The black cross represents the destruction of the instance.

Constructors are displayed in green.

Destructors are blue.

Return messages are dotted red lines.

Other functions and methods are black.

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

To jump to the corresponding portion of source code:

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.

Advanced

Multi-Thread Support

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

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

Multi thread settings:

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

- **Maximum number of threads:** This value sets the size of the thread management table inside the Target Deployment Port. Lower values save memory on the target platform. Higher values allow more simultaneous threads.

- **Dump note on thread creation:** When selected, the UML Sequence Diagram displays a note ("**Thread Creation**") each time a new thread is created.

- **Dump note on thread schedule:** When selected, the UML Sequence Diagram displays a note ("**Thread Schedule**") each time a thread's schedule is changed.

Partial Trace Flush

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

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

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

To set Partial Trace Flush from the Node Settings:

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

2. Select one or several nodes in the Configuration pane.

3. Select the **Runtime Analysis** node and the **Runtime Tracing** node.

4. Select **Runtime Tracing Control**.

5. Set the **Partial trace flush** setting to **Yes** or **No** to activate or disable the mode.

6. When you have finished, click **OK** to validate the changes.

To toggle message dump from within the source code:

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 trace flush** setting to **Yes** or **No** to activate or disable the mode.

6. When you have finished, click **OK** to validate the changes.

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

Trace Item Buffer

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

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

To activate or de-activate trace item buffering:

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

2. Select one or several nodes in the Configuration pane.

3. Select the Runtime Analysis node and the Runtime Tracing node.

4. Select Runtime Tracing Control.

5. Set the **Buffer trace items** setting to **Yes** or **No** to activate or disable the mode.

6. Set the size of the buffer in the **Items buffer size** box.

7. When you have finished, click **OK** to validate the changes.

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

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 a node in the **Project Explorer** pane.

4. Select the Runtime Analysis node and the Runtime Tracing node.

5. Select Trace Control.

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

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

8. 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.

Graphical User Interface

The graphical user interface (GUI) of PurifyPlus for Linux provides an integrated environment designed to act as a single, unified work space for all 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, analysis reports, UML sequence diagrams. For example:

- 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.

Discovering the GUI

When you launch the PurifyPlus for Linux Graphical User Interface (GUI), you are first greeted with the Start Page and a series of windows. Click the elements below to learn how to use them:

- The Start Page is your main starting point when you launch the GUI

- The Project Explorer is where you create, develop and execute your project nodes

- The Properties Window provides information about node properties

- The Output Window displays the output of command line tools and compilers

- The Standard Toolbars provide quick and convenient access to the most commonly used features

- The Report Explorer allows you to navigate through analysis reports

Start Page

When you launch the graphical user interface, the first element that appears is the PurifyPlus for Linux Start Page.

The Start Page is the central location of the application. From here, you can create a new project, start a new activity and navigate through existing project reports.

The Start Page contains the following sections:

- **Welcome:** General information for first-time users of the product.

- **Get Started:** This section lists your recent projects as well as a series le projects provided with PurifyPlus for Linux.

- **Activities:** This section displays a series of new activities. Click a new activity to launch the corresponding activity wizard. A project must be open before you can select a new activity.

- **Examples:** A set of sample projects for tutorial or demonstration purposes. You can use these projects to get familiar with the product.

- **Support:** Links to Customer Support and online documentation.

To reset the recent files list:

1. Select the Start page and click the **Reset** button in the toolbar.

2. Click the **Reload** button the reload the Start page.

Output Window

The Output Window displays messages issued by product components or custom features.

The first tab, labelled **Build**, is the standard output for messages and errors. Other tabs are specific to the built-in features of the product or any user defined tool that you may have added.

To switch from one console window to another, click the corresponding tab. When any of the Output Window tabs receives a message, that tab is automatically activated.

When a console message contains a filename, double-click the line to open the file in the Text Editor. Similarly when a test report appears in the Output Window, double-click the line to view the report.

Output Window Actions

Right-click the Output Window to bring up a pop-up menu with the following options:

- **Edit Selected File:** Opens the editor with the currently selected filename.

- **Copy:** Copies the selection to the clipboard.

- **Clear Window:** Clears the contents of the Output Window.

To hide or show the Output Window:

1. From the **View** menu, select **Other Windows** and **Output Window**.

Project Explorer

The Project Explorer allows you to navigate, construct and execute the components of your project. The Project Explorer organizes your workspace from two viewpoints:

- **Project Browser:** This tab displays your project as a tree view, as it is to be executed.

- **Asset Browser:** Source code components are displayed on an object or elementary level.

To change views, select the corresponding tab in the lower section of the **Project Explorer** window.

Project Browser

The **Project Browser** displays the following hierarchy of nodes:

- **Projects:** the Project Explorer's root node. Each project can contain one or more sub-projects.

- **Results:** after execution, this node can be expanded to display the resulting report sub-nodes and files, allowing you to control those files through a CMS system such as Rational ClearCase.

- **Test groups:** provide a way to group and organize application nodes into one or more test campaigns

- **Application nodes:** represent your application, to which you can apply SCI instrumentation for Memory Profiling, Performance Profiling, Code Coverage and Runtime Tracing.

- **External Command nodes:** these allow you to add shell command lines at any point in the Test Campaign.

After execution of an applicationnode, double-click the node to open all associated available reports.

When you run a **Build** command in the **Project Browser**, the product parses and executes each node from the inside-out and from top to bottom. This means that the contents of a parent node are executed in sequence before the actual parent node.

Asset Browser

The Asset Browser displays all the files contained in your project. The product parses the files and displays individual components of your source files, such as classes, methods, procedures, functions, units and packages.

Use the Asset Browser to easily navigate through your source files.

In Asset Browser, you can select the type of Asset Browser in the **Sort Method** box at the top of the **Project Explorer** window. Each view type can be more or less relevant depending on the programming language used:

- **By Files:** This view displays a classic source file and dependency structure

- **By Objects:** Primarily for C++ and Java, this view type presents objects and methods independently from the file structure

- **By Packages:** This is mostly relevant for Java and displays packages and components

Use the **Sort** button to activate or disable the alphabetical sort.

Double-click a node in the Asset Browser to open the source file in the text editor at the corresponding line.

To switch Project Explorer views:

1. Click the **Project Browser** or **Asset Browser** tab.

To hide or show the Project Explorer:

1. Right-click an empty area within the toolbar.

2. Select or clear the **Project Window** menu item.

or from the **View** menu, select **Other Windows** and **Project Window**.

Properties Window

The **Properties Window** box contains information about the node selected in the Project Explorer. It also allows you to modify this information. The information

available in the Properties Window depends on the view selected in the Project Explorer:

- Project Browser

- Asset Browser

Project Browser

Depending on the node selected, any of the following relevant information may be displayed:

- **Name:** Is the name carried by the node in the Project Explorer.

- **Exclude from Build:** Excludes the node from the Build process. When this option is selected a cross is displayed next to the node in the **Project Explorer**.

- **Execute in background:** Enables the build and execution of more than one test or application node at the same time.

- **Relative path:** Indicates the relative path of the file.

- **Full path:** Indicates the entire path of the file.

- **Instrumented type**: You can select either **Yes** or **No**.

Asset Browser

Select the type of Object View in the **Sort Method** box at the top of the **Project Explorer** window: **By Object**, **By Files**, or **By Packages**. Depending on the sort method selected, and the type of object or file, any of the following relevant information may be displayed:

- **Name:** is the name carried of the file, object or package.

- **Filters (for folders):** is the file extension filter for files in that folder. See Creating a Source File Folder.

- **Name:** is the name carried of the file or package.

- **Relative path:** indicates the relative path of the file.

- **Full path:** indicates the entire path of the file.

To open the Properties window:

1. In the **Project Explorer**, right-click a node.

2. Select **Properties...** in the pop-up menu.

To hide or show the Properties window:

1. Right-click an empty area within the toolbar.

2.   Select or clear the *<object>* **Property** menu item.

or from the **View** menu, select **Other Windows** and *<object>* **Property**.

Report Explorer

The **Report Explorer** allows you to navigate through all text and graphical reports, including:

- Memory Profiling, Performance Profiling and Code Coverage reports

- UML Sequence Diagram reports from the Runtime Tracing feature

- Metrics produced by the Metrics Viewer

The actual appearance of the Report Explorer contents depends on the nature of the report that is currently displayed, but generally the Report Explorer offers a dynamic hierarchical view of the items encountered in the report.

Click an item in the Report Explorer to locate and select it in the **Report Viewer** or **UML/SD Viewer** window.

To hide or show the Report Explorer:

1.   Right-click an empty area within the toolbar.

2.   Select or clear the **Report Explorer** menu item.

Standard Toolbars

The toolbars provide shortcut buttons for the most common tasks.

The following toolbars are available

- Main toolbar

- View toolbar

- Build toolbar

- Status bar

Main Toolbar

The main toolbar is available at all times:

- The **New File** button creates a new blank text file in the Text Editor.

- The **Open** button allows you to load any project, source file or report file supported by the product.

- The **Save File** button saves the contents of the current window.

- The **Save All** button saves the current workspace as well as all open files.

- The **Cut** , **Copy** and **Paste** buttons provide the standard clipboard functionality.

- The **Undo** and **Redo** buttons allow you undo or redo the last command.

- The **Find** button allows you to locate a text string in the active Text Editor or report window.

View Toolbar

The View toolbar provides shortcut buttons for the Text Editor and report viewers.

- The **Choose zoom Level** box and the **Zoom In** and **Zoom Out** buttons are classic Zoom controls.

- The **Reload** button refreshes the current report in the report viewer. This is useful when a new report has been generated.

- The **Reset Observation Traces** button clears cumulative reports such as those from Code Coverage, Memory Profiling or Performance Profiling.

Build Toolbar

The build toolbar provides shortcut buttons to build and run the application.

- The **Configuration** box allows you to select the target configuration on which the test will be based.

- The **Build** button launches the build and executes the node selected in the Project Explorer. You can configure the Build Options for the workspace by selecting the **Options** button.

- The **Stop** button stops the build or execution.

- The **Clean Parent Node** button removes files created by previous tests.

- The **Execute Node** button executes the node selected in the Project Explorer.

Status Bar

The Status bar is located at the bottom of the main GUI window. It includes a **Build Clock** which displays execution time, and the **Green LED** which flashes when work is in progress.

To hide or show a toolbar:

1. Right-click an empty area within the toolbar.

2. Select and clear those toolbars you want to display or hide.

or from the **View** menu, select **Toolbars** and the toolbar(s) you want to display or hide.

GUI Components and Tools

In addition to these main windows, the PurifyPlus for Linux 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.

Report Viewer

The Report Viewer allows you to view Runtime Analysis reports from any of the Runtime Analysis features

Most reports are produced as XML-based **.xrd** files, which are generated during the execution of the test or application node.

To navigate through the report:

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**.

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.

Understanding Reports

PurifyPlus for Linux generates Runtime Analysis reports based on the execution of your application.

Runtime Analysis Reports

- Memory Profiling

- Performance Profiling

- Code Coverage

- Runtime Tracing

Setting a 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.

Report Viewer Toolbar

The Report toolbar eases report navigation with the **Report Viewer**.

Report Viewer commands are available when a **Report Viewer** window is open:

- The **Previous Failed Test** and **Next Failed Test** buttons allow you to quickly navigate through the Failed items.

- The **Failed Tests Only** or **All Tests** button toggles between the two display modes.

Report Viewer Style Preferences

The **Preferences** dialog box allows you to change the appearance of your Runtime Analysis reports.

To choose Report Editor colors and attributes

1. Select the **Report Viewer** node:

- **Background color:** This allows you to choose a background color for the **Report Viewer** window.

2. Expand the **Report Viewer** node, and select **Syntax Color**:

- **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.

- **Font:** This allows you to change the font type and size for the selected style.

- **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.

- **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

3. Click **OK** to apply your changes.

Text Editor

The product GUI provides its own Text Editor for editing and browsing 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 PurifyPlus for Linux is its tight integration with the rest of the GUI. You can click items within the **Project Explorer**, **Output Window**, or any 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 PurifyPlus for Linux 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 and C++ source code. 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.

Text Editor Preferences

The **Preferences** dialog box allows you to change the appearance of the source code and scripts in the Text Editor.

To choose Editor report colors and attributes:

1. Select the **Editor** node.

- **Font:** This allows you to change the general font type and size for Editor. This parameter is overridden for defined styles by the **Style** font setting. This parameter can be overridden for defined styles by the **Style** font settings.

- **Global Colors**: This is where you select background colors for text categorized as **Normal**, **Information** or **Error** as well as the general background color. Click a color to open a standard color palette.

- **Autodetect parenthesis and bracket mismatch** - When this option is selected, the **Error** color is used when the Editor detects a missing bracket "[]" or parenthesis "()".

- **Tabulation length:** This specifies the tabulation length, which is equivalent to a number of inserted spaces.

2. Expand the **Editor Viewer** node, and select **Styles**:

- **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.

107

- **Font:** This allows you to change the font type and size for the selected style.

- **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.

- **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

3. Click **OK** to apply your changes.

Tools Menu

About the Tools Menu

The **Tools** menu is a user-configurable menu that allows you to access personal tools from the PurifyPlus for Linux graphical user interface (GUI). You can customize the Tools menu to meet your own requirements.

Custom tools can be applied to a selection of nodes in the Project Explorer. Selected nodes can be sent as a parameter to a user-defined tool application. A series of macro variables is available to pass parameters on to your tool's command line.

See the section **GUI Macro Variables** in the **Rational PurifyPlus for Linux Reference Manual** for detailed information about using the macro command language.

Using the Tools Menu

To use a user-defined tool:

1. Select an icon from the **Project Explorer** pane.

2. Click the **Tools** menu and select the tool you want to use.

To add a new tool to the Tools menu:

1. From the **Tools** menu, select **Toolbox...**

2. To create an entirely new tool, click **Add...** If you want to copy from an existing tool, select the existing tool, click **Copy** and click **Edit...**

3. Edit the tool in the **Tool Edit** box.

4. Click **OK** and **Close**.

To edit a user-defined tool:

1. From the **Tools** menu, select **Toolbox...**

2. Select the tool that you want to modify and click **Edit...**

3. Edit the tool in the **Tool Edit** box.

4.   Click **OK** and **Close**.

To remove a tool from the Tools menu:

1.   From the **Tools** menu, select **Toolbox...**

2.   Select an existing tool from the tool list.

3.   Click **Remove** and **Close**.

Tool Configuration

The **Tool Configuration** dialog allows you to configure a new or existing tool.

In the **Tools** menu, each tool appears as a submenu item, or **Name**, with one or several associated actions or **Captions**.

Identification

In this tab, you describe how the tool will appear in the **Tools** menu.

* Enter the **Name** of the tool submenu as it will appear in the Tools menu and a **Comment** that is displayed in the lower section of the Toolbox dialog box.

* Select **Change Management System** if the tool is used to send and retrieve from a change management system. When **Change Management System** is selected, **Check In** and **Check Out** actions are automatically added to the Action tab (see below) and a **Change Management System** toolbar is activated.

* Clear the **Add to Tools menu** checkbox if you do not want the tool to be added to the Tools menu.

* Select Send messages to custom tab in the Output Window if you want to view the tool's text output in the Output Window.

* Use the **Icon** button to attach a custom icon to the tool that will appear in the **Tools** menu. Icons must be either **.xpm** or **.png** graphic files and have a size of 22x22 pixels.

Actions

This tab allows you to describe one or several actions for the tool.

* The **Actions** list displays the list of actions associated with the tool. If **Change Management System** is selected on the **Identification** tab, **Check In** and **Check Out** tool commands will listed here. These cannot be renamed or removed.

* **Menu text** is the name of the action that will appear in the **Tools** submenu.

- • **Command** is a shell command line that will be executed when the tool action is selected from **Tools** menu. Command lines can include toolbox macro variables and functions.

Click **OK** to validate any changes made to the Tool Edit dialog box.

To add a new action:

1. Enter a **Caption** and a **Command**.

2. Click **Add**.

To remove an action from the list:

1. Select an action in the **Actions** list.

2. Click **Remove**.

To modify an action:

1. Select an action in the **Actions** list.

2. Make any changes in the **Caption** or **Command** lines.

3. Click **Modify**.

Test Process Monitor

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** Currently only the Code Coverage feature provides data for the Test Process Monitor. You can, however, build your own files with the **tpmadd** command-line feature. See the **Reference Manual** for further information.

Changing Curve Properties

The **Curve Properties** menu allows you to change the way a particular graph is displayed.

To change the curve color:

1. Right-click a curve.

2. From the pop-up menu, select **Change Curve Color**.

3. Use the **Color Palette** to select a new color, and click **OK**.

To hide a curve:

1. Right-click a curve.

2. From the pop-up menu, select **Hide Curve**.

To set a maximum value:

Changing the maximum displayed value for a curve actually changes the scale at which it is displayed. For instance, when a curve only reaches 100, there is no point in displaying it at on a scale of 1000, unless you want to compare it with another curve that uses that scale.

1. Right-click a curve.

2. From the pop-up menu, select **Set Max Value**.

3. Enter the scale value, and click **OK**.

    Note Setting a maximum value lower than the actual maximum value of a curve can result in erratic results.

To display a scale:

For any curve, you can display a scale on the right or left-hand side of the graph. When you display a new scale, it replaces any previously displayed one.

1. Right-click a curve.

2. From the pop-up menu, select **Right Scale** or **Left Scale**.

Custom Curves

In some cases, you may want to remove certain figures from a chart to make it more relevant. The custom curves capability allows you to alter the chart by selecting the records that you want to include.

Note Using the custom curves capability does not impact the actual database. If you remove a record from the chart by using the custom curves function, the actual record remains in the database and may impact other figures.

Custom curves create a new metric, using the name of the base metric, with a Custom prefix.

To create a custom curve:

1.	Make sure a user is selected in the **Report Explorer** pane. If not, select a user.

2.	From the **Project** menu, select **Test Process Monitor** and **Custom Curves**.

3.	In the **Custom Curves** dialog box, select a metric and the start and end date of your chart.

4.	The record list displays all the records contained in the database of that metric. Select the records that you want to use for your custom curve. Clear the records that you do not want to use.

5.	Click **OK**. A new metric is created.

To change a custom curve:

1.	From the **Project** menu, select **Test Process Monitor** and **Custom Curves**.

2.	In the **Custom Curves** dialog box, select the Custom metric that you want to modify.

3.	Select the records that you want to use for your custom curve. Clear the records that you do not want to use.

4.	Click **OK**.

Event Markers

Use event markers to identify milestones or special events within your Test Process Monitor chart. An event marker is identified by the date of the event and a marker label.

Event markers appear as bold vertical lines in a Test Process Monitor chart.

To create an event marker:

1.	Right-click the location where you want to put the chart

2.	From the pop-up menu, select **Event Properties** and **New Event**.

3.	Enter the date of the event, and a marker label, and click **OK**.

To remove an event marker:

1.	Right-click the event marker that you want to hide.

2.	From the pop-up menu, select **Delete Event**.

To hide a specific event marker:

Hiding a marker does not remove it. You can still make the marker reappear.

1.	Right-click the event marker that you want to hide.

2. From the pop-up menu, select **Hide Event**.

To hide or show all event markers:

1. In the **Test Process Monitor** toolbar, click the **Events** button to hide all event markers.

2. Click again to show all hidden event markers.

Setting the Time Scale

The Scale capability defines the period that you want to view in the **Test Process Monitor** window. This option allows you to select an annual, monthly or daily view, as well as a user-definable time period.

To set the time scale:

1. Select a user in the **Report Explorer** pane.

2. From the **Project** menu, select **Test Process Monitor**, **Scale** and the desired time scale.

3. If you chose **Customize**, enter the start and end date of the period that you want to monitor, and click **OK**.

Test Process Monitor Toolbar

The Test Process Monitor (TPM) toolbar is useful for navigating through TPM charts.

These buttons are available when a TPM window is open:

- The **Clear** button removes all curves from the chart.

- The **Hide Event** button hides the displayed event markers.

- The **Floating Schedule** button toggles the automatic location of new curves.

To hide or show a toolbar:

1. Right-click an empty area within the toolbar.

2. Select and clear those toolbars you want to display or hide.

Adding a Metric

Metrics generated Code Coverage or other tools are directly available through the Test Process Monitor. Each metric file contains one or several fields.

To open a metric database a metric chart:

1. From the **Project** menu, select **Test Process Monitor** and either **Project** or **Current Workspace**. **Current Workspace** applies to the user of the current workspace. **Project** applies to all workspace users in the project.

2. If a new metric database is detected, you need to provide a name for the metric, as well as a label for each field of the database.

3. In the **Report Explorer**, select a user.

4. From the **Project** menu, select **Test Process Monitor**, the metric and the field that you want to display.

You can add as many curves as you want to the chart.

To hide a curve:

1. Right-click a curve.

2. From the pop-up menu, select **Hide Curve**.

UML/SD Viewer

About the UML/SD Viewer

The UML/SD Viewer renders sequence diagram reports as specified by the UML standard.

UML sequence diagram can be produced directly via the execution of the SCI-instruction application when using the Runtime Tracing feature.

Navigating through UML Sequence Diagram

There are several ways of moving around the UML sequence diagrams displayed by the UML/SD Viewer:

- **Navigation Panel:** Click and drag the **Navigation** button in the lower right corner of the **UML/SD Viewer** window to scroll through a miniature navigation pane representing the entire UML sequence diagram.

- **Free scroll:** Press the **Control** key and the left mouse button simultaneously. This displays a compass icon, allowing you to scroll the UML sequence diagram in all direction by the moving the mouse.

- **Report Explorer:** The Report Explorer is automatically activated when the UML/SD Viewer is activated. The Report Explorer offers a hierarchical view of instances. Click an item in the **Report Explorer** to locate and select the corresponding UML representation in the main **UML/SD Viewer** window.

Time Stamping

The UML/SD Viewer displays time stamping information on the left of the UML sequence diagram. Time stamps are based on the execution time of the application on the target.

You can change the display format of time stamp information in the UML/SD Viewer Preferences.

The following time format codes are available:

- **%n** - nanoseconds

- **%u** - microseconds

- **%m** - milliseconds

- **%s** - seconds

- **%M** - minutes

- **%H** - hours

These codes are replaced by the actual number. For example, if the time elapsed is 12ms, then the format **%mms** would result in the printed value **12ms**. If the number 0 follows the % symbol but precedes the format code, then 0 values are printed to the viewer - otherwise, 0 values are not printed. For example, if the time elapsed is 10ns, and the selected format code is **%0mms %nns**, then the time stamp would read **0ms 10ns** .

> **Note** To change the format code you must press the Enter key immediately after selecting/entering the new code. Simply pressing the **OK** button on the **Preferences** window will not update the time stamp format code.

Coverage Bar

In C, C++ and Java, the coverage bar provides an estimation of code coverage.

> **Note** The coverage bar is unrelated to the Code Coverage feature. For detailed code coverage reports, use the dedicated Code Coverage feature.

When using the Runtime Tracing feature, the UML/SD Viewer can display an extra column on the left of the UML/SD Viewer window to indicate code coverage simultaneously with UML sequence diagram messages.

The UML/SD Viewer code coverage bar is merely an indication of the ratio of *encountered* versus *declared* function or method entries and potential exceptions since the beginning of the sequence diagram.

If new declarations occur during the execution the graph is recalculated, therefore the coverage bar always displays a increasing coverage rate.

To activate or disable coverage tracing with a Java application:

1. Before building the node-under-analysis, open the Memory Profiling settings box.

2. Set **Coverage Tracing** to **Yes** or **No** to respectively activate or disable code coverage tracing for the selected node.

3. Click **OK** to override the default settings of the node

To hide the coverage bar:

1. Right-click inside the **UML/SD Viewer** window.

2. From the pop-up menu, select **Hide Coverage**.

To show the coverage bar:

1. Right-click inside the **UML/SD Viewer** window.

2. From the pop-up menu, select **Show Coverage**.

Memory Usage Bar

When using the Runtime Tracing feature on a Java application, the UML/SD Viewer can display an extra bar on the left of the UML/SD Viewer window to indicate total memory usage for each sequence diagram message event.

The memory usage bar indicates how much memory has been allocated by the application and is still in use or not garbage collected.

In parallel to the UML sequence diagram, the graph bar represents the allocated memory against the highest amount of memory allocated during the execution of the application.

This ratio is calculated by subtracting the amount of free memory from the total amount of memory used by the application. The total amount of memory is subject to change during the execution and therefore the graph is recalculated whenever the largest amount of allocated memory increases.

A tooltip displays the actual memory usage in bytes.

To activate or disable coverage tracing with a Java application:

1. Before building the node-under-analysis, open the **Memory Profiling** settings box.

2. Set **Coverage Tracing** to **Yes** or **No** to respectively activate or disable coverage tracing for the selected node.

3. Click **OK** to override the default settings of the node

To hide the memory usage bar:

1. Right-click inside the **UML/SD Viewer** window.

2. From the pop-up menu, select **Hide Memory Usage**.

To show the memory usage bar:

1. Right-click inside the **UML/SD Viewer** window.

2. From the pop-up menu, select **Show Memory Usage**.

Thread Bar

When using the Runtime Tracing feature on C, C++ and Java code, the UML/SD Viewer can display an extra column on the left of its window to indicate the active thread during each UML sequence diagram event.

Each thread is displayed as a different colored zone. A tooltip displays the name of the thread.

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.

Step-by-Step mode

When tracing large applications, it may be useful to slow down the display of the UML sequence diagram. You can do this by using the Step-by-Step mode.

To activate Step-by-Step mode:

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.

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**.

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**.

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**.

Editing Trigger or Filter Events

Use the Event Editor to create or modify event triggers or filters for UML sequence diagrams:

- **Filters:** Specified events are hidden or shown in the UML sequence diagram.

- **Start triggers:** The UML/SD Viewer starts displaying the sequence diagram when a specified event is encountered. If no event matches the output of the application, the diagram will appear blank.

- **Stop triggers:** The UML/SD Viewer stops displaying the sequence diagram when a specified event is encountered.

Events can be related to messages, instances, notes, synchronizations, actions or loops.

To define an event or filter:

1.   Specify a name for the event.

2.   Select the type of UML element you want to define for the event and select **Activate**. Several types of elements can be activated for a single filter or trigger event.

3.   Click **More** or **Fewer** to add or remove line to the event criteria.

4.   From the drop-down criteria box, select a criteria for the filter, and an argument.

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 ![icon] 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 UML Sequence Diagram

The UML/SD Viewer has an extensive search facility that allows users to locate specific UML sequence diagram elements by searching for a text string.

To search for a text string inside the UML/SD Viewer:

1. Click inside a **UML/SD Viewer** window to activate it.
2. 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.

UML/SD Viewer Toolbar

The UML/SD Viewer toolbar provides shortcut buttons to commands related to viewing UML sequence diagrams.

UML/SD Viewer commands are only available when a UML sequence diagram is open.

- The **Filter** button allows you to define a sequence diagram filter.

- The **Trigger** button sets sequence diagram triggers.

The following buttons are only available when using the Step-by-Step mode.

- The **Step** button moves the UML/SD Viewer to the next selected event.

- The **Select** button allows you to select the type of event to trace.

- The **Continue** button draws everything to the end of the trace diagram.

- The **Restart** button restarts Step-by Step mode.

- The **Pause** button pauses the On-the-Fly display mode. The application continues to run.

The TDF file selector is only available when using the Split TDF File feature.

- Click the button to select a **.tdf** dynamic trace file from the list.

- Click the and buttons to select the previous or next file in the list.

To hide or show a toolbar:

1. Right-click an empty area within the toolbar.

2. Select and clear those toolbars you want to display or hide.

3. Click **OK**.

UML/SD Viewer Preferences

The **Preferences** dialog box allows you to change the appearance of the UML Sequence Diagram reports.

To choose UML sequence diagram preferences:

1. Select the **UML/SD Viewer** node:

- **Background:** This allows you to choose a background color for the UML sequence diagram.

- **Panel:** This allows you to choose a background color for panels in the UML sequence diagram.

- **Panel Background:** This allows you to choose a background color for selected panels.

- Coverage Bar: This allows you to choose a background color for the coverage bar.

- Memory Usage: This allows you to choose a background color for the memory usage bar.

- **Print Page header:** Select this option to print a page header.

- **Print Page footer:** Select this option to print a page footer.

- **Display Page Breaks:** When this option is selected, the UML/SD Viewer displays horizontal and vertical dash lines representing the page size for printing.

- **Show tooltip in UML/SD Viewer:** Use this option to hide or show the information tooltip in the UML/SD Viewer.

- **Time Stamp Format:** Use the editable box to select the format in which time stamps are displayed in the UML/SD Viewer. See Time Stamping.

2. Expand the **UML/SD Viewer** node, and select **Styles** or **Styles System Test**:

- **Styles:** This list allows you to select one or several styles that you want to change. To change several styles at the same time, you can perform multiple selections in the style list.

- **Font:** This allows you to change the font type and size for the selected style.

- **Text Color:** This allows you to change the foreground and background colors for the selected style. This opens a standard color palette.

- **Text Attributes:** This allows you to set the selected style to Bold, Italic, Underlined or Dashed.

3. Click **OK** to apply your changes.

Configurations and Settings

Two major concepts of PurifyPlus for Linux 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:

125

- You can create a Configuration for each programming language or compiler involved in your project.

- 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 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.

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

The actual settings available for each node depend on the type of node and the language of the selected Configuration.

General Settings

The General settings are part Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Host Configuration

The Host Configuration area lets you override any information about the machine on which the Target Deployment Port is to be compiled.

- **Hostname:** The *hostname* of the machine. By default this is the local host.

- **Address:** The IP address of the host. For the local host, use 127.0.0.1.

- **Socket Uploader Port:** The default value is 7777.

- **Target Deployment Port:** This allows you to change the Target Deployment Port for the selected nodes. Child nodes will use the default Configuration Settings from this Target Deployment Port, such as compilation flags.

Source File Information

The Source File Information settings are only available on the project node as they apply to how PurifyPlus for Linux extracts source file information and dependency files to be displayed in the **Asset Browser** view of the **Project Explorer**. These setting apply to the entire project and cannot be overridden at the node level.

- **Directories for include files:** Specifies a list of include directories for the file tagging facility.

- **Get struct definition like a class:** Extracts *struct* definitions and display them as classes in the **Asset Browser**.

Directories

- **Build:** Specify an optional working directory for the Target Deployment Port. This is where the generated application will be executed on the target host.

- **Temporary:** Enter the location for any temporary files created during the Build process

- **Report:** Specify the directory where test results are created.

- **Java Main Class (for Java only):** Specifies the name of the main class for Java programs.

Target Deployment Port

The Target Deployment Port (TDP) Settings allow you to override the TDP used for a particular node in the current Configuration. By default, the TDP used is that of the current Configuration.

- **Name:** Displays the name of the TDP.

- **Directory name:** Specifies the TDP directory

- **Initial definition file:** Indicates the default **.ini** file in the TDP directory.

- **Source file language:** Sets the current language of the TDP.

- **Object file extension:** Specifies the default extension for object files produced with the current TDP.

- **Binary file extension:** Specifies the default extension for executable binaries produced with the current TDP.

- **Source file extension:** Specifies the default extension for source files used with the current TDP.

To edit the General settings for a node:

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

2. Select a node in the **Project Explorer** pane.

3. In the Configuration Settings list, expand **General**.

4. Select **Host Configuration, Directories** or **Target Deployment Port**.

5. When you have finished, click **OK** to validate the changes.

Build Settings

The **Build** settings are part of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Compiler Settings

- **Preprocessor options:** Specific compilation flags to be sent to the Test Compiler.

- **Compiler flags:** Extra flags to be sent to the compiler.

- **Preprocessor macro definitions:** Specify any macro definition that are to be sent to both the compiler preprocessor (if used) and the Test Compilers. Several generation conditions must be separated by a comma ',' with no space, as in the following example:
  `WIN32,DEBUG=1`

- **Directories for Include Files:** Click the **...** button to create or modify a list of directories for included files when the include statement is encountered in source code and test scripts. In the directory selection box, use the **Up** and **Down** buttons to indicate the order in which the directories are searched.

- **Boot Class Path (for Java only):** Click the **...** button to create or modify the Boot Class Path parameter for the JVM.

- **Class Path (for Java only):** Click the **...** button to create or modify the Class Path parameter for the JVM.

Linker Settings

This area contains parameters to be sent to the linker during the build of the current node.

- **Link Flags:** Flags to be sent to the linker.

- **Additional objects or libraries:** A list of object libraries to be linked to the generated executable. Enter the command line option as required by your linker. Please refer to the documentation provided with your development tool for the exact syntax.

- **Directories for Libraries:** Click the ... button to create or modify a list of directories for library link files. In the directory selection box, use the Up and Down buttons to indicate the order in which the directories are searched.

Build Target Deployment Port Settings

This area relates to the parameters of the Target Deployment Port on which is based the Configuration:

- **Measure time used by:** Selects between a real-time **Operating system clock** or a **Process or task clock** for time measurement, if both options are available in the current Target Deployment Port. Otherwise, this setting is ignored.

- **Maximum on-target buffer size:** This sets the size of the I/O buffer. A smaller I/O buffer can save memory when resources are limited. A larger buffer improves performance.
  The default setting for the I/O buffer is **1024** bytes.

- **Multi-threads:** This box, when selected, protects Target Deployment Port global variables against concurrent access when you are working in a multi-threaded environment such as Posix, Solaris or Windows. This can cause an increase in size of the Target Port as-well-as an impact on performance, therefore select this option only when necessary.

- **Maximum number of threads:** When the multi-thread option is enabled, this setting sets the maximum number threads that can be run at the same time by the application.

- **Run Garbage Collector at exit (for Java only):** This setting runs the JVM garbage collection when the application terminates.

To edit the Build settings for a node:

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

2. Select a node in the **Project Explorer** pane.

3. In the **Configuration Settings** list, expand **Build**.

4. Select **Compiler**, **Linker** or **Target Deployment Port**.

5. When you have finished, click **OK** to validate the changes.

External Command Settings

The External Command settings are part of the Configuration Settings dialog box, which allows you to configure settings for each node.

Use the External Command setting to set a command line for External Command nodes. An External Command is a command line that can be included at any point in your workspace. External Commands can contain PurifyPlus for Linux GUI macro variables, making them context-sensitive. See the **GUI Macro Variables** chapter in the **Reference Manual**.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

To edit the External Command settings for one or several nodes:

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

2. Select a node in the **Project Explorer** pane.

3. Select the **External Command** node and enter a **Command** line.

4. When you have finished, click **OK** to validate the changes.

Static Metric Settings

The Static Metric settings are part of the Configuration Settings dialog box, which allows you to configure settings for each node.

Use the Static Metric settings to change any project settings related to the calculation of static metrics.

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.

*   **One level metrics:** By default, **.met** static metric files are produced for source files as well as all dependency files that are found by the Source Code Parser. Set **One level metrics** to **Yes** to restrict the calculation of static metrics only to the source files displayed in the Project Browser.

*   **Analyzed directories:** This setting allows you to restrict the generation of **.met** metric files only to files which are located in the specified directories.

To edit the Static Metrics settings for one or several nodes:

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

2. Select a node in the **Project Explorer** pane.

3. Select the **Static Metrics** node.

4. When you have finished, click **OK** to validate the changes.

Runtime Analysis Settings

General Runtime Analysis Settings

The General Runtime Analysis settings are part of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Snapshot Settings

In some case, such as with applications that never terminate or when working with timing or memory-sensitive targets, you might need to dump traces at specifics points in your code.

- **On Function Entry:** Allows you to specify a list of function names, from your source code, that will dump traces at the beginning of the function.

- **On Function Return:** Allows you to specify a list of function names, from your source code, that will dump traces at the end of the function.

- **On Function Call:** Allows you to specify a list of function names, from your source code, that will dump traces before the function is called.

For each tab, click the **...** button to open the function name selection box. Use the **Add** and **Remove** buttons to create a list of function names.

See Generating SCI Dumps for more information.

Selective Instrumentation

By default, runtime analysis features instrument all components of source code under analysis.

The Selective Instrumentation settings allow you to more finely define which units (classes and functions) you want to instrument and trace.

- **Units excluded from instrumentation:** Click the **...** button to access a list of units (classes and functions) that can be excluded from the instrumentation process. Click a unit to select or clear a unit. Use the **Select File** and **Clear File** buttons to select and clear all units from a source file.

- Files excluded from instrumentation: Click the ... button and use the Add and Remove buttons to select the files to be excluded.

- Instrument inline methods: Extends instrumentation to inline methods.

- **Instrument included methods or functions:** Extends instrumentation to included methods or functions.

- Directories excluded from instrumentation: Click the ... button and use the Add, Remove buttons to select the files to be excluded.

Static File Storage

Depending on the runtime analysis feature, the product generates **.tsf** or **.fdc** temporary static data files during source code instrumentation of the application under analysis.

- **Code Coverage Static File Storage (.fdc):** These settings apply to Code Coverage **.fdc** static trace files:

- **Build directory:** Select this option to use the current directory for all generated files.

- **Other directory:** Select this option to define a specific directory.

- **Source directory:** Select this option to use the same directory as the source under analysis.

- **Use single temporary file (.fdc):** By default, Code Coverage produces one **.fdc** file for each instrumented source file. Select this option to use a single **.fdc** file for all instrumented source files, and specify its location.

- **FDC Directory:** When using the **Use single temporary file (.fdc)** option in the previous setting, specify a location for the **.fdc** file.

- **Memory Profiling, Performance Profiling, and Runtime Tracing Storage:** This setting applies to Memory Profiling, Performance Profiling and Runtime Tracing **.tsf** static trace files.

- **Build directory:** Select this option to use the current directory for all generated files.

- **Other directory:** Select this option to define a specific directory.

- **Source directory:** Select this option to use the same directory as the source under analysis.

- **Use single temporary file (.tsf):** By default, Memory Profiling, Performance Profiling and Runtime Tracing produces one **.tsf** file for each instrumented source file. Select this option to use a single **.tsf** file for all instrumented source files, and specify its location.

- **TSF Directory:** When using the **Use single temporary file (.tsf)** option in the previous setting, specify a location for the **.tsf** file.

Miscellaneous Options

- **Label Instrumented Files:** Select this option to add an identification header to files generated by the Instrumentor, including the command line used to generate the file, the version of the product, date and operating system information.

- **Full template instantiation:** By default unused methods are ignored by the Instrumentor. Set this option to **Yes** to analyze all template methods, even if they are not used.

- **Additional Instrumentor Options**: This setting allows you to add command line options for the Instrumentor. Normally, this line should be left blank.

To edit the General Runtime Analysis settings for a node:

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

2.   Select a node in the **Project Explorer** pane.

3.   In the Configuration Settings list, expand **Runtime Analysis** and **General**.

4.   Select **Snapshot**, **Selective Instrumentation**, **Static File Storage** or **Miscellaneous**.

5.   When you have finished, click **OK** to validate the changes.

Memory Profiling Settings

The **Memory Profiling Instrumentation Control** and **Memory Profiling Misc. Options** settings are part of the **Runtime Analysis** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Instrumentation Control

*   **File in use (FIU):** When the application exits, this option reports any files left open.

*   **Memory in use (MIU):** When the application exits, this option reports allocated memory that is still referenced.

*   **Signal (SIG):** This option indicates the signal number received by the application forcing it to exit.

*   Freeing Freed Memory (FFM) and Late Detect Free Memory Write (FMWL): Select Display Message to activate detection of these errors.

*   **Free queue length (blocks)** specifies the number of memory blocks that are kept free.

*   **Free queue size (Kbytes)** specifies the total buffer size for *free queue* blocks. See Freeing Freed Memory (FFM) and Late Detect Free Memory Write (FMWL).

*   **Display Detect Array Bounds Write (ABWL):** Select **Yes** to activate detection of this error.

*   **Red zone length (bytes)** specifies the number of bytes added by Memory Profiling around the memory range for bounds detection.

*   **Number of functions:** specifies the maximum number of functions reported from the end of the CPU call stack. The default value is 6.

Misc. Options

- **Trace File Name (.tpf):** This box allows you to specify a filename for the generated **.tpf** trace file.

- **Global variables to exclude from observation (for Java only):** This box specifies a list of global variables that are not to be inspected for memory leaks. This option can be useful to save time and instrumentation overhead on trusted code. Use the **Add** and **Remove** buttons to add and remove global variables.

JVMPI

- **Object hashtable size:** Specifies the size of hashtables for objects where *<size>* must be 64, 256, 1024 or 4096 values.

- **Class hashtable size:** Specifies the size of hashtables for classes where *<size>* must be 64, 256, 1024 or 4096 values.

- **Take a Snapshot:** You can select one of the following options:

- **On method entry or return or dump snapshot button**: Uses a specified method to perform snapshot or the GUI snapshot button as specified in the Enable dump Snapshot button setting.

- **After each Garbage Collection**: Takes a snapshot each time the JVM garbage collector runs.

- Enable dump snapshot button and Delay Snapshot until next Garbage Collection: Specify the trigger method.

- **Host name used by dump Snapshot button:** Use this option to specify a hostname for the JVMPI Agent to communicate with the GUI.

- **Port Number used by dump Snapshot button:** Use this option to specify a port number for the JVMPI Agent to communicate with the GUI.

- **TPF file name (.tpf):** Specifies the name of the Memory Profiling trace dump file produced by the JVMPI Agent.

- **TXF file name (.txf):** Specifies the name of the Memory Profiling static trace dump file.

- **Display only listed methods:** Use the Add and Remove buttons to add and remove methods to be listed by the Java Memory Profiling report.

- **Collect referenced objects:** Sets the filter to be used with the Java Memory Profiling Report.

- **Display only listed packages:** Use this setting to filter out of the report the packages that do not match the specified full package name (package and class).

- **Display only listed classes:** Use this setting to filter out of the report the classes that do not match the specified full classes.

- **Display call stack for listed methods:** Use this setting to list the methods for which the call stack is to be displayed in the Java Memory Profiling report.

To edit the Memory Profiling settings for one or several nodes:

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

2. Select a node in the **Project Explorer** pane.

3. Select the **Runtime Analysis** node and the **Memory Profiling** node.

4. Select either **Instrumentation Control**, **Misc. Options** or **JVMPI**.

5. When you have finished, click **OK** to validate the changes.

Performance Profiling Settings

The **Performance Profiling** settings are part of the **Runtime Analysis** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Trace File Name (.tqf): This box allows you to specify a filename for the generated **.tqf** trace file for Performance Profiling.

To edit the Performance Profiling settings for one or several nodes:

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

2. Select a node in the **Project Explorer** pane.

3. Select the **Runtime Analysis** node and the **Performance Profiling** node.

4. When you have finished, click **OK** to validate the changes.

Code Coverage Settings

The Code Coverage Instrumentation Control settings are part of the **Runtime Analysis** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Instrumentation Control Settings

You can use the Coverage Type settings to declare various types of coverage.

- Coverage Level Functions or Methods: select between function Entries, With exits, or None.

- **Coverage Level Calls:** select **Yes** or **No** to toggle call code coverage for C only.

- **Coverage Level Blocks:** select the desired block code coverage method.

- **Coverage Level Conditions:** select condition code coverage for C only.

Please see Selecting Coverage Types for details on using each coverage type with each language.

You can combine, enable, or disable any of these coverage types before running the application node. All coverage types selected for instrumentation can be filtered out in the Code Coverage Viewer.

- **Mode:** This setting specifies the Instrumentation Modes to be used by Code Coverage.

- **Default (Optimized for Code Size and Speed):** This setting uses one byte per branch to indicate branch coverage.

- **Compact (Optimized for Memory):** This setting uses one bit per branch. This method saves target memory but uses more CPU time.

- **Report Hit Count:** This adds information about the number of times each branch was executed. This method uses one integer per branch.

Selective Code Coverage Instrumentation

- **Ternary coverage (for C and C++ only):** For C and C++, when this option is selected, Code Coverage is extended to ternary expressions as statement blocks.

- **Functions to Exclude from Calls Code Coverage:** Specifies a list of functions to be excluded from the call coverage instrumentation type, such as **printf** or **fopen**. Use the **Add**, **Remove** buttons to tell Code Coverage the functions to be excluded.

Miscellaneous Options

- **Trace File Name (.tio):** this allows you to specify a path and filename for the **.tio** dynamic coverage trace file.

- **Compute deprecated metrics:** This setting is for compatibility with third party tools designed for previous versions of the product. Set this to **No** in most cases.

- **User comment:** This adds a comment to the Code Coverage Report. This can be useful for identifying reports produced under different Configurations. To view the comment, click the a magnifying glass symbol that is displayed at the top of your source code in the Code Coverage Viewer.

To change the Code Coverage Instrumentation Control setting for an application or test node.

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

2. Select a node in the **Project Explorer** pane.

3. Select the **Runtime Observation** node, and the **Coverage** node.

4. Select **Instrumentation Control**.

5. When you have finished, click **OK** to validate the changes.

Runtime Tracing Settings

The Runtime Tracing Control settings are part of the **Runtime Analysis** node of the Configuration Settings dialog box, which allows you to configure settings for each node.

By default, the settings of each node are inherited from those of the parent node. When you override the settings of a parent node, changes are propagated to all child nodes within the same Configuration. Overridden fields are displayed in bold.

Instrumentation Control

- **Trace File Name (.tdf):** This allows you to force a filename and path for the dynamic **.tdf** file. By default, the **.tdf** carries the name of the application node.

- **Functions called within a return expression are sequenced:** For C only. With this option, the UML/SD Viewer displays calls located in return expressions as if they were executed sequentially and not in a nested manner.

- **Collapse unnamed classes and structures:** For C++ only. With this option, unnamed *structs* and *unions* are not instrumented.

- **Display class template instantiation in a note:** For C++ only. With this option, the UML/SD Viewer will not display notes for template instances for each template class instance.

Trace Control

- **Split Trace File Enable:** See Splitting trace files for more information on this setting.

- Maximum Size (Kbytes): This specifies the maximum size for a split .tdf file. When this size is reached, a new split .tdf file is created.

- **File name prefix:** By default, split files are named as **att_**<*number*>**.tdf**, where <*number*> is a 4-digit sequence number. This setting allows you to replace the **att_** prefix with the prefix of your choice.

- **Automatic loop detection enable:** Loop detection simplifies UML sequence diagrams by summarizing repeating traces into a loop symbol. Loops are an extension to the UML sequence diagram standard and are not supported by UML.

- Options (Reserved for future use): This setting allows you to add command line options. Normally, this line should be left blank.

- **Display largest call stack length:** When selected, the Target Deployment Port records the highest level attained by the call stack during the trace. This information is displayed at the end of the UML Sequence Diagram in the UML/SD Viewer as **Maximum Calling Level Reached**.

Target Deployment Port Settings

These settings allow you to set compilation flags that define how the Runtime Tracing feature interacts with the Target Deployment Port. These are general settings for the Target Deployment Port.

- **Disable on-the-fly mode:** When selected, this setting stops on-the-fly updating of the dynamic **.tdf** file. This option is primarily for Target Deployment Ports that use **printf** output.

- **Trace Buffer Enable** and **Partial Trace Flush Enable:** Please see Trace Item Buffer and Partial Trace Flush for more information about these settings.

- Maximum number of recorded Trace elements before buffer flush

- When receiving user signal: No Action, Flush Call Stack, Trace On/Off

- **Record and display Time Stamp:** This setting adds time stamp information to each element in the UML sequence diagram generated by Runtime Tracing.

- Record and display Heap Size (for Java only): This setting enables the Heap Size Bar in the UML sequence diagram produced by Runtime Tracing.

- **Record and display Thread Info (for Java only):** This setting enables the Thread Bar in the UML sequence diagram produced by Runtime Tracing.

To edit the Runtime Tracing settings for one or several nodes:

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

2. Select a node in the **Project Explorer** pane.

3. Select the **Runtime Analysis** node and the **Runtime Tracing** node.

4. Select **Instrumentation Control**, **Trace Control** or **Target Deployment Port Settings**.

5. When you have finished, click **OK** to validate the changes.

Selecting Configurations

Although a project can use multiple Configurations, as well as multiple TDPs, there must always be at least one active Configuration.

The active Configuration affects build options, individual node settings and even wizard behavior. You can switch from one Configuration to another at any time, except during build activity, when the green LED flashes in the Build toolbar.

To switch Configurations:

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**.

Working with Projects

Working with Projects

The project is your main work area in PurifyPlus for Linux, 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.

Understanding Projects

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 PurifyPlus for Linux.

A project is materialized as a directory in your file system, which contains everything you need to test and analyze your code:

- Source code

- Analysis result files

In the PurifyPlus for Linux 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.

- **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.

Sub-Projects

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. 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.

Results Node

By default, each application 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.

Creating a Group

The Group node is designed to contain several application nodes. This allows you to organize workspace by grouping applications together.

This also allows you to build and run a specific group of application nodes without running the entire workspace.

To create a group node:

1. In the **Project Explorer**, right-click the workspace node or right-click any application node.

2. From the pop-up menu, select **Add Child** and **Group**.

3. In the **New Group** box, enter the name of the group.

4. Click **OK**.

Manually Creating an Application Node

Application nodes are the main building blocks of your workspace. An application node typically contains the source files required to build the application.

The preferred method to create an application node is to use the Activity Wizard, which guides you through the entire creation process.

However, if you are re-using existing components, you might want to create an empty application node and manually add its components to the workspace.

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.

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

143

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 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 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.

Importing Files from a Makefile

The PurifyPlus for Linux 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 PurifyPlus for Linux.

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.

To import files from a makefile:

1. Close any open projects.

2. From the **File** menu, select **Import** and **Import 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**.

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, PurifyPlus for Linux 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 PurifyPlus for Linux 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 **Build** menu, select **Refresh Asset Browser**, or press the **F9** key.

To activate or de-activate the automatic refresh:

With the automatic file tagging option, files are automatically refreshed whenever a file is loaded into the workspace or selected in the Project Explorer.

1. From the **Edit** menu, select **Preferences**.

2. Select the **Project** preferences node.

3. Select or clear the **Activate file tagging** option, and then click **OK**.

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.

Deleting a Node

Removing nodes from a project does not actually delete the files, but merely removes them from the Project Explorer's representation.

To delete a node from the Project Explorer:

1. Select one or several nodes that you want to delete.

2. From the **Edit** menu, select **Delete** or press the **Delete** key.

Renaming a Node

Renaming a node in the Project Explorer involves modifying the properties of the node.

To change the name of a node:

- 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**.

Viewing File Properties

You can obtain and change file or node properties by opening the **Properties** window.

To view file properties:

1. Right-click a file in the **Project Explorer**.

2. Select **Properties...** from the pop-up menu.

Excluding a Node from a Build

In some cases, you might want to temporarily exclude one or several nodes from the build process. This can be done directly in the Project Explorer, as described below, or through the Properties window.

> **Note** If you exclude a node that contains child nodes, such as an application node, a group or even a project, none of the contents of the node are executed.

In the Project Explorer, excluded nodes are displayed with a ✖ symbol. Normally built nodes are displayed with a ◔ symbol.

To exclude a node from the build:

1. In the **Project Explorer**, select the node that you want to exclude from the build.

2. In the **Properties** window set the Build property to No.

To cancel the exclusion of a node:

1. In the **Project Explorer**, select the node that you want to exclude from the build.

2. In the **Properties** window set the **Build** property to **No**.

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**.

Adding Files to the Project

The Project Explorer centralizes all Project files in a unique location. For PurifyPlus for Linux to access and analyze source files, they must be accessible from the Project Explorer.

Files are automatically added when you use the Activity Wizard.

To add files to the Project Explorer:

1. In the **Project Explorer**, select the **Object Browser** tab

2. In the **Sort Method** box, select **By Files**.

3. From the **Project** menu, select **Add to Current Project** and **New File...**

4. This opens the file selector. In the file **Type** box, select the type of files that are to be added.

5. Locate and select one or several files to be added, and click **Open**.

The selected files will appear under the Source sections of the Project Explorer.

If you have the Automatic source browsing option enabled, your source files will be analyzed, making their components directly accessible in the Project Explorer.

Selecting Build Options

The PurifyPlus for Linux GUI allows you to specify the items that will be performed during a build.

The **Stages** section contains the compilation options. In most cases, you will need to select the **All** option to ensure the test is up to date.

The **Runtime Analysis** section allows you to enable debugging and Runtime Analysis features.

To select build options:

1.   From the **Build** menu, select **Options**.

2.   Select the Runtime Analysis features (Memory Profiling, Performance Profiling, Code Coverage and Runtime Tracing) and build options to use them on the current node.

Building and Running a Node

You build and execute workspace nodes by using the **Build** ▷button on the Build toolbar. The build process compiles, links, deploys, executes, and then retrieves results. However, you first have to specify the various build options.

You can use the **Build** ▷command to execute any application node, as well as a single specific source file, a group node or even the whole project.

> **Note**   When you run the **Build** ▷command, all open files are saved. This means that any unsaved changes will actually be taken into account for the build.

Before building a node:

1.   Select the correct Configuration for your target in the build toolbar.

2.   Exclude any temporarily unwanted nodes from the build.

3.   Select the build options for the test.

4.   If necessary, clean up files left by any previous executions by clicking the **Clean** ⟳button.

To build and execute the node:

1.   From the Build toolbar, click the **Build** ▷button.

2.   During run-time, the Build Clock indicates the execution time and the green LED flashes. The Project Explorer displays a check mark next to each item to mark progression of the build process.

3.   When the build process is finished, you can view the related test reports.

To stop the execution:

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.

Cleaning Up Generated Files

In some cases, you might want to delete any files created by a build execution, such as to perform the build process in a clean environment or when you are running short of disk space.

Use the **Clean All Generated Files** command to do this.

To clean your workspace:

1.   From the Build toolbar, click the **Clean All Generated Files** button.

Creating a Source File Folder

The Project Explorer Asset Browser provides a convenient way of viewing the source files in your project.

To make this even more convenient, you can create custom folders to accommodate any file types. This makes navigation through your source files even easier.

Note The Asset Browser provides a virtual navigation interface. The actual files do not change location. Use the **Properties Window** to view the actual file locations.

To create a custom folder:

1.   In the **Asset Browser**, select the **By 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.

Opening a Report

Because of the links between the various views of the GUI, there are many ways of opening a runtime analysis report in PurifyPlus for Linux. The most common ones are described here.

> **Note**   Some reports require opening several files. For instance, when manually opening a UML sequence diagram, you must select at the complete set of **.tsf** files as well as the **.tdf** file generated at the same time. A mismatch in **.tsf** and **.tdf** files would result in erroneous tracing of the UML sequence diagram.

To open a report from the Project Explorer:

1. Execute your test with the **Build** command.

2. Right-click the application node.

3. From the pop-up menu, select **View Report** and then the appropriate report.

Note   Reports cannot be viewed before the application 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 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.

Troubleshooting a Project

When executing a node for the first time in PurifyPlus for Linux, 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.

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.

Editing Preferences

Rational PurifyPlus for Linux has many **Preference** settings that allow you to configure various components of the graphical user interface.

To edit product preferences:

1.   From the **Edit** menu, select **Preferences**.

2.   In the tree-view, select the component that you want to configure.

3.   Make any changes to the preferences.

4.   Click **OK**.

Project Preferences

The **Project Preferences** dialog box lets you set parameters for the PurifyPlus for Linux project.

In the **Preferences** dialog box, select **Project** to change the project preferences.

- **Automatic file tagging:** Select this option to activate the Project Explorer's automatic parsing mode, in which all source code and script components are

automatically listed. If disabled, you will have to manually refresh the File View each time you modify the structure of a file.

- **Calculate static metrics:** Select this option to ensure that static metrics are recalculated whenever a file is added, modified or refreshed in the Project Explorer window.

- **Verbose output:** Select this option to prompt the PurifyPlus for Linux GUI to report detailed information to the Output Window during execution. Use this option to debug any compilation issues.

- **Show report nodes in Project Explorer:** Select this option to display runtime analysis reports in the Project Browser once they have been successfully generated. Report nodes appear inside their application nodes.

Connection Preferences

The **Preferences** dialog box allows you to customize PurifyPlus for Linux.

The **Connections** node of the **Preferences** dialog box lets you set the network parameters for the graphical user interface.

1. In the **Preferences** dialog box, select the **General** node and **Connections**.

   - **Allow remote connections:** This allows external commands and tools to send messages to the GUI over a network. For example, this enables the Runtime Tracing on-the-fly capability on remote hosts.

   - For information only, the **Current TCP/IP port** is automatically selected by GUI.

2. Click **OK** to apply your changes.

Activity Wizards

The Start Page provides with a full set of activity wizards to help you get started with a new project or activity.

To start a new activity wizard:

1. From the **Start Page**, click **New Activities**

2. Select the activity of your choice.

New Project Wizard

When PurifyPlus for Linux start, the Start Page offers to either open an existing project or create a new project. The New Project wizard creates a brand new project.

To create a new project:

1.   From the **Start Page**, select **New Project**.

2.   In the **Project Name**, enter a name for the project.

3.   In the **Location** box, change the default directory if necessary and click **Next** to continue.

4.   Select one or several Target Deployment Ports for the new project.

      The Wizard creates a Configuration based on each selected Target Deployment Port. Later, when working with the project, any changes are made to the Configuration Settings, not to the Target Deployment Port itself.

5.   Click the **Set as Active** button to set the current TDP. The active port is the default Configuration to be used in your project.

6.   Click **Finish**.

Once your project has been created, the wizard opens the **Activities** page.

Runtime Analysis Wizard

The Runtime Analysis Wizard helps you create a new application node in the Project Explorer. Basically, an application node represents the build of your C, C++ 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 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 features to the application under analysis.

Command Line Interface

Rational PurifyPlus for Linux 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 acquire runtime analysis data without conflicting with your native compiler and linker.

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

- **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 complete syntax and command line reference for each of these command line tools is covered in the **Rational PurifyPlus for Linux Reference Manual**.

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.

To run a specific node from a command line:

1.   Set up and configure your project in the GUI.

2.   Save your project and close the GUI.

3.   Type the following command :
     studio -r <node>.{[.<node>]} <project_file>

     where *<node>* is the node to be executed and *<project>* is the **.rtp** project file.

The *<node>* hierarchy must be specified from the highest node in the project (excluding the actual project node) to the target node to be executed, with periods ('**.**') separating each item:
     <node>{[.<node>]}

Example

The following command opens the **project.rtp** project in the GUI, and runs the *app2* application node, located in *group1* of the sub-project *subproject1*:
     studio -r subproject1.group1.app2 project.rtp

Command Line Runtime Analysis for C and C++

The runtime analysis features for C and C++ include:

*   Memory Profiling

*   Performance Profiling

*   Code Coverage

*   Runtime Tracing

These features use Source Code Insertion (SCI) technology. When analyzing C and C++ code, the easiest way to implement SCI features from the command line is to use the C and C++ Instrumentation Launcher.

The Instrumentation Launcher is designed to fit directly into your compilation sequence; simply add the **attolcc** command in front of your usual compilation or link command line.

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

To perform runtime analysis on C or C++ source code:

1. First, set up the necessary environment variables. See Setting Environment Variables.

2. Edit your usual makefile with the following command line:
    ```
    attolcc [-options] [--settings] -- <compiler command line>
    ```

Where *<compiler command line>* is the command that you usually invoke to build your application.

For example:
```
attolcc -- cc -I../include -o appli appli.c bibli.c -lm
attolcc -TRACE -- cc -I../include -o appli appli.c bibli.c -lm
```

Please refer to the **Instrumentation Launcher** section of the **Reference Manual** for information on attolcc options and settings, or type **attolcc --help** on the command line.

3. After execution of your application, in order to process SCI dump information (i.e. the runtime analysis results), you need to separate the single output file into separate, feature-specific, result files. See Splitting the SCI Dump File.

4. Finally, launch the **Graphical User Interface** to view the test reports. See the **Graphical User Interface** command line section in the **Rational PurifyPlus for Linux Reference Manual**.

Command Line Runtime Analysis for Java

The runtime analysis features for Java covered in this section include:

*   Performance Profiling

*   Code Coverage

*   Runtime Tracing

These features use Source Code Insertion (SCI) technology. Memory Profiling for Java relies on JVMPI instead of SCI technology. Please refer to the JVMPI Agent section of the **Reference Manual**.

The easiest way to implement SCI from the command line is to use the Java Instrumentation Launcher: *javic*. The product provides two methods for use of javic:

- **Java Instrumentation Launcher:** designed to fit directly into your compilation sequence; simply add the **javic** command in front of your usual compilation or link command line

- **Java Instrumentation Launcher for Ant:** this integrates javic with the Apache Jakarta Ant utility

For details of command line usage and option syntax, see the **Reference Manual**.

To perform runtime analysis on Java source code:

1. First, set up the necessary environment variables. See Setting Environment Variables.

2. Edit your usual makefile by adding the Java Instrumentation Launcher to the command line:
   ```
   javic [-options] -- <compiler command line>
   ```

Where *<compiler command line>* is the command that you usually invoke to build your application.

Please refer to the Instrumentation Launcher section of the **Reference Manual** for information on the options and settings.

3. After execution, to obtain the final test results, as well as any SCI dump information, you need to separate the output file into separate result files. See Splitting the SCI Dump File.

4. Finally, launch the **Graphical User Interface** to view the test reports. See the **Graphical User Interface** command line section in the **Reference Manual**..

Command Line Examples

This section describes two basic examples of using PurifyPlus for Linux Runtime Analysis features through the Command Line Interface:

- Runtime Analysis using the Instrumentation Launcher

- Calculating Metrics using the Metrics

  Note   These examples are for UNIX platforms only.

Runtime Analysis using the Instrumentation Launcher

This example demonstrates using Runtime Analysis features 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 PurifyPlus for Linux environnment, type the following command:
   ```
   . <install_dir>/PurifyPlusForLinux.2003.06.00/ppluslinuxinit.sh
   ```

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

   Refer to the **Rational PurifyPlus for Linux 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 PurifyPlus for Linux GUI to view the results, type the following command:
    ```
    studio  `find . -name "*.fdc"` `find . -name "*.tsf"` *.tio *.tpf
    *.tqf *.tdf
    ```

Calculating Metrics

This example demonstrates producing static metrics of the source code contained in the **BaseStation_C** example project with the **metcc** command line. This example is provided in the **examples** directory of PurifyPlus for Linux.

To calculate metrics using the command line:

1. Copy the BaseStation_C project to your working directory, type the following command:
    ```
    cp -r
    <install_dir>/releases/PurifyPlusForLinux.2003.06.00/examples/Bas
    eStation_C/src
    ```

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

2. To set up the PurifyPlus for Linux environnment, type the following command:
    ```
    . <install_dir>/PurifyPlusForLinux.2003.06.00/ppluslinuxinit.sh
    ```

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

    Refer to the **Rational PurifyPlus for Linux Installation Guide** for information about setting up and launching the product.

3. To calculate metrics without a makefile, type the following command:
    ```
    metcc -- gcc -c UmtsCode.c
    ```

4. To calculate metrics of all the files in your makefile, type the following command:
    ```
    make CC="metcc -- gcc" CPP="metcc -- gcc"
    ```

5. To start the PurifyPlus for Linux GUI to view the results, type the following command:
    ```
    studio *.met
    ```

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.

Library Paths

UNIX platforms require the following additional environment variable:

- **On Solaris and Linux platforms:** LD_LIBRARY_PATH points to $TESTRTDIR/lib/*<platform>*/*<os>*

- **On HP-UX platforms:** SH_LIB points to **$TESTRTDIR/lib/**_<platform>_/_<os>_

- **On AIX platforms:** LIB_PATH points to **$TESTRTDIR/lib/**_<platform>_/_<os>_

_wh_ere _<platform>_ is the hardware platform and _<os>_ is the current operating system.

Example

The following example shows how to set these variables for PurifyPlus for Linux 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 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:

4.   #define ATL_WITHOUT_STUDIO

5.   Make any necessary changes by adjusting the corresponding macros in the file.

The **product_model.h** file is self-documented, and you can adjust every macro to one of the values listed. Each macro is set to a default value, so you can keep everything unchanged if you don't know how to set them.

> **Note**   Pay attention to correctly set the macros starting with **USE_**, because these macros set which features of PurifyPlus for Linux 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.

Take note of the directory where this file is stored, you will need it in order to compile the generated or instrumented source files.

Instrumenting and Compiling the Source Code

The runtime analysis features (Memory Profiling, Performance Profiling, Code Coverage and Runtime Tracing) all use SCI instrumentation technology to insert analysis and SCI dump routines into your source code.

Requirements

Before compiling an SCI-instrumented source file, you must make sure that:

*   A working C, C++ or Java compiler is installed on your system

*   If you compile on a target different from the host where the generated file has been produced, the instrumented file must have been produced using option **-NOPATH**, and the sub-directory lib of the selected Target Deployment Port directory must be copied onto the target.

There are two alternatives to instrument and compile your source code:

*   Using the Instrumentation Launcher in your standard makefile

*   Using the Instrumentor and Compiler separately.

Instrumentation Launcher

The Instrumentation Launcher replaces your actual compiler command in your makefiles. This launcher transparently takes care of source code preprocessing, instrumentation and compiling.

See the command line information for the Instrumentation Launcher in the **Reference Manual**.

Instrumentation and Compilation

Alternatively, you can use the actual Instrumentor command line tools to instrument the source files.

See the command line information for each Instrumentor in the **Reference Manual**.

If you are compiling on a different target, you must copy the TDP /lib directory over to that target.

Add to the include search path the **/lib** sub-directory that you have copied onto the target. In C and C++, use the **-I** compiler option. In Java, add the directory to the **CLASSPATH**.

After this, simply compile the instrumented source file with your compiler.

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++ linker is installed on your system

- You have compiled every source file, including any instrumented source files, of your application under test

- You have compiled the Target Deployment Port library.

Linking

You just have to add the Target Deployment Port library object to the object files linked together.

Running the Application

Once you have produced a binary tester or instrumented application, you want to run it in order to obtain test or SCI analysis information.

By default, the generated SCI dump file is named **atlout.spt**.

To run the test application binary:

1. Check that the current directory is correct, relatively to the previously specified trace file, if the trace files was specified with a relative path.

2. Run the binary. When the application terminates, the trace file should be available.

Splitting the Trace Dump File

When you use several features together, the executable produces a multiplexed trace file, containing several outputs targeting different features from PurifyPlus for Linux . By default, the trace file is named **atlout.spt**.

Requirements

In most cases, you must split the **atlout.spt** trace file into several files for use with each particular Report Generator or the product GUI.

To do this, you must have a working *perl* interpreter. You can use the *perl* interpreter provided with the product in the **/bin** directory.

To split the trace file:

1. Use the **atlsplit** tool supplied in the **/bin** directory of PurifyPlus for Linux :
   ```
   atlsplit atlout.spt
   ```

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

- **.rio test result files:** process with a Report Generator

- **.tio Code Coverage report files:** view with Code Coverage Viewer

- **.tdf dynamic trace files:** view with UML/SD Viewer

- **.tpf Memory Profiling report files:** view with Memory Profiling Viewer

- **.tqf Performance Profiling report files:** view with Performance Profiling Viewer

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.

Compiler reports that **options.h** is missing	Ensure that you have correctly prepared the **options.h** file, and that this file is located in a directory that is searched by your compiler (this is usually specified with **-I** or **/I** option on the compiler command line).
Compiler reports that **TP.h** file is missing	If you are compiling on a target different from the host where the generated file has been produced, double-check the above specific requirements to compilation on a different target.
	If the test compiler and C/C++ compiler are executed on the same machine, ensure you have not used the **-NOPATH** option on the test compiler command line, and that the **ATLTGT** environment variable was correctly set while the test script

	compiler was executed.
Linkage fails because of undefined references	Ensure you have successfully compiled the Target Deployment Port library object, and have included it in your linked files
	Ensure you have correctly configured the **products.h** options file.
	If you are using a test feature, ensure that you are linking both source under test and additional files. You may also want to add some stubs in your **.ptu** or **.otd** test script.
	Ensure the options set in **options.h** (if required) are coherent with the options set in **products.h.**
Errors are reported through *#error* directives	You may have selected a combination of options in **products.h** which is incompatible. The error messages help you to locate the inconsistencies.

Working with Other Development Tools

Rational PurifyPlus for Linux 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 Rational ClearCase

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, PurifyPlus for Linux 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 PurifyPlus for Linux 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 Rational ClearQuest

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 PurifyPlus for Linux you can directly submit a report from a runtime analysis report.

To submit a ClearQuest report from PurifyPlus for Linux:

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.

CMS Preferences

The **Preferences** dialog box allows you to change the settings related to the integration of the product with Rational ClearCase or other configuration management software (CMS).

To change configuration management settings:

1. Select the **CMS** node.

*   **Repository directory:** Use this box to specify the location of the vault directory for the CMS tool.

*   **Selected Configuration Management System:** Use this box to select Rational ClearCase or a different CMS tool. Before setting this option, make sure that the CMS system has been configured in Tools menu.

2. Click **OK** to apply your changes.

ClearQuest Preferences

The **Preferences** dialog box allows you to specify the location of the Rational ClearQuest database.

Please refer to the documentation provided with ClearQuest for more information.

To change ClearQuest preferences:

1. Select the **ClearQuest** node.

*   **Schema Repository:** Use this box to select the schema repository you want to use.

- **Database:** Use this box to enter the location of the ClearQuest database.

- **User Name** and **Password:** Enter the user information provided by your ClearQuest administrator.

2. Click **OK** to apply your changes.

Customizing Configuration Management

Out of the box, the product offers configuration management support for Rational ClearCase, but the product can be configured to use most other Configuration Management Software (CMS) that uses a vault and local repository architecture and that offers a command line interface.

To configure the product to work with your version control software:

1. Add a new CMS tool to the Toolbox with the command lines for checking files into and out of the configuration management software. This activates the **Check In** and **Check Out** commands in the Project Explorer and the ClearCase Toolbar.

2. Set up version control repository in CMS Preferences.

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

D

G

H

I

R