Rational® PurifyPlus for Linux

ONLINE TUTORIAL

VERSION: 2002 RELEASE 2 - SR1

support@rational.com
http://www.rational.com

IMPORTANT NOTICE

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

Part Number: 800-025997-000

Version: 2002 Release 2 - SR1

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

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

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

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

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

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

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

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

Online Tutorial Contents

Overview ... **3**

C and C++ Track ... 3

Preparation ... 3

Goals of the Tutorial ... 4

Java Track ... 5

Preparation ... 5

JDK Installation ... 6

Goals of the Tutorial ... 7

Runtime Analysis with Rational PurifyPlus for Linux ... **9**

C and C++ Track ... 9

Runtime Analysis for C and C++ ... 9

Runtime Analysis with Rational PurifyPlus for Linux ... 9

Memory Profiling ... 9
Performance Profiling ... 10
Code Coverage Analysis ... 10
Runtime Tracing ... 11

Runtime Analysis Exercises ... 12

Exercise One ... 13
Exercise Two ... 18
Exercise Three ... 30

Conclusion ... 38

Command Line Usage of PurifyPlus for Linux ... 38
Conclusion - with a Word about Process ... 39

Java Track ... 40

Runtime Analysis for Java ... 40

Runtime Analysis with Rational PurifyPlus for Linux ... 40

Memory Profiling ... 40
Performance Profiling ... 41

Code Coverage Analysis 42
Runtime Tracing 42
Runtime Analysis Exercises 43
Exercise One 45
Exercise Two 51
Exercise Three 64
Conclusion 72
Command Line Usage of PurifyPlus for Linux 72
Conclusion - with a Word about Process 76

Conclusion 79

Proactive Debugging 79
Questions? 80

Technical Support 81

Overview

1

C and C++ Track

Preparation

In this tutorial, you will be working with a simulated mobile phone and UMTS Base Station. Source files for the base station (the mobile phone executable is provided) are located within the PurifyPlus for Linux installation folder, in the folder \examples\BaseStation_C\src. (If you don't have write permission to the installation location of PurifyPlus for Linux, then you will need to copy the examples folder - and its contents - to a new location. Otherwise, you will be unable to perform any part of the Tutorial that creates/modifies files.)

UMTS - Universal Mobile Telecommunications System - is a Third Generation (3G) mobile technology that will enable 2Mbit/s streaming not only of voice and data, but also of audio and visual content. A UMTS base station is a switching network device enabling the communication of multiple UMTS-enabled mobile phones.

The mobile phone simulator consists of both a Graphical User Interface (GUI) as well as of internal logic. The GUI is constructed from OS-independent graphical C++ classes; the logic within the simulator is constructed from OS-independent C and C++ code.

The mobile phone executable is located within the PurifyPlus for Linux installation folder, in the folder **\examples\BaseStation_C\MobilePhone**. The name of the executable depends on your Linux distribution:

- Linux SuSE: MobilePhone.Linux
- Linux RedHat: MobilePhone.Linux_redhat

(A launcher shell script - **MobilePhone.sh** - is provided as well.)

The UMTS base station is fully operational, constructed from OS-independent C and C++ code. You are provided with both the source code and an executable for the base station. The UMTS base station executable is located within the PurifyPlus for Linux installation folder, in the folder **\examples\BaseStation_C**. The name of the executable depends on your Linux distribution:

- Linux SuSE: BaseStation.Linux
- Linux RedHat: BaseStation.Linux_redhat

(A launcher shell script - **BaseStation.sh** - is provided as well.)

Since efforts are always being made to update or improve the tutorial - as well as PurifyPlus for Linux itself - a customer-only webpage has been created. This page, accessible via the PurifyPlus for Linux **Help** menu item **Latest News and Updates for Users**, will contain news, patches, etc. for current users. Feel free to check this page for updates before usage of this tutorial.

Goals of the Tutorial

The UMTS base station has been pre-loaded with errors; your responsibility, during the tutorial, will be to uncover:

- a memory leak
- a performance bottleneck
- a logic error in C code
- a logic error in C++ code

In addition, you will:

- analyze the code coverage achieved via UMTS base station interaction
- improve your understanding of the code via runtime tracing

To accomplish the above, you will manipulate the UMTS base station through manual interaction with a mobile phone simulator.

To continue this tutorial, follow the C and C++ track in the lesson Runtime Analysis with PurifyPlus for Linux.

Java Track

Preparation

In this tutorial, you will be working with a simulated mobile phone and UMTS base station. Source files for the base station (the mobile phone executable is provided) are located within the PurifyPlus for Linux installation folder, in the folder **\examples\BaseStation_Java\src**. (If you don't have write permission to the installation location of PurifyPlus for Linux, then you will need to copy the **examples** folder - and its contents - to a new location. Otherwise, you will be unable to perform any part of the Tutorial that creates/modifies files.)

UMTS - Universal Mobile Telecommunications System - is a Third Generation (3G) mobile technology that will enable 2Mbit/s streaming not only of voice and data, but also of audio and visual content. A UMTS base station is a switching network device enabling the communication of multiple UMTS-enabled mobile phones.

The mobile phone simulator consists of both a Graphical User Interface (GUI) as well as of internal logic. The GUI is constructed from OS-independent graphical C++ classes; the logic within the simulator is constructed from OS-independent J2SE-compliant code.

The mobile phone executable is located within the PurifyPlus for Linux installation folder, in the folder **\examples\BaseStation_C\MobilePhone** - that is, the executable is not located in the BaseStation_Java folder. The name of the executable depends on your Linux distribution:

- Linux SuSE: MobilePhone.Linux
- Linux RedHat: MobilePhone.Linux_redhat

(A launcher shell script - **MobilePhone.sh** - is provided as well.)

The UMTS base station is fully operational, constructed from OS-independent J2SE-compliant code. You are provided with both the source code and an executable for the base station. The UMTS base station executable is located within the PurifyPlus for Linux installation folder, in the folder **\examples\BaseStation_Java**. The name of the executable depends on your Linux distribution:

- Linux SuSE: BaseStation.Linux
- Linux RedHat: BaseStation.Linux_redhat

(A launcher shell script - **BaseStation.sh** - is provided as well.)

Since efforts are always being made to update or improve the tutorial - as well as PurifyPlus for Linux itself - a customer-only webpage has been created. This page, accessible via the PurifyPlus for Linux **Help** menu item **Latest News and Updates for Users**, will contain news, patches, etc. for current users. Feel free to check this page for updates before usage of this tutorial.

JDK Installation

Performance of the demo assumes access to the J2SE 1.3.1 or 1.4.0 SDK.

If neither J2SE distribution is currently installed on your machine, you can freely download them as follows:

NOTE: The following are the recommended J2SE distributions. Technically, any SDK that is 100% J2SE 1.3.1 or 1.4.0 compliant can be used with PurifyPlus for Linux. However, only the following distributions have been verified as supported.

For J2SE 1.3.1 on Linux (both RedHat and SuSE):

1. Go to http://java.sun.com/j2se/1.3/download.html
2. Select the SDK download link for "Linux GNUZIP Tar shell script"
3. Download and install the SDK onto your machine

For J2SE 1.4.0 on Linux (both RedHat and SuSE)

1. Go to http://java.sun.com/j2se/1.4/download.html
2. Select the SDK download link for "Linux GNUZIP Tar shell script"
3. Download and install the SDK onto your machine

Goals of the Tutorial

The UMTS base station has been pre-loaded with errors; your responsibility, during the tutorial, will be to uncover:

- poor memory management
- a performance bottleneck
- a logic error in Java code

In addition, you will:

- analyze the code coverage achieved via UMTS base station interaction
- improve your understanding of the code via runtime tracing

To accomplish the above, you will manipulate the UMTS base station

through manual interaction with a mobile phone simulator.

To continue this tutorial, follow the Java track in the next lesson: Runtime Analysis with Rational PurifyPlus for Linux.

Runtime Analysis with Rational PurifyPlus for Linux

2

C and C++ Track

Runtime Analysis for C and C++

Runtime analysis refers to the ability of PurifyPlus for Linux to monitor an application as it executes. There are a variety of advantages to be gained from this monitoring:

- memory profiling
- performance profiling
- code coverage analysis
- runtime tracing

Runtime Analysis with Rational PurifyPlus for Linux

Memory Profiling

Dynamically working with system memory can be quite a complicated affair. If you're not careful, your code might either:

- Fail to free memory - referred to as a memory leak
- Mistakenly reference non-allocated memory - referred to as an array bounds read or array bounds write

A memory leak detection utility monitors an application as it executes, keeping an eye on memory usage to ensure the above problems don't occur. If they do occur, the detection utility points out the sequence of events

leading up to the poor usage of memory, helping you deduce the cause of the error and thereby repair your code.

This function is provided in Rational PurifyPlus for Linux by the memory profiling feature for the C and C++ languages.

Performance Profiling

Optimal performance is, needless to say, crucial for business-critical systems. Measuring performance can be quite difficult, however, particularly when it comes to determining the specific functional bottlenecks in your system.

That's where performance profiling monitors come in. These tools watch your application as it executes, measuring statistics such as:

- How often a function is called
- How long it takes for that function to execute
- Which functions are the bottlenecks of your application

With this information you can optimize your code, ensuring all constraints placed upon your system are accommodated.

This function is provided in Rational PurifyPlus for Linux by the performance profiling feature for the C and C++ languages.

Code Coverage Analysis

One of the greatest difficulties a developer experiences is a failure to determine the portions of code that have gone untested. For many systems, failure is not an option, so every part of an application must be thoroughly tested to ensure there is no unhandled scenario or dead code.

In addition, product managers need a concrete measurement to determine where the team is in the development cycle - in particular, how much more

testing needs to be done. A decreasing number of defects does not necessarily mean the product is ready; it might simply mean the portions of code that have been tested appear to be ready.

Code coverage measurement tools monitor your running application, flagging every line of code as it executes. Advanced tools - such as PurifyPlus for Linux - are also able to differentiate different types of execution, such as whether or not a **do-while** loop executed 0 times, 1 time, or 2 or more times. These advanced measurements are crucial for software systems upon which customer confidence relies.

This function is provided in Rational PurifyPlus for Linux by the code coverage feature for the C and C++ languages.

Runtime Tracing

As all developers quickly learn, intentions don't necessarily translate into reality. There can often be a vast difference between what you want to happen and what actually happens as your application executes.

This problem becomes more severe when the code is inherited. Yes, you could try to piece things together yourself, but system complexity might just undercut your efforts at understanding the code.

And what about multi-threaded applications? If you've ever encountered race conditions or deadlocks, you know how difficult it can be to uncover the source of the problem.

This is where runtime tracing monitors come in. These utilities graphically display the sequence of function or method calls in your running application - as well as the active threads - illustrating through pictures what is actually happening. With this information, unexpected exceptions can be easily traced back to their source, complex procedures can be distilled to their essence, threading conflicts can be resolved and inherited code can jump off the page and display its inherent logic.

This function, using the industry standard Unified Modeling Language for its graphical display, is provided in Rational PurifyPlus for Linux by the runtime tracing feature for the C and C++ languages.

Runtime Analysis Exercises

The following exercises will walk you through a typical use case involving the four runtime analysis features of PurifyPlus for Linux to which you have just been introduced. Pay close attention not only to the capabilities of these features but also to how they are used. The better you understand these features, the more quickly you will be able to adopt them within your own development process.

Reminders before you begin:

1. If you have never run this tutorial before, make sure your machine has a temporary folder in which you can store the test project you will be creating. For the tutorial, it is assumed that the test project will be stored in a folder called **tmp**

2. 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. Is the GNU compiler located during installation of PurifyPlus for Linux the only GNU compiler installed on your machine? If so, skip the rest of this section. If not - or if you are not sure - then you should ensure the proper compiler will be accessible by performing the following:

- From a command prompt, execute the shell script 'ConfigureGcc.sh'
- Follow the prompts until the proper GNU compiler is located

This shell script is located in the folder:

- Red Hat - (install dir)/releases/PurifyPlusForLinux.v2002R2/bin/intel/linux_redhat
- SuSE - (install dir)/releases/PurifyPlusForLinux.v2002R2/bin/intel/linux_suse

This shell script depends on a properly configured environment. If, for some reason, your environment is not properly configured (indicated by the interactive dialog):

- Set your current directory to the applicable shell script folder mentioned above
- Execute the environment configuration shell script - '.ppluslinuxinit.sh'

3. If you have run this tutorial before, don't forget to undo the source file edits you made the last time you ran through it. The following files are modified during the tutorial:

- **PhoneNumber.cpp**
- **UmtsCode.c**
- **UmtsServer.cpp**

Exercise One

Introduction to Exercise One

In this exercise you will:

- create a new project in which the UMTS base station source code will be referenced

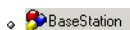
If you need a refresher about the application you will be using during this tutorial, look here; otherwise, please proceed.

Creating a Project

Typically, there is a one-to-one relationship between your current development project and a PurifyPlus for Linux project. Although your development project may consist of more than one application, these applications often possess a common theme. Use the PurifyPlus for Linux project to enforce that theme.

1. To start Rational PurifyPlus for Linux, type **studio** on the command line.
2. Select the **Get Started** link on the left-hand side of the PurifyPlus for Linux user interface (UI). Two links will appear on the right-hand side of the UI - one called **New Project** and one called **Open Project**. Select the **New Project** link. You should now see the **New Project Wizard**.
3. In the **Project Name** field, enter **BaseStation** (no spaces). In the **Location** field, select the ... button, browse to the folder in which you want the BaseStation project to be stored and then select it. (This Tutorial will assume that the project has been stored in the folder \usr\tmp) Click the **Next** button.
4. Select, from the list of supported GNU distributions, the one you intend to use.Note that GNU compilers support both C and C++.
5. Click the **Finish** button.

That's it. The project has been created - named **BaseStation** - and a project node by the same name appears on the Project Browser tab of the Project Explorer window on the right-hand side of the UI:



Creating a New Activity

Now that you have created a project, it is time to specify:

- your development project's source files

- the type of testing or runtime analysis activity you would like to perform first

1. Once a project has been created, the user is automatically brought to the **Activities** page. In this tutorial you are starting with a focus on the runtime analysis features, so select the **Runtime Analysis** link. This will bring up the **Runtime Analysis Wizard**.

2. In the window entitled **Application Files**, you must list all source files for your current development project. For this tutorial, you will directly select the source files. Select the **Add** button.

3. Browse to folder into which you have installed PurifyPlus for Linux and then access the folder **\examples\BaseStation_C\src**

4. Make sure **All C++ Files** in the **Files of Type** dropdown box is selected, then left-click-hold-and-drag over all of the eight C and C++ source files. Now click the **Open** button. You should see these eight files listed in the large listbox of the **Application Files** window. Click the **Next** button.

5. At this time, an analysis engine parses each source file - referred to as tagging. This process recognizes extracts the various classes and associated methods to simplify code browsing.

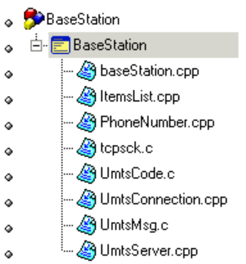
6. In the window entitled **Selective Instrumentation** you have the ability to select those functions/procedures/methods/classes that should not be instrumented for runtime analysis. Such selective instrumentation ensures that the instrumentation overhead is kept to a minimum. For this Tutorial, you will be monitoring everything, so simply click the **Next** button.

7. You have now reached the window entitled **Application Node Name**. Enter the name of the application node that will be created at the conclusion of the Runtime Analysis Wizard; since you will be monitoring execution of the UMTS base station, type the word **BaseStation** within the text field labeled **Name**.

8. Click the **Next** button.

9. You are now confronted with the Summary window. Everything should be in order, so click the **Finish** button.

The BaseStation application node has now been created. The Project Browser tab of the Project Explorer window should appear as follows:



Conclusion of Exercise One

Have a look at the right side of your screen. This is the Project Explorer window, and within it two tabs are visible.

The first - the **Project Browser** tab - contains a reference to all group, application and test nodes created for the active project. The project node, named **BaseStation**, contains an application node named **BaseStation**; the application node contains a list of all of the source files required to build the UMTS base station application. (Though the project and application nodes have the same name, this is not a requirement.)

The second tab - the **Asset Browser** tab - lets you browse all of your source and test files. If the selected **Sort Method** is **By Files**, you are presented with a file-by-file listing of test scripts, source code and source code dependents (such as header files). Note how each header file can be expanded to display every class, function, and method declaration, while each source file can be expanded to display every defined object and method or function. Double-clicking any test script/source file/header file node will open its contents within the PurifyPlus for Linux editor; double-clicking any class declaration or method definition node will open the

relevant source file/header file to the very line of code at which the definition/declaration occurs.

There are two other sort methods as well on the Asset Browser. The first, **By Objects**, let's you filter down to classes and methods, independent of the source files. The second, **By Packages**, is primarily applicable to Java packages.

You may have noticed along one of the toolbars at the top of the UI that the GNU compiler you selected in the New Project Wizard is listed in a dropdown box. In fact, this is not a reference to the compiler ; rather, it is a reference to a Configuration whose base compiler is the one you selected in the wizard. (Configurations are initially named after their base compiler, but this name can be changed.) Should you have multiple configurations for the same project, use this dropdown box to select the active Configuration for execution.

Finally, to the right of the Configuration dropdown list is the Build button ▷. This button is used to build your application for application nodes and the test harness for test nodes. The test harness consists of:

- source files needed to build the application of interest
- stubs
- a test driver

The downward-facing arrow associated with the Build button lets the user decide from which point the build process should initiate and what runtime analysis features should be used. The runtime analysis features do not have to be used at the same time; this Build options window provides a quick and simple method for deselecting undesired runtime analysis features immediately prior to execution of the build process.

Armed with this knowledge, proceed to Exercise Two.

Exercise Two

Introduction to Exercise Two

In this exercise you will:

- build and execute the UMTS base station application
- manually interact with the UMTS base station application
- view the runtime analysis reports derived from your interaction

Building and Executing the Application

1. When performing runtime analysis, your source code must be instrumented. Instrumentation, by default, is enabled for all four runtime analysis features - that is, for memory profiling, performance profiling, code coverage analysis and runtime tracing. All four features are turned on by default. In order to instrument, compile, link, and execute the UMTS base station application in preparation for runtime analysis, simply ensure the **BaseStation** application node is selected on the **Project Browser** tab of the Project Explorer window, and then click the build button ▷. Do so now.

   NOTE: More information about the source code insertion technology can be found in the **User Guide**, in the chapter **Product Overview->Source Code Insertion**.

2. Notice that in the Output Window at the bottom of the screen, on the **Build** tab, you can watch the preprocessing, instrumentation, compilation, and link phases of the build process as they occur. A double-click on an error listed within any of the Output Window tabs opens the relevant source code file to the appropriate line in the PurifyPlus for Linux Editor.

3. The build process has completed, and the UMTS base station is running, when the UML-based sequence diagram generated by the runtime

tracing feature appears. (More about this feature in a moment.)

4. Close the Project Explorer window on the right-hand side of the UI by clicking its ☒ button.

Notice how the graphically displayed data in the **Runtime Trace** viewer dynamically grows - this is because the UMTS base station is being actively monitored. The UMTS base station endlessly searches for mobile phones requesting registration; the Runtime Trace viewer reflects this endless loop. If you wish, use the pause button on the toolbar to stop the dynamic trace for a moment (the trace is still being recorded, just no longer displayed in real time). In addition, use the zoom buttons on the toolbar to get a better view of the graphical display (or right-click-hold within the Runtime Trace viewer and select the **Zoom In** or **Zoom Out** options). Undo the Pause when you're ready to proceed.

You'll look at the Runtime Trace viewer in more detail later. Of primary importance right now is interaction with the UMTS base station. You'll do this by using the mobile phone simulator mentioned earlier in the Overview section of this tutorial. Through this manual interaction you will expose memory leaks, performance bottlenecks, incomplete code coverage, and dynamic runtime sequencing.

Interacting with the Application

1. Start the mobile phone by running the provided mobile phone executable built for your operating system. The mobile phone executable is located within the PurifyPlus for Linux installation folder in the folder **\examples\BaseStation_C\MobilePhone**. The name of the executable is **MobilePhone.Linux.** (A launcher shell script - **MobilePhone.sh** - is provided as well.)

2. Click the mobile phone's On button ().

3. Wait for the mobile phone to connect to the UMTS base station (if you

watched the Runtime Trace viewer closely, you would have noticed a display of all the internal method calls of the UMTS base station that occur when a phone attempts to register). The current system time should appear in the mobile phone window when connection has been established.

4. Once connected, dial the phone number **5550000**, then press the button to send this number to the UMTS base station (again, try to see the Runtime Trace viewer update).

5. Unfortunately, the party you are dialing is on the line so you'll find the phone is busy. Shut off the simulator by closing the mobile phone window via the button in its upper right corner.

   The UMTS base station is designed to shut off when a registered phone goes off line. Not a great idea for the real world, but it serves the Tutorial's purposes well. Alternatively, you could have just used the Stop Build/Execute button located next to the Build button on the toolbar .

6. The UMTS base station has stopped running when the green execution light next to the execution timer - located beneath the Project Explorer window on the lower right-hand side of the UI - stops flashing . Wait for it to stop flashing.

   Everything that occurred at the code level in the UMTS base station was monitored by all four runtime analysis features. Once the UMTS base station stopped (i.e. once the instrumented application stopped), all runtime analysis information was written to user accessible reports that are directly linked to the UMTS base station source code. In order to look at these reports:

7. Reopen the Project Explorer window by selecting the menu item **View->Other Windows->Project Window**

8. In the Project Explorer window, on the Project Browser tab, double-click the **BaseStation** application node. All four runtime analysis reports will open. (Alternatively, right-click the **BaseStation** application node and select **View Report**->**All**.)

9. Close the Project Explorer window and the Output Window (at the bottom of the UI) to create room for the now-opened reports. You may also want to resize the left-hand window to gain additional room.

Runtime Analysis - Runtime Tracing

1. Select the **Runtime Trace** tab.

2. As you recall, the Runtime Trace viewer displayed all objects and all method calls involved in the execution of the UMTS base station code. Using the toolbar buttons, zoom out from the tracing diagram until you can see at least five vertical bars.

3. Make sure you are looking at the top of the runtime tracing diagram using the slider bar on the right.

What you are looking at is a sequence diagram of all events that occurred during the execution of your code. This sequence diagram uses a notation taken from the Unified Modeling Language, thus it can be correctly referred to as a UML-based sequence diagram.

The vertical lines are referred to as lifelines. Each lifeline represents either a C source file or a C++ object instance. The very first lifeline, represented by a stick figure, is considered the "world" - that is, the operating system. In this UMTS base station tracing diagram, the next lifeline to the right represents an object instance named **Obj0**, derived from the **UmtsServer** class.

Green lines are constructor calls, black lines are method calls, red lines are method returns, and blue lines are destructor calls. Hover the mouse over any method call to see the full text. Notice how every call and call return is time stamped.

Everything in the Runtime Trace viewer is hyperlinked to the monitored source code. For example, if you click on the **Obj0::UmtsServer** lifeline, the header file in which the UmtsServer class declaration appears is opened for you, the relevant section highlighted. (Close the source file by right-clicking the tab of the Text Editor and selecting **Close**.) All function calls can be left-clicked as well in order to view the source code. Look at the very top of the **Obj0::UmtsServer** lifeline. It's "birth" appears to consist of a **List()** constructor first, then a **UmtsServer()** constructor. Why a call to the List() constructor if the object is an instance of the UmtsServer class? Click on the **UmtsServer()** lifeline again - see how the UmtsServer() constructor inherits from the List() class? This is why the List() constructor is called first. Click the two constructor calls if you wish to pursue this matter further.

Notice how the window on the left-hand side of the user interface - called the **Report Window** - contains a reference to all classes and class instances. Double-clicking any object referenced in this window will jump you to its birth in the Runtime Trace viewer. This window can also be used to filter the runtime tracing diagram.

4. In the left-hand window, close the node labeled **NETWORKNODE.H** - notice how all objects derived from the NetworkNode class declared in this header file are reduced to a single lifeline.

5. Reopen the node labeled **NETWORKNODE.H**.

You've probably noticed the vertical graph with the green bar to the left of the Runtime Trace viewer. This is the **Coverage Bar**. It highlights, in synchronization with the trace diagram, the percentage of total code coverage achieved during execution of the monitored application. The Coverage Bar's caption states the percentage of code coverage achieved by the particular interaction presently displayed in the Runtime Trace viewer. Scroll down the trace diagram; note how code coverage gradually increases until a steady state is achieved. This steady state is achieved following the moment at which the mobile phone has

connected to the UMTS base station. Dialing the phone number increases code coverage a bit; shutting off the phone creates a last burst of code coverage up until the moment the UMTS base station is shut off. Can you locate where, on the trace diagram, the mobile phone simulator first connected to the UMTS base station? (The Coverage Bar can be toggled on and off using the right-click-hold menu within the Runtime Trace viewer.)

NOTE: If the C++ code in the UMTS base station spawned multiple threads, the Coverage Bar would be joined by the **Thread Bar**, a vertical graph highlighting the active thread at any given moment within the trace diagram. A double-click on this bar would open a threading window, detailing thread state changes throughout your application's execution. This thread monitoring feature is also available for the Java language.

NOTE: For Java only, the Coverage Bar would be accompanied by a **Memory Usage Bar**, a vertical graph showing the total amount of allocated memory at any given moment within the trace diagram. Such a diagram would be used to expose memory intensive parts of your program that may in fact be needless churn that slows down overall execution time. You could trigger garbage collection immediately prior to suspect moments within your application, using the Runtime Trace viewer to help you decide where the garbage collection should occur, to see if memory usage has become excessive.

Continue to look around the trace diagram. Can you locate the repetitive loop in which the UMTS base station looks for attempted mobile phone registration (it always starts with a call to the C function **tcpsck_data_ready**)? You can filter out this loop using a couple of methods. One is to simply hover the mouse over a method or function call you wish to filter, right-click-hold and select **Filter Message**. An alternative method would be to build your own filter. You will do both.

6. Hover the mouse over any call of the **tcpsck_data_ready** function, right-click-hold and select **Filter Message** - the function call should disappear from the entire trace.

7. Select the menu item **Runtime Trace->Filters** (you'll see the filter you just performed listed here) Click the **Import** button, browse to the installation folder and then the folder **\examples\BaseStation_C**, and then **Open** the filter file **filters.tft**

8. Left-click the checkbox next to the just imported filter named **BaseStation Phone Search Filter**.

9. Click the **OK** button.

   The loop has been removed.

   Not only can the runtime tracing feature capture standard function/method calls, but it can also capture thrown exceptions.

10. View the very bottom of the runtime tracing diagram using the slider bar.

    Do you see the icon for the catch statement - catch exception (you may have to drag the slider bar slightly upward; closing the NETWORKNODE.H node in the left-hand report window will also make things easier to see)? This **Catch Exception** statement is preceded by a diagonal **Throw Exception**. Why diagonal? Because when the exception was thrown, prior to executing the Catch statement, the **LostConnection** constructor and **UmtsMsg** destructor were called. Click various elements to view the source code involved in the thrown exception and thus decipher the sequence of events.

    This exception occurred by design, but it is clear how the runtime tracing feature, through the power of UML, would be extremely useful if you have:

- inherited old or foreign code

- unexpected exceptions
- questions about whether what you designed is occurring in practice

Further Work

Runtime Analysis- Memory Profiling

1. Select the **Memory Profile** tab.

The Memory Profile viewer displays a record of improper memory usage within the application of interest. First, block and byte memory use is summarized for you in a bar chart, immediately followed by a textual description to the same information. What you have is a record of:

- total number of blocks/bytes allocated for the entire run
- total number of non-freed blocks/bytes allocated for the entire run
- total number of blocks/bytes in use at any one time

If any memory errors were detected, or if any warnings are warranted, those comments are listed next. The Report Window on the left hand side of the screen gives you a quick look at the contents of the report - your manual interaction with the UMTS base station via the simulated mobile phone has resulted in the creation of **Test #1**. If you click an item in the Report Window, the memory profiling report will scroll to the proper location.

2. On the Report Window, left-click the **ABWL** error.

   Apparently, the memory profiling feature has detected a **Late Detect Array Bounds Write** (**ABWL**) - in other words, the UMTS base station code attempted to add data to an array element that does not exist. This error report is followed by the call stack, with the last function in the call stack listed first. Notice how each function is highlighted; clicking on the functions in the call stack will jump you to the relevant source code. Each source code file is highlighted at the line in which memory was

requested - in this particular case, some part of the UMTS base station code overwrote an array, thereby causing the ABWL error.

The **ABWL** is followed by one **File In Use** (**FIU**)warning and five **Memory Leak** (**MLK**)warnings. The **File In Use** warning references **<internal use>** - in other words, the file is being used by the memory profiling feature. As for the memory leaks - well it looks like you have some work to do here. Although it is conceivable the memory leak occurs by design (e.g. perhaps some clean-up code has not yet been written), assuredly the UMTS base station is not meant to have any.

Finally, the exit code is printed - look for the informational/warning note in the viewer starting with the words **Program exit code**. The memory profile report lists the exit code as a warning if it is of any value other than 0.

Notice how easily this information has been acquired; no work was required on your part. A real advantage is that memory leak detection can now be part of your regression test suite. Traditionally, if developers looked for memory leaks at all, it was done while using a debugger - a process that does not lend itself to automation and thus repeatability. The memory profiling feature lets you automate memory leak detection.

Further Work

Runtime Analysis - Performance Profiling

1. Select the Performance Profile tab.

The Performance Profile viewer displays the execution time for all functions or methods executing within the application of interest, thereby allowing the user to uncover potential bottlenecks. First, the three functions or methods requiring the most amount of time are displayed graphically in a pie chart (up to six functions will be displayed if each is individually responsible for more than 5% of total

execution time). This is then followed by a sortable list of every function or method, with timing measurements displayed.

Notice how the function **tcpsck_data_ready** was responsible for around 45% to 50% of the time spent processing information in the UMTS base station. By looking at the table, where times are listed in microseconds, we can see that this function's average execution time was between 1 to 2 seconds (it will vary somewhat based on your machine) and that it has no descendents - i.e. it never calls and then awaits the return of other functions or methods (which explains why the **Function** time matches the **F+D** time). Is this to be expected? If you wished, you could click on the function name in the table to jump to that function to see if its execution time can be reduced.

Each column can be used to sort the table - simply click on the column heading.

2. Click the column heading entitled **F+D Time**

It is probably no surprise that the **main()** procedure - combined with its descendents - takes the longest time to execute overall. Notice, though, that the **main()** procedure itself only takes around 300us to execute - so there doesn't appear to be any bottleneck here. The **main()** procedure spends its life waiting for the UMTS base station to exit.

As with the memory profiling feature, notice how easy it was to gather this information. Performance profiling can now also be part of your regression test suite.

Further Work

Runtime Analysis - Code Coverage Analysis

1. Select the **Code Coverage** tab.

And finally, here you have the code coverage analysis report. The code

coverage feature exposes the code coverage achieved either through manual interaction with the application of interest or via automated testing.

On the left hand side of the screen, in the Report Window, you see a reference to **Root** and then to all of the source and header files of the UMTS base station. **Root** is a global reference - that is, to overall coverage. For each individual source and header file, a small icon to the left indicates the level of coverage (green means covered, red means not covered).

In the Code Coverage viewer, on the **Source** tab, a graphical summary of total coverage is presented in a bar chart - that is, information related to **Root**. Five levels of code coverage are accessible when the source code is C++, and those five levels are represented here. (Four more levels of coverage are accessible when working with the C language - up to and including Multiple Conditions/Modified Conditions.) Notice how, on the toolbar, there is a reference to these five possible coverage levels F E B I L.

2. Deselect **Loops Code Coverage** L

   Notice how the bar chart is updated.

3. Reselect **Loops Code Coverage** L

4. In the Report Window to the left, select the **PhoneNumber.cpp** node.

   The **Source** tab now displays the source code located in the file **PhoneNumber.cpp**. This code is colored to reflect the level of coverage achieved. Green means the code was covered, red means the code was not covered.

5. In the Report Window, expand the **PhoneNumber.cpp** node and then select the **void PhoneNumber::clearNumber()** child node

   The **clearNumber()** function should now be visible on the **Source** tab.

Notice how its **for** instruction is colored orange and sitting on a dotted underline. This is because the **for** statement was only partially covered.

6. Click on the orange **for** keyword in the **clearNumber()** function

   As you can see, the **for** loop was only executed multiple times, not once or zero times. Why should you care? Well some certification agencies require that all three cases be covered for a **for** statement to be considered covered. If you don't care about this level of coverage, just deselect **Loops Code Coverage**:

7. On the toolbar, deselect **Loops Code Coverage** L.

   Now the **for** loop is green. If you would like to add a comment to your code indicating how this loop is not covered by typical use of the mobile phone simulator, have a look at the code by right-clicking the **for** statement and selecting **Edit Source**.

8. Select the **Rates** tab in the Code Coverage viewer

   The **Rates** tab is used to display the various coverage levels for

- the entire application
- each source file
- individual functions/methods

   Click various nodes in the Report Window in order to browse the Rates tab. Note how a selection of the Root node gives you a summary of the entire application.

9. Select the menu item **File**->**Save Project**

   Further Work

Conclusion of Exercise Two

With virtually minimal effort, you have successfully instrumented your

source code for all four runtime analysis features. Manual interaction (in your case, via a mobile phone simulator) was monitored, and the subsequent runtime analysis results were displayed for you graphically. Source code is immediately accessible from these reports, so nothing prevents the developer from using the results to correct possible anomalies.

In addition, using the Test by Test option provided with each runtime analysis feature (introduced in the Further Work section for code coverage), you can easily discern the effectiveness of a test, ensuring maximal reuse without waste.

Your next step is to use the runtime analysis results to remove memory leaks, improve performance, and increase code coverage.

Exercise Three

Introduction to Exercise Three

In this exercise you will:

- Improve the UMTS base station code by eliminating memory leaks and by improving performance
- Increase code coverage
- Rerun your manual interaction to verify that the defects have been fixed

Using Memory Profiling to Remove Memory Leaks

By using the call stacks displayed in the Memory Profile viewer, you will deduce the corrections that need to be made to eliminate memory errors.

1. Select the **Memory Profile** tab.

2. Select the **ABWL** error node in the Report Window on the left hand side of the screen.

Have a look at the call stack for the Late Detect Array Bounds Write error. Three C++ methods are listed.

3. Select the last function first, the one that occurs inside **main()**

   Within the main() procedure a **UmtsServer** object is instantiated. Nothing looks out of sorts here, so return to the call stack.

4. Close the source file for the main() procedure, and then click the second function from the bottom in the call stack referenced by the **ABWL** error - the **UmtsServer** constructor.

   The next function in the stack is the **UmtsServer** constructor. The line in the constructor that is flagged, the creation of a **NetworkNodes** object, is a call to the **List** constructor. Continue to follow the sequence of events.

5. Close the source file for the **UmtsServer** constructor, and then click the top function in the call stack referenced by the **ABWL** error - the **List** constructor.

   The highlighted line is a call to malloc. A quick look at this function shows that a return to the UmtsServer constructor is fairly quick, and nothing seems unusual. You should continue to track the string of events as they happened to see if the ABWL error shows itself. Return to the UmtsServer constructor.

6. Close the source file for the **List** constructor, and then click the second function from the bottom in the call stack referenced by the **ABWL** error - the **UmtsServer** constructor.

   What happens next? The **NetworkNodes** object was assigned 3 **List** objects in an array. Immediately following the call to the **List** constructor, 4 elements are assigned to this array. Not good. The **NetworkNodes** object should be an array of 4 **List** objects, not 3.

7. In the source code, change the line

```
networkNodes = new List(3);
to
networkNodes = new List(4);
```

8. Click the menu item **File->Save**. The revised file UmtsServer.cpp is saved and both file tagging (that is, function/method/class extraction) and static metrics recalculation are performed.

   This should fix the ABWL error. Before redoing you manual test to verify if the memory error was fixed, move on to the Performance Profile viewer and see if you can streamline the performance of the UMTS base station code.

   As for the other memory warnings - that's for you to figure out!

Using Performance Profiling to Improve Performance

Now you will use information in the Performance Profile viewer to determine if you can improve performance in the UMTS base station code.

1. Select the Performance Profile tab.

2. Within the table, left-click the column title Avg F Time (Average Function Time) in order to sort the table by this column. (You may want to scroll down the report a bit to view more data elements in the table.)

   For this exercise you have sorted by the Average Function Time - that is, you're looking at functions that take, on average, the longest time to execute. This is isn't the only potential type of bottleneck in an application - for example, perhaps it is the number of times one function calls its descendants that is the problem - but for this exercise, you will look here first.

   As the developer of this UMTS base station, you would know that the C function **tcpsck_data_ready()** does take a fair amount of time to execute - so you won't look here first (although feel free to have a look if you wish). Instead look at a different function in the table.

3. Select the link for the C function **checkUmtsNetworkConnection()**

   A quick look at the source code shows you that the developer treated this as a dummy function, inserting a "time-waster" to make it appear as if the function were executing. Simply comment out the line.

4. Change the code from

```
doSomeStuff(1);
to
// doSomeStuff(1);
```

5. Select the menu item **File->Save**

   This way, the **checkUmtsNetworkConnection()** method will do nothing at all. The next time you perform the manual test, this C++ method should have an execution time of 0.

   There is another UmtsServer class method that also needs to be improved. Have a look, if you wish.

Using Code Coverage Analysis to Improve Code Coverage

You will now use the information gathered by the code coverage analysis feature to modify the manual test in such a way as to improve code coverage.

1. Select the **Code Coverage** tab.

2. If necessary, select the **Source** tab of the Code Coverage viewer

3. In the Report Window on the left-hand side of the screen, open the **UmtsConnection.cpp** node and then select the **processMessages()** child node

4. Drag the slider bar down slightly until you see the line:

   if (strcmp(msg->phoneNumber,"5550001")==0)

Notice how the **if** statement was never true - the **else** block is green, but the

if block is red. In order to improve coverage of this **if** statement, you need to make the boolean expression evaluate to true.

According to this code, the **if** expression would evaluate to true if mobile phone sends the phone number **5550001**. You should do that.

You will now rerun the UMTS base station executable, restart the mobile phone simulator, and dial this new phone number. When you have finished, you will check the memory profiling, performance profiling, and code coverage analysis reports to see if you have improved matters.

Redoing the Manual Test

You have changed some source code, so some of the UMTS base station code will have to be rebuilt. The integrated build process of PurifyPlus for Linux is aware of these changes, so you do not have to specify the particular files that have been modified.

1. Select the menu item View->Other Windows->Project Window.
2. Select the Project Browser tab in the Project Explorer window that has now appeared on the right-hand side of the UI.
3. Right-click the BaseStation application node and select Build (If you select Rebuild, all files will be rebuilt. Build simply rebuilds those files that have been changed. If no files had been changed, you could have just selected Execute BaseStation.)
4. Once the UMTS base station is running (indicated by the appearance of the Runtime Trace viewer), run the mobile phone simulator as before.
5. Click the mobile phone's On button .
6. Wait for the mobile phone to connect to the UMTS base station (if you watch the dynamic trace closely, you'll notice a display of all the actions that occur when a phone registers with the server). The time should

appear in the mobile phone window.

7. Once connected, dial the phone number 5550001, then press the [OK] button again to send this number to the UMTS base station (again, watch the dynamic trace update).

8. Success! You have connected to the intended party. Stop right here to see the results of your work. Close the mobile phone window by clicking the [X] button on the right side of its window caption. As you may recall, this action will shut down the UMTS base station as well.

9. The UMTS base station has stopped running when the green execution light next to the execution timer - located beneath the Project Explorer window on the lower right-hand side of the UI - stops flashing [00:00:19]. Wait for it to stop flashing.

10. In the Project Explorer window, on the Project Browser tab, double-click the BaseStation application node. All four runtime analysis reports will open with refreshed information. (Alternatively, right-click the BaseStation node and select View Report->All.)

11. Close the Project Explorer window to the right and the Output Window at the bottom.

So have you improved your code and increased code coverage?

Verifying Success

Was the memory leak eliminated?

1. Select the **Memory Profile** tab.

2. Maximize the window

3. In the Report Window on the left-hand side of the screen, look inside the node labeled **Test #2** - do you see the **ABWL** error anymore?

You successfully eliminated the **ABWL** error. Have you improved performance?

4. Select the Performance Profile tab.

5. Select the menu option Performance Profile->Test by Test

6. In the Report Window on the left-hand side of the screen, left-click the node labeled Test #2

7. Sort the table by Avg F Time - do you see the function checkUmtsNetworkConnection()?

You successfully improved performance. Was code coverage improved?

8. Select the **Code Coverage** tab.

9. In the Report Window on the left-hand side of the screen, open the node for **UmtsConnection.cpp** and then left-click the method **processMessages()**

10. Scroll down until you can see the **if** statement for which you have attempted to force an evaluation of true - did you? Has code coverage been improved?

You successfully improved code coverage. Note, by the way, that you can discern what this second manual interaction has gained you in terms of code coverage.

11. Select the menu option Code Coverage->Test by Test

12. In the Report Window on the left-hand side of the screen, reselect the method processMessages()

13. With your mouse anywhere within the Source tab of the Code Coverage viewer, right-click and select CrossRef

14. Scroll the Code Coverage viewer to expose the line of code that has been newly covered and then left-click it:

```
strcpy(response.command,cmd_accepted);
```

Notice that only **Test #2** is mentioned. However, what tests are listed for the **if** statement itself?

15. Left-click the line

```
if (strcm(msg->phoneNumber,"5550001")==0)
```

    Both **Test #1** and **Test #2** are listed. As further proof, do the following.

16. With your mouse anywhere on the **Source** tab of the **Code Coverage** viewer, right-click and deselect **Cross Reference**

17. In the Report Window, on the left-hand side of the screen, open the **Tests** node and deselect the checkbox next to **Test #2**.

Since you have deselected **Test #2**, all you are left with is the code coverage that has resulted from running **Test #1**, and **Test #1** never forced the **if** statement to evaluate to true. Thus the newly covered code has become red again - in other words, unevaluated.

Conclusion of Exercise Three

After correcting the UMTS base station code directly in the PurifyPlus for Linux Text Editor, you simply rebuilt your application and used the mobile phone simulator to initiate further interaction. A second look at the runtime analysis reports validated the accuracy of your changes. Consider the speed with which you could perform these monitoring activities once you are familiar with the user interface...

Conclusion

Command Line Usage of PurifyPlus for Linux

Your experience with PurifyPlus for Linux has been focused on usage of its Graphical User Interface. However, everything can equally be performed from the command line.

The key to command line usage is the **attolcc** instrumentation launcher and the **attolcc1/attolccp** instrumentor. Both perform the instrumentation necessary for runtime analysis. **attolcc** provides the additional capability of calling your compiler following instrumentation; **attolcc1** and **attolccp** simply instrument, leaving the compilation and linkage phase to you.

If you browse to the PurifyPlus for Linux installation folder, and then open the folder **examples\BaseStation_C\src**, you will find a makefile within that is configured to instrument and then build the UMTS base station. It is a classic example of how to use **attolcc**.

NOTE - The Reference Manual can provide you with detailed information about **attolcc**, **attolcc1** and **attolccp**.

When you execute instrumented code from the command line, a single runtime analysis output file is created in the build directory. This file - named **atlout.spt** - is a multiplexed data file that must subsequently be split into individual report files for each runtime analysis feature. This split is performed by the function **atlsplit**:

```
atlsplit atlout.spt
```

NOTE - The Reference Manual can provide you with detailed information about **atlsplit**.

The files created by the split are:

- .tio, .fdc - code coverage report files

- .tsf, .tdf - runtime tracing report files
- .tpf - memory profiling report files
- .tqf - performance profiling report files

To view the actual reports in PurifyPlus for Linux, simply pass these files to the PurifyPlus for Linux binary:

```
studio *.tsf *.fdc *.tio *.tqf *.tdf *.tpf
```

Conclusion - with a Word about Process

Rational PurifyPlus for Linux has been built expressly with the development of mission and business-critical software in mind. Effort has been made to ensure that runtime analysis can be blended as seamlessly as possible into your current development process; minimal overhead stands between you and the use of a full complement of runtime analysis features.

So use them! It should be automatic - part of all your development and regression testing efforts. As you have seen, these features are only a mouse-click away so there is absolutely no drain on your time.

You may be concerned about the instrumentation - "But I don't want my final product to be an instrumented application. Doesn't it have to be if I'm testing instrumented code?" No, it does not have to be:

1. Using the code coverage feature, generate a series of tests that cover 100% of your code
2. Instrument that code for full runtime analysis
3. Uncover and address all reliability errors as you test (e.g. memory leaks, overly slow functions, improper function flow, untested code)
4. Now uninstrument your code - that is, simply shut off all runtime analysis features and rebuild your application
5. Run your regression suite of tests once more, this time looking only for

functional errors

6. No errors? Time to move on to the next iteration or - even better - ship.

Make it part of your development process, just another step before you check in code for the night. Rational PurifyPlus for Linux simplifies runtime analysis to such an extent that there is no longer a reason not to do it.

Java Track

Runtime Analysis for Java

Runtime analysis refers to the ability of PurifyPlus for Linux to monitor an application as it executes. There are a variety of advantages to be gained from this monitoring:

- memory profiling
- performance profiling
- code coverage analysis
- runtime tracing

Runtime Analysis with Rational PurifyPlus for Linux

Memory Profiling

One of the reasons for Java's success is its ability to perform memory management - that is, Java is designed to ensure memory is properly allocated and freed. Does this mean you, as a developer, no longer have any responsibility regarding your software's usage of memory?

No.

There are two primary reasons for a developer to remain vigilant:

- Java applications CAN leak memory. Not in the traditional way, where memory is no longer referenced by your application and yet not accessible by the system OS - such a problem can not occur. However, if you allocate memory, use it, then fail to free (i.e. dereference), then the Java garbage collector will never reclaim it. Do this enough and your system will still run out of memory.
- Excessive memory usage can result in application slowdown. Do you know how much memory your application is using at any given time? If you have access to limited memory, do you know how much your application has allocated? Are there places in your code that could be optimized to use less memory, thereby freeing systems resources for other activities?

A memory profiling utility indicates a running tally of allocated memory as well as those portions of your code that reference memory at a specified moment in time (such as when the program exits). Such information can be used to ensure all unnecessary memory has been dereferenced and that memory usage has been optimized.

This function is provided in Rational PurifyPlus for Linux by the memory profiling feature for the Java language.

Performance Profiling

Optimal performance is, needless to say, crucial for business and mission critical systems. Measuring performance can be quite difficult, however, particularly when it comes to determining the specific functional bottlenecks in your system.

That's where performance profiling monitors come in. These tools watch your application as it executes, measuring statistics such as:

- How often a function is called
- How long it takes for that function to execute

- Which functions are the bottlenecks of your application

With this information you can optimize your code, ensuring all real-time constraints placed upon your system are accommodated.

This function is provided in Rational PurifyPlus for Linux by the performance profiling feature for the Java language.

Code Coverage Analysis

One of the greatest difficulties a developer experiences is a failure to determine the portions of code that have gone untested. For many embedded systems, failure is not an option, so every part of an application must be thoroughly tested to ensure there is no unhandled scenario or dead code.

In addition, product managers need a concrete measurement to determine where the team is in the development cycle - in particular, how much more testing needs to be done. A decreasing number of defects does not necessarily mean the product is ready; it might simply mean the portions of code that have been tested appear to be ready.

Code coverage measurement tools monitor your running application, flagging every line of code as it executes. Advanced tools - such as PurifyPlus for Linux - are also able to differentiate different types of execution, such as whether or not a **do-while** loop executed 0 times, 1 time, or 2 or more times. These advanced measurements are critical for software certification in industries such as avionics.

This function is provided in Rational PurifyPlus for Linux by the code coverage feature for the Java language.

Runtime Tracing

As all software developers quickly learn, intentions don't necessarily

translate into reality. There can often be a vast difference between what you want to happen and what actually happens as your application executes.

This problem becomes more severe when the code is inherited. Yes, you could try to piece things together yourself, but system complexity might just undercut your efforts at understanding the code.

And what about multi-threaded applications? If you've ever encountered race conditions or deadlocks, you know how difficult it can be to uncover the source of the problem.

This is where runtime tracing monitors come in. These utilities graphically display the sequence of function or method calls in your running application - as well as the active threads - illustrating through pictures what is actually happening. With this information, unexpected exceptions can be easily traced back to their source, complex procedures can be distilled to their essence, threading conflicts can be resolved and inherited code can jump off the page and display its inherent logic.

This function, using the industry standard Unified Modeling Language for its graphical display, is provided in Rational PurifyPlus for Linux by the runtime tracing feature for the Java language.

Runtime Analysis Exercises

The following exercises will walk you through a typical use case involving the four runtime analysis features of PurifyPlus for Linux to which you have just been introduced. Pay close attention not only to the capabilities of these features but also to how they are used. The better you understand these features, the more quickly you will be able to adopt them within your own development process.

Reminders before you begin:

1. If you have never run this tutorial before, make sure your machine has a temporary folder in which you can store the test project you will be

creating. For the tutorial, it is assumed that the test project will be stored in a folder called **tmp**

2. Do you have JDK 1.3.1 or 1.4.0 installed? This is necessary for performance of the tutorial. See this page for more information.

3. 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 local JDK. Only the GNU compiler and JDK specified within these dialogs will be accessible within PurifyPlus for Linux.
   Is the JDK located during installation of PurifyPlus for Linux the only JDK installed on your machine? If so, skip the rest of this section. If not - or if you are not sure, or if the JDK was installed after PurifyPlus for Linux - then you should ensure the proper JDK will be accessible by performing the following:

- From a command prompt, execute the shell script 'ConfigureJavac.sh'
- Follow the prompts until the proper JDK is located

  This shell script is located in the folder:

- Red Hat - (install dir/releases/PurifyPlusForLinux.v2002R2/bin/intel/linux_redhat
- SuSE - (install dir)/releases/PurifyPlusForLinux.v2002R2/bin/intel/linux_suse

  This shell script depends on a properly configured environment. If, for some reason, your environment is not properly configured (indicated by the interactive dialog):

- Set your current directory to the applicable shell script folder mentioned above
- Execute the environment configuration shell script - '.ppluslinuxinit.sh'

4. If you have run this tutorial before, don't forget to undo the source file

edits you made the last time you ran through it. The following files are modified during the tutorial:

- **NetworkLoadMonitor.java**
- **LogServer.java**

Exercise One

Introduction to Exercise One

In this exercise you will:

- create a new project in which the UMTS base station source code will be referenced

If you need a refresher about the application you will be using during this tutorial, look here; otherwise, please proceed.

Creating a Project

Typically, there is a one-to-one relationship between your current development project and a PurifyPlus for Linux project. Although your development project may consist of more than one application, these applications often possess a common theme. Use the PurifyPlus for Linux project to enforce that theme.

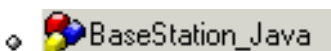
1. To start Rational PurifyPlus for Linux, type **studio** on the command line.
2. Select the **Get Started** link on the left-hand side of the PurifyPlus for Linux user interface (UI). Two links will appear on the right-hand side of the UI - one called **New Project** and one called **Open Project**. Select the **New Project** link. You should now see the **New Project Wizard**.
3. In the **Project Name** field, enter **BaseStation_Java** (no spaces). In the **Location** field, select the ... button, browse to the folder in which

you want the BaseStation project to be stored and then select it. (This Tutorial will assume that the project has been stored in the folder \usr\tmp) Click the **Next** button.

4. Select, from the list of supported JDK distributions, the one you intend to use.

5. Click the **Finish** button.

That's it. The project has been created - named **BaseStation_Java** - and a project node by the same name appears on the Project Browser tab of the Project Explorer window on the right-hand side of the UI:



Creating a New Activity

Now that you have created a project, it is time to specify:

- your development project's source files
- the type of testing or runtime analysis activity you would like to perform first

1. Once a project has been created, the user is automatically brought to the **Activities** page. In this tutorial you are starting with a focus on the runtime analysis features, so select the **Runtime Analysis** link. This will bring up the **Runtime Analysis Wizard**.

2. In the window entitled **Application Files**, you must list all source files for your current development project. For this tutorial, you will directly select the source files. Select the **Add** button .

3. Browse to folder into which you have installed PurifyPlus for Linux and then access the folder **\examples\BaseStation_Java\src\baseStation**

4. Make sure **All Java Files** in the **Files of Type** dropdown box is selected, then left-click-hold-and-drag over all of the eleven Java source files.

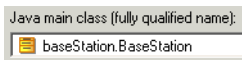
Now click the **Open** button. You should see these eleven files listed in the large listbox of the **Application Files** window. Click the **Next** button.

5. At this time, an analysis engine parses each source file - referred to as tagging. This process is used to extract the various methods and classes located within each source file, simplifying code browsing within the UI.

6. In the window entitled **Selective Instrumentation** you have the ability to select those classes/methods that should not be instrumented for runtime analysis. Such selective instrumentation ensures that the instrumentation overhead is kept to a minimum. For this Tutorial, you will be monitoring everything, so simply click the **Next** button.

7. In the window entitled **Configuration Settings for Java**, you need to define your application's class path as well as the fully qualified name of the main class for your application.

- In the **Class path** text box, click the button, then the button, and then browse to and select the folder **\examples\BaseStation_Java\ src** (located in the PurifyPlus for Linux installation folder). The package used by the Java-based UMTS base station is named **baseStation**, and it's located in the **src** folder you just referenced.
- In the **Java main class** text box, select the **BaseStation** class from the dropdown list and then prepend that name with its package reference - type **baseStation.** (including the final period) immediately before the **BaseStation** reference. Your screen should look like this:



Now click the **Next** button.

8. You have now reached the window entitled **Application Node Name**. Enter the name of the application node that will be created at the

conclusion of the Runtime Analysis Wizard; since you will be monitoring execution of the Java-based UMTS base station, type the word **BaseStation** within the text field labeled **Name**.

9. You also need to make some minor changes to the way you would like PurifyPlus for Linux to work with your JDK. These modifications are specifically aimed at the memory profiling feature and are being used simply to illustrate additional concepts within the Tutorial. At the bottom of the Application Node Name window, click the **Configuration Settings** button.

10. Expand the **Runtime Analysis** node on the left-hand side of the **Configuration Settings** window, expand the **Memory Profiling** child node, and then left-click the **JVMPI** child node.

11. PurifyPlus for Linux uses the JVMPI interface of supported JVMs to acquire memory profiling information. The following custom changes should be made to the Configuration for the purposes of this tutorial:

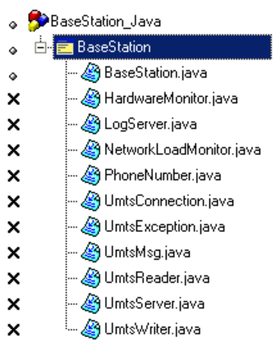
- On the right-hand side of the window, set the Value of the **Take a Snapshot** setting to **After Each Garbage Collection**. Though it is possible to interactively take memory snapshots during execution, setting this option ensures you will have sufficient data to work with in this tutorial.
- Set the Value of **Display Only Listed Packages** to **baseStation** (the Value is case-sensitive, so enter it carefully). This setting ensures you filter out references to objects derived from classes not explicitly defined within the application-under-test.
- Set the Value of **Collect Referenced Objects** to **Yes**. By collecting referenced objects, the memory profiling diff functionality will provide greater visibility into whether or not the application-under-test is properly allocating/deallocating objects.

12. In the **Configuration Settings** window, click the **Apply** button and then the **Ok** button.

13. In the **Application Node Name** window, click the **Next** button.

14. You are now confronted with the Summary window. Everything should be in order, so click the **Finish** button.

The **BaseStation** application node has now been created in the Project Explorer window, on the Project Browser tab, located on the right-hand side of the user interface. If you expand the **BaseStation** application node, you should see the following:



(Why the × in front of all but one .java file? This is because the build process need only reference the source file containing the main Java class when calling the Java compiler. This source file is **BaseStation.java**.)

Conclusion of Exercise One

Have a look at the right side of your screen. This is the Project Explorer window, and within it two tabs are visible.

The first - the **Project Browser** tab - contains a reference to all group, application and test nodes created for the active project. The project node, named **BaseStation_Java**, contains an application node named **BaseStation**; the application node contains a list of all of the source files required to build the UMTS base station application.

The second tab - the **Asset Browser** tab - lets you browse all of your source

and test files. If the selected **Sort Method** is **By Files**, you are presented with a file-by-file listing of test scripts, source code and source code dependents (this last is applicable to C and C++ only). Note how each source file can be expanded to display every class declaration and method definition within them. Double-clicking any test script/source file node will open its contents within the PurifyPlus for Linux editor; double-clicking any class declaration or method definition node will open the relevant source file/header file to the very line of code at which the definition/declaration occurs. (To close a Text Editor window, right-click its associated tab and select **Close**.)

There are two other sort methods as well on the Asset Browser. The first, **By Objects**, lists classes and methods independent of their associated source files. The second, **By Packages**, sorts source files based on their associated Java packages.

You may have noticed along one of the toolbars at the top of the UI that the JDK you selected in the New Project Wizard is listed in a dropdown box. In fact, this is not a reference to the JDK; rather, it is a reference to the Configuration whose base JDK was the one you selected in the wizard. (Configurations are initially named after their base JDK, but this name can be changed.) Should you have multiple configurations for the same project, use this dropdown box to select the active Configuration for execution.

Finally, to the right of the Configuration dropdown list is the Build button ▷. This button is used to build your application for application nodes and the test harness for test nodes. The test harness consists of:

- source files needed to build the application of interest
- stubs
- a test driver

The downward-facing arrow associated with the Build button lets the user decide from which point the build process should initiate and what runtime

analysis features should be used. The runtime analysis features do not have to be used at the same time; this Build options window provides a quick and simple method for deselecting undesired runtime analysis features immediately prior to execution of the build process.

Armed with this knowledge, proceed to Exercise Two.

Exercise Two

Introduction to Exercise Two

In this exercise you will:

- build and execute the UMTS base station application
- manually interact with the UMTS base station application
- view the runtime analysis reports derived from your interaction

Building and Executing the Application

1. When performing runtime analysis, your source code must be instrumented. Instrumentation, by default, is enabled for all four runtime analysis features - that is, for memory profiling, performance profiling, code coverage analysis and runtime tracing. All four features are turned on by default. In order to instrument, compile and execute the UMTS base station application in preparation for runtime analysis, simply ensure the **BaseStation** application node is selected on the **Project Browser** tab of the Project Explorer window, and then click the build button ▷. Do so now.

   NOTE: More information about the source code insertion technology can be found in the **User Guide**, in the chapter **Product Overview->Source Code Insertion**.

2. Notice that in the Output Window at the bottom of the screen, on the **Build** tab, you can see the instrumentation and compilation phases of the

build process as they occur. A double-click on an error listed within any of the Output Window tabs opens the relevant source code file to the appropriate line in the PurifyPlus for Linux Text Editor.

3. The build process has completed, and the UMTS base station is running, when the UML-based sequence diagram generated by the runtime tracing feature appears. (More about this feature in a moment.)

4. Close the Project Explorer window on the right-hand side of the UI by clicking its button; do the same for the Output Window at the bottom of the UI.

Notice how the graphically displayed data in the **Runtime Trace** viewer dynamically grows - this is because the UMTS base station is being actively monitored. The UMTS base station endlessly searches for mobile phones requesting registration; the Runtime Trace viewer reflects this endless loop. If you wish, use the pause button on the toolbar to stop the dynamic trace for a moment (the trace is still being recorded, just no longer displayed in real time). In addition, use the zoom buttons on the toolbar to get a better view of the graphical display (or right-click-hold within the Runtime Trace viewer and select the **Zoom In** or **Zoom Out** options). Undo the Pause when you're ready to proceed.

You'll look at the Runtime Trace viewer in more detail later. Of primary importance right now is interaction with the UMTS base station. You'll do this by using the mobile phone simulator mentioned earlier in the Overview section of this tutorial. Through this manual interaction you will expose careless memory usage, performance bottlenecks, incomplete code coverage, and dynamic runtime sequencing.

Interacting with the Application

1. Start the mobile phone by running the provided mobile phone executable built for your operating system. The mobile phone executable is located within the PurifyPlus for Linux installation folder in the folder

\examples\BaseStation_C\MobilePhone. The name of the executable is **MobilePhone.Linux.** (A launcher shell script - **MobilePhone.sh** - is provided as well.)

2. Click the mobile phone's On button ( ).

3. Wait for the mobile phone to connect to the UMTS base station (if you watched the Runtime Trace viewer closely, you would have noticed a display of all the internal method calls of the UMTS base station that occur when a phone attempts to register). The current system time should appear in the mobile phone window when connection has been established.

4. Once connected, dial the phone number **5550000**, then press the button to send this number to the UMTS base station (again, try to see the Runtime Trace viewer update).

5. Unfortunately, the party you are dialing is on the line so you'll find the phone is busy. Shut off the simulator by closing the mobile phone window via the button in its upper right corner.

   The UMTS base station is designed to shut off when a registered phone goes off line. Not a great idea for the real world, but it serves the Tutorial's purposes well. Alternatively, you could have just used the Stop Build/Execute button located next to the Build button on the toolbar .

6. The UMTS base station has stopped running when the green execution light next to the execution timer - located beneath the Project Explorer window on the lower right-hand side of the UI - stops flashing . Wait for it to stop flashing.

   Everything that occurred at the code level in the UMTS base station was monitored by all four runtime analysis features. Once the UMTS base station stopped (i.e. once the instrumented application stopped), all runtime analysis information was written to user accessible reports that

are directly linked to the UMTS base station source code. In order to look at these reports:

7. Reopen the Project Explorer window by selecting the menu item **View->Other Windows->Project Window**

8. In the Project Explorer window, on the Project Browser tab, double-click the **BaseStation** application node. All four runtime analysis reports will open. (Alternatively, right-click the **BaseStation** application node and select **View Report->All**.)

9. Close the Project Explorer window to create room for the now-opened reports. You may also want to resize the left-hand window to gain additional room.

Runtime Analysis - Runtime Tracing

1. Select the **Runtime Trace** tab.

2. As you recall, the Runtime Trace viewer displayed all objects and all method calls involved in the execution of the UMTS base station code. Using the toolbar buttons , zoom out from the tracing diagram until you can see at least four vertical bars.

3. Make sure you are looking at the top of the runtime tracing diagram using the slider bar on the right.

4. Right-click within the runtime tracing diagram and select **Hide Memory Usage Bar**. Repeat in order to select **Hide Coverage Bar** and **Hide Thread Bar**. You will return to these bars in a moment.

   What you are looking at is a sequence diagram of all events that occurred during the execution of your code. This sequence diagram uses a notation taken from the Unified Modeling Language, thus it can be correctly referred to as a UML-based sequence diagram.

   The vertical lines are referred to as lifelines. Each lifeline represents a

Java object instance. The very first lifeline, represented by a stick figure, is considered the "world" - that is, the operating system. In this UMTS base station tracing diagram, the next lifeline to the right represents an object instance named **Obj0**, derived from the **UmtsServer** class.

Green lines are constructor calls, black lines are method calls, red lines are method returns, and blue lines are destructor calls. Hover the mouse over any method call to see the full text. Notice how every call and call return is time stamped.

Everything in the Runtime Trace viewer is hyperlinked to the monitored source code. For example, if you click on the **Obj0::UmtsServer** lifeline, the source file in which the UmtsServer class definition appears is opened for you, the relevant section highlighted. (Close the source file by right-clicking the tab of the Text Editor and selecting **Close**.) All function calls can be left-clicked as well in order to view the source code. Look at the very top of the **Obj0::UmtsServer** lifeline. It's "birth" consists of a **UmtsServer()** constructor. Left-click the constructor if you wish to view the steps that occur when an object of the UmtsServer class is instantiated.

Notice how the window on the left-hand side of the user interface - called the **Report Window** - contains a reference to all classes and class instances. Double-clicking any object referenced in this window will jump you to its birth in the Runtime Trace viewer. This window can also be used to filter the runtime tracing diagram; closing a node associated with a source file or class will collapse all of the associated lifelines into a single, consolidated lifeline.

Continue to look around the trace diagram. Can you locate the repetitive loop in which the UMTS base station looks for attempted mobile phone registration (it always starts with a call to the method **baseStation.LogServer.checkLog()**)? You can filter out this loop using a couple of methods. One is to simply hover the mouse over a method or function call you wish to filter, right-click-hold and select **Filter**

Message. An alternative method would be to use a predefined filter. You will do both.

5. Hover the mouse over any call of the **baseStation.LogServer.checkLog()** method, right-click-hold and select **Filter Message** - the function call should disappear from the entire trace.

6. Select the menu item **Runtime Trace->Filters** (you'll see the filter you just performed listed here) Click the **Import** button, browse to the installation folder and then the folder **\examples\BaseStation_Java**, and then **Open** the filter file **filters.tft**

7. Left-click the checkbox next to the just imported filter named **BaseStation Phone Search Filter**.

8. Click the **OK** button.

   The loop has been removed.

9. Using the Zoom Level dropdown list on the toolbar, select a level of 50%:

10. Right-click-hold in the Runtime Trace viewer and select **Show Memory Usage Bar**.

    You can now see, along the left-hand side of the runtime tracing diagram, a red, vertical bar. This bar is the **Memory Usage Bar**, and it is a graphical representation of the amount of memory allocated by the monitored application at any moment represented within the runtime tracing diagram. The caption of the Memory Usage Bar indicates the maximum amount of allocated memory that occurred during execution, while the mouse tool tip can be used to discern the amount of allocated memory at any moment along the graph. (Depending on your JVM, you may also notice garbage collection, indicated by areas where there is a

sudden drop in the number of allocated bytes.)

This diagram can be used to expose memory intensive parts of your program that may in fact be needless churn that slows down overall execution time. You could trigger garbage collection immediately prior to suspect moments within your application, using the Runtime Trace viewer to help you decide where the garbage collection should occur, to study whether or not memory usage has become excessive. Note that this feature is specific to Java support.

11. Right-click-hold in the Runtime Trace viewer and select **Hide Memory Usage Bar**.

12. Right-click-hold in the Runtime Trace viewer and select **Show Coverage Bar**.

    Now you are looking at the **Coverage Bar**. It highlights, in synchronization with the runtime tracing diagram, the percentage of total code coverage achieved during execution of the monitored application. The Coverage Bar's caption states the overall percentage of code coverage achieved by the particular interaction presently displayed in the Runtime Trace viewer. Scroll down the runtime tracing diagram; note how code coverage gradually increases until a steady state is achieved. This steady state is achieved following the moment at which the mobile phone has connected to the UMTS base station. Dialing the phone number increases code coverage a bit; shutting off the phone creates a last burst of code coverage up until the moment the UMTS base station is shut off. Can you locate where, on the runtime tracing diagram, the mobile phone simulator first connected to the UMTS base station? Note that the Coverage Bar is available for all supported languages.

13. Right-click-hold in the Runtime Trace viewer and select **Hide Coverage Bar.**

14. Right-click-hold in the Runtime Trace viewer and select **Show Thread Bar**.

    Now you are looking at the **Thread Bar**. The UMTS base station is actually a multi-threaded application; the Thread Bar graphically indicates the active thread at any given moment within the runtime tracing diagram. (Hovering your mouse over the Bar reveals the name of the active thread within a tool tip.) A left-click on the Thread Bar opens a threading window, detailing thread state changes throughout your application's execution. Pressing the Filter button in this detail window specifies the state of each thread within the region of the Thread Bar that was double-clicked. Note that this thread monitoring feature is also available for the C++ language.

15. Right-click-hold in the Runtime Trace viewer and select **Hide Thread Bar.**

    Not only can the runtime tracing feature capture standard function/method calls, but it can also capture thrown exceptions.

16. View the very bottom of the runtime tracing diagram using the slider bar.

    Do you see the icons for the catch statement - catch exception ? The first **Catch Exception** statement is preceded by a diagonal **Throw Exception**. Why diagonal? Because when the exception was thrown, prior to executing the Catch statement, the **UmtsException** constructor was called. Click various elements to view the source code involved in the thrown exception and thus decipher the sequence of events.

    This exception occurred by design, but it is clear how the runtime tracing feature, through the power of UML, would be extremely useful if you have:

- inherited old or foreign code

- unexpected exceptions
- questions about whether what you designed is occurring in practice

  Further Work

Runtime Analysis- Memory Profiling

1. Select the **Memory Profile** tab.
2. Select the menu item **Memory Profile**->**Hide/Show Data**->**Hide/Show Referenced Objects**.

The Memory Profile viewer displays a memory usage report for the application of interest. The Report Window on the left-hand side of the UI displays a list containing each memory snapshot and the time at which they occurred; as you may recall, the Configuration was updated so that a snapshot would occur immediately following each garbage collection. The Memory Profile tab contains a sortable table (i.e. sortable via a left-click on a column header) with the following information:

- **Method** - Each method that, when called, resulted in the instantiation of an object. A left-click on any method names brings you to the portion of source code in which this method has been defined.
- **Referenced Object Class** - If any method in the first column continues to reference an object at the time of the snapshot, the object is listed in this column. Of course, many objects are allocated and deallocated before a snapshot - in this case, the object allocation is recorded but the object reference is not.
- **Allocated Objects**- Total number of objects created by a method throughout execution of the monitored application.
- **Allocated Bytes** - Total number of bytes associated with the objects created by a method.
- **L + D Allocated Objects** - Total number of objects created by the "local"

method and by any descendant methods - that is, by any method that was called as a result of the specified method.

- **L + D Allocated Bytes** - Total number of bytes associated with the objects created by the "local" method and by any descendant methods.

Note how this table is referred to as a "snapshot" at the very top. A user is able to predefine moments at which a memory snapshot should take place - this is done via Configuration Settings. At each snapshot, the JVMPI interface of the targeted JVM is queried and information about each individual method is acquired. For example, if you have designed a particular, cyclic portion of your code to deallocate all unnecessary memory prior to each iteration, set a snapshot to occur each time the cycle is entered. The Memory Profile report contains diff functionality - you will explore this capability later - that can tell you if additional memory remains allocated when the cycle is reentered.

Notice how easily this information has been acquired; no work was required on your part. A real advantage is that memory profiling can now be part of your regression test suite. Traditionally, if embedded developers looked for careless memory allocation/deallocation at all, it was done while using a debugger - a process that does not lend itself to automation and thus repeatability. The memory profiling feature lets you automate memory leak detection.

Further Work

Runtime Analysis - Performance Profiling

1. Select the Performance Profile tab.

The Performance Profile viewer displays the execution time for all methods executing within the application of interest, thereby allowing the user to uncover potential bottlenecks. First, the one or more methods requiring the most amount of time are displayed graphically in a pie chart (up to six functions will be displayed if each is individually

responsible for more than 5% of total execution time. This is then followed by a sortable list of every method, with timing measurements displayed.

Notice how the function **readString()** was responsible for around 75% to 85% of the time spent processing information in the UMTS base station. By looking at the table, where times are listed in microseconds, we can see that this function's average execution time was between 6 to 7 seconds (it will vary somewhat based on your machine) and that it has no descendents - i.e. it never calls and then awaits the return of other functions or methods (which explains why the **Function** time matches the **F+D** time). Is this to be expected? If you wished, you could click on the function name in the table to jump to that function to see if its execution time can be reduced.

Each column can be used to sort the table - simply click on the column heading.

2. Click the column heading entitled **F+D Time**

Interestingly, though **readString()** clearly uses the largest amount of execution time, it is not the "slowest" method when considering descendants. That distinction goes to **readMsg()**; though quick by itself, it's execution time when including descendants is the slowest of all. However, a quick investigation of the **readMsg()** function would reveal that this function calls - and that awaits the return of - **readString()**, which explains why the execution time of **readMsg()** takes longer than **readString()**.

Of course, since this is a multi-threaded application, it is possible for one function to reveal itself as the slowest performer while, overall, the monitored application is typically busy doing other things. This would explain why the runtime tracing diagram does not indicate monopolization of UMTS base station execution following a call to the **readString()** method (have a look; search for *readString* using the option on the toolbar), and thus why performance profiling is such a

valuable supplement to code optimization.

As with the memory profiling feature, notice how easy it was to gather this information. Performance profiling can now also be part of your regression test suite.

Further Work

Runtime Analysis - Code Coverage Analysis

1. Select the **Code Coverage** tab.

   And finally, here you have the code coverage analysis report. The code coverage feature exposes the code coverage achieved either through manual interaction with the application of interest or via automated testing.

   On the left hand side of the screen, in the Report Window, you see a reference to **Root** and then to all of the source files of the UMTS base station. **Root** is a global reference - that is, to overall coverage. For each individual source file, a small icon to the left indicates the level of coverage (green means covered, red means not covered).

   In the Code Coverage viewer, on the **Source** tab, a graphical summary of total coverage is presented in a bar chart - that is, information related to **Root**. Five levels of code coverage are accessible for Java, and those five levels are represented here. (Four more levels of coverage are accessible when working with the C language - up to and including Multiple Conditions/Modified Conditions.) Notice how, on the toolbar, there is a reference to these five possible coverage levels F E B I L.

2. Deselect **Loops Code Coverage** L.

   Notice how the bar chart is updated.

3. Reselect **Loops Code Coverage** L**.**

4. In the Report Window to the left, select the **HardwareMonitor.java**

node.

The **Source** tab now displays the source code located in the file **HardwareMonitor.java**. This code is colored to reflect the level of coverage achieved. Green means the code was covered, red means the code was not covered.

Within the **run()** method you should see a **while** statement that is colored orange and sitting on a dotted underline. This is because the **while** statement was only partially covered.

5. Click on the orange **while** keyword in the **run()** method.

   As you can see, the **while** loop was only executed multiple times, not once or zero times. Why should you care? Well some certification agencies require that all three cases be covered for a **while** loop to be considered covered. If you don't care about this level of coverage, just deselect **Loops Code Coverage**:

6. On the toolbar, deselect **Loops Code Coverage** L.

   Now the **while** loop is green. If you would like to add a comment to your code indicating how this loop is not covered by typical use of the mobile phone simulator, access the code by right-clicking the **while** statement and selecting **Edit Source**.

7. Select the **Rates** tab in the Code Coverage viewer

   The **Rates** tab is used to display the various coverage levels for

- the entire application
- each source file
- individual methods

Click various nodes in the Report Window in order to browse the Rates tab. Note how a selection of the Root node gives you a summary of the

entire application.

8. Select the menu item **File->Save Project**

   Further Work

Conclusion of Exercise Two

With virtually minimal effort, you have successfully instrumented your source code for all four runtime analysis features. Manual interaction (in your case, via a mobile phone simulator) was monitored, and the subsequent runtime analysis results were displayed for you graphically. Source code is immediately accessible from these reports, so nothing prevents the developer from using the results to correct possible anomalies.

In addition, using the Test by Test option provided with each runtime analysis feature (introduced in the Further Work section for code coverage), you can easily discern the effectiveness of a test, ensuring maximal reuse without waste.

Your next step is to use the runtime analysis results to remove memory leaks, improve performance, and increase code coverage.

Exercise Three

Introduction to Exercise Three

In this exercise you will:

- Improve the UMTS base station code by correcting memory usage errors and by improving performance
- Increase code coverage
- Rerun the manual test to verify that the defects have been fixed

Using Memory Profiling to Remove Memory Leaks

By using the diff functionality of the memory profiling feature, you will uncover poor object allocation/deallocation practice within the code.

1. Select the **Memory Profile** tab.

2. If you performed the Further Work section for memory profiling, skip this step; otherwise, select the menu item **Memory Profile**->**Show/Hide Data**->**Diff with Previous Snapshot**.

   Two new columns have appeared - **Referenced Objects Diff AUTO** and **Referenced Bytes Diff AUTO**. These columns contain a diff between each snapshot and the previous snapshot for every listed method; the word "referenced" refers to those objects for which a reference exists following a snapshot. It is also possible for the user to diff any two selected snapshots; this custom diff would be labeled USER to differentiate it from the AUTO diff you will be studying. (Note that a blank cell in any diff column means the object did not exist in the previous snapshot.)

   Recall that the snapshots for this Tutorial occurred immediately after each garbage collection. This means that any object references uncovered by a diff are suspicious; referenced objects can not be deallocated by the garbage collector. Although, from one perspective, memory leaks are not possible with Java, failure to dereference objects will still, in the end, monopolize memory and potentially cause problems with your software.

3. Sort by the column **Referenced Objects Diff AUTO** by clicking on the column header.

4. Search the various snapshots for a method that recurrently is responsible for continuously referenced objects.

   Have you noticed that the **GetChannels()** method reappears

throughout? Perhaps you should look at the code to understand why this method is so often associated with continuously referenced objects.

5. Left-click any reference to the **GetChannels()** method in the first column of the table.

6. Scroll the Text Editor until you can view the **GetChannels()** method.

   Inspection of the **GetChannels()** function reveals that it creates ten new channels each time it is called - which means ten channels should be removed (i.e. dereferenced) elsewhere in the code. This dereferencing is the responsibility of the **ReleaseChannels()** method, located right below the definition of **GetChannels()**, and the **for** statement of this method has been improperly written. Currently, the **ReleaseChannels()** method only dereferences nine objects. You need to fix the code.

7. Modify the **for** statement of the **ReleaseChannels()** method as follows (you are adding an =):

   Change the code from

```
for (i=0;i<10;i++)
to
for (i=0;i<=10;i++)
```

8. Select the menu item **File->Save**. The source file will be retagged - that is, analyzed. You may close the Output Window at the bottom of the UI if you wish.

9. Right-click the tab for the source file you have just modified and select **Close**.

   This should fix the problem. Before redoing you manual test to verify if the memory error was fixed, move on to the Performance Profile viewer and see if you can streamline the performance of the UMTS base station code.

   As for the other methods that appear to continuously reference objects

following garbage collection - are they also leaking? That's for you to figure out!

Using Performance Profiling to Improve Performance

Now you will use information in the Performance Profile viewer to determine if you can improve performance in the UMTS base station code.

1. Select the **Performance Profile** tab.
2. Within the table, left-click the column title **Function Time** in order to sort the table by this column.

   For this exercise you have sorted by the Function Time - that is, you're looking at functions that take the longest time, overall, to execute. This is isn't the only potential type of bottleneck in an application - for example, perhaps it is the number of times one function calls its descendants that is the problem - but for this exercise, you will look here.

   As the developer of this UMTS base station, you would know that the method **read_string()** takes a fair amount of time to execute - so you won't look here first (although feel free to have a look if you wish). Instead look at the second function in the table.
3. Select the link for the method **checkLog()**.

   A quick look at the source code shows you that the developer has added an inexplicable loop - perhaps a dummy function to act as a "time-waster". Simply comment out the line.
4. Change the code from

```
for (x=1,y=100000;x<=100000;x++) y=y/x;
to
// for (x=1,y=100000;x<=100000;x++) y=y/x;
```

5. Select the menu item **File->Save**.

6. Right-click the tab for the source file you have just modified and select **Close**.

You have now eliminated a loop that was adding significant execution time to the **checkLog()** method.

Using Code Coverage Analysis to Improve Code Coverage

You will now use the information gathered by the code coverage analysis feature to modify the manual test in such a way as to improve code coverage.

1. Select the **Code Coverage** tab.

2. If necessary, select the **Source** tab of the Code Coverage viewer.

3. In the Report Window on the left-hand side of the screen, open the **UmtsConnection.java** node, then open the **baseStation.UmtsConnection** child node, and then select the **run()** child node.

4. Drag the slider bar down slightly until you see the line:

   case_connected:

Notice how the **if** statement was never true - only the **else** block is green, but the **if** block is red. In order to improve coverage of this **if** statement, you need to make the boolean expression evaluate to true.

According to this code, the **if** expression would evaluate to true if mobile phone sends the phone number **5550001**. You should do that.

You will now rerun the UMTS base station executable, restart the mobile phone simulator, and dial this new phone number. When you have finished, you will check the memory profiling, performance profiling, and code coverage analysis reports to see if you have improved matters.

Redoing the Manual Test

You have changed some source code, so some of the UMTS base station code will have to be rebuilt. The integrated build process of PurifyPlus for Linux is aware of these changes, so you do not have to specify the particular files that have been modified.

1. Select the menu item **View->Other Windows->Project Window**.
2. Select the **Project Browser** tab in the Project Explorer window that has now appeared on the right-hand side of the UI.
3. Right-click the BaseStation application node and select **Build** (If you select **Rebuild**, all files will be rebuilt. Build simply rebuilds those files that have been changed. If no files had been changed, you could have just selected **Execute BaseStation**.)
4. Once the UMTS base station is running (indicated by the appearance of the Runtime Trace viewer), run the mobile phone simulator as before. (Note how the runtime trace appears to stop - this is because the filter is still applied and thus the recurrent loop is not visible.)
5. Click the mobile phone's On button **.**
6. Wait for the mobile phone to connect to the UMTS base station (if you watch the dynamic trace closely, you'll notice a display of all the actions that occur when a phone registers with the server). The time should appear in the mobile phone window.
7. Once connected, dial the phone number 5550001, then press the button again to send this number to the UMTS base station (again, watch the dynamic trace update).
8. Success! You have connected to the intended party. Stop right here to see the results of your work. Close the mobile phone window by clicking the button on the right side of its window caption. As you

may recall, this action will shut down the UMTS base station as well.

9. The UMTS base station has stopped running when the green execution light next to the execution timer - located beneath the Project Explorer window on the lower right-hand side of the UI - stops flashing 00:00:19. Wait for it to stop flashing.

10. In the Project Explorer window, on **the Project Browser** tab, double-click the **BaseStation** application node. All four runtime analysis reports will open with refreshed information. (Alternatively, right-click the **BaseStation** node and select **View Report**->**All**.)

11. Close the Project Explorer window to the right and the Output Window at the bottom.

So have you improved your code and increased code coverage?

Verifying Success

Was the memory leak eliminated?

1. Select the **Memory Profile** tab.

2. In the Report Window on the left-hand side of the UI, left-click the first snapshot for **Test #2**.

3. Select the column header for **Reference Bytes Diff AUTO**, then select the column header for **Reference Objects Diff AUTO**.

4. Scroll down and study each of the snapshots for Test #2 - is the **GetChannels()** method still responsible for referenced objects?

You successfully eliminated the memory leak. Have you improved performance?

5. Select the **Performance Profile** tab.

6. Select the menu option **Performance Profile->Test by Test**
7. In the Report Window on the left-hand side of the screen, left-click the node labeled **Test #1** in order to deselect it.
8. Sort the table by **Function Time** is it is not sorted by this value already.
9. Do you see the function **checkLog()**?

You successfully improved performance. Was code coverage improved?

10. Select the **Code Coverage** tab.
11. In the Report Window on the left-hand side of the screen, open the node for **UmtsConnection.java**, open the **baseStation.UmtsConnection** child node, then left-click the **run()** node.
12. Select the menu option **Code Coverage->Test by Test.**
13. Scroll down until you can see the **if** statement for which you have attempted to force an evaluation of true - did you? Has code coverage been improved?

You successfully improved code coverage. Note, by the way, that you can discern what this second manual interaction has gained you in terms of code coverage.

14. With your mouse anywhere within the **Source** tab of the **Code Coverage** viewer, right-click and select **CrossRef**
15. Scroll the Code Coverage viewer to expose the line of code that has been newly covered and then left-click it:

```
message.setCommand(UmtsMsg.ACCEPTED);
```

Notice that only **Test #2** is mentioned. However, what tests are listed for the **if** statement itself?

16. Left-click the line

```
if (message.getPhoneNumber().equals("5550001"))
```

Both **Test #1** and **Test #2** are listed. As further proof, do the following.

17. With your mouse anywhere on the **Source** tab of the **Code Coverage** viewer, right-click and deselect **Cross Reference**

18. In the Report Window, on the left-hand side of the screen, open the **Tests** node and deselect the checkbox next to **Test #2**.

    Since you have deselected **Test #2**, all you are left with is the code coverage that has resulted from running **Test #1**, and **Test #1** never forced the **if** statement to evaluate to true. Thus the newly covered code has become red again - in other words, unevaluated.

Conclusion of Exercise Three

After correcting the UMTS base station code directly in the PurifyPlus for Linux Text Editor, you simply rebuilt your application and used the mobile phone simulator to initiate further interaction. A second look at the runtime analysis reports validated the accuracy of your changes. Consider the speed with which you could perform these monitoring activities once you are familiar with the user interface...

Conclusion

Command Line Usage of PurifyPlus for Linux

Your experience with PurifyPlus for Linux has been focused on usage of its Graphical User Interface. However, everything can be equally performed from the command line.

The key to command line usage is the **javic** instrumentation launcher, the **javi** instrumentor, the JVMPI Agent (named **pagent**) and a **javic** Java class for those interested in using **ant**. Both **javic** and **javi** perform the instrumentation necessary for runtime analysis. **javic** provides the

additional capability of calling your compiler following instrumentation; **javi** simply instruments, leaving the compilation phase to you. The JVMPI agent is a dynamic library associated with most J2SE-compliant JVM distributions - it is used to enable memory profiling. The **javic** Java class implements **javic** for use with **ant**, the Java-based build tool provided by the Jakarta Project (http://jakarta.apache.org/ant).

NOTE: For specific information about **javic**, **javi,** the JVMPI agent (**pagent**) and the **javic** Java class, see the Reference Manual.

Before going into further detail about using PurifyPlus for Linux from the command line, note the following regarding data acquisition:

When you execute instrumented code from the command line, a single runtime analysis output file is created in the build directory. This file - named **atlout.spt** - is a multiplexed data file that must subsequently be split into individual report files for each runtime analysis feature. This split is performed by the function **atlsplit**:

```
atlsplit atlout.spt
```

NOTE - The Reference Manual can provide you with detailed information about **atlsplit**.

The files created by the split are:

- tio, .fdc - code coverage report files
- tsf, .tdf - runtime tracing report files
- jpt - memory profiling report files
- tqf - performance profiling report files

Four use cases will be outlined below regarding work from the command line. The source files for the UMTS base station, which will be the subject of the following examples, are located in the PurifyPlus for Linux installation folder, in the folder **examples\BaseStation_Java\src**. This folder contains the

baseStation package which, in turn, contains the Java source files.

Compilation and Execution of the UMTS BaseStation - Not Instrumented for Runtime Analysis

- Compilation of UMTS base station
  1. Make sure the working directory is **(...)/BaseStation_Java/src/baseStation**
  2. Compile the UMTS base station.
     ```
     javac BaseStation.java -classpath
     (...)/BaseStation_Java/src
     ```
- Execution of UMTS base station
  1. Move up one directory to **(...)/BaseStation_Java/src**
  2. Execute the UMTS base station.
     ```
     java baseStation.BaseStation -classpath (...)
     /BaseStation_Java/src
     ```
- Execute the mobile phone simulator to interact with the UMTS base station
  1. Move to the directory **(...)/examples/BaseStation_C/MobilePhone**
  2. Launch the mobile phone.

Compilation and Execution of the UMTS BaseStation - Instrumented for Runtime Analysis

- Compilation of UMTS base station
  1. Make sure the working directory is **(...)/BaseStation_Java/src/baseStation**
  2. Compile the UMTS base station
     ```
     javic -perfpro -trace -proc=ret -block=logical -- javac
     ```

```
BaseStation.java -classpath (...)/BaseStation_Java/src
```

- Execute the UMTS base station and turn on the JVMPI agent

  1. Move down one directory to **(...)/BaseStation_Java/src/baseStation/javi.jir** This folder, created by **javic**, contains instrumented source files.

  2. Execute the UMTS base station and turn on the JVMPI agent
     ```
     java baseStation.BaseStation -classpath
     (...)/BaseStation_Java/src/baseStation/javi.jir -Xint -
     Xrunpagent:-OUT=["../BaseStation.jpt"]-H_C1-H_O5-D_GC-
     P_M[[,baseStation]]-D_O_N-U_S["c"]-N_U["BaseStation"]
     ```

- Execute the mobile phone simulator to interact with the UMTS base station

  1. Move to the directory (...)/examples/**BaseStation_C/MobilePhone**

  2. Launch the mobile phone.

- Split the multiplexed runtime analysis data file

  1. Make sure working directory is **(...)/BaseStation_Java/src/baseStation**

  2. Split the data file
     ```
     atlsplit atlout.spt
     ```

- Open the runtime analysis reports in PurifyPlus for Linux

  1. Run PurifyPlus for Linux, passing the report files as parameters
     ```
     studio *.tsf *.fdc *.tio *.tqf *.tdf *.jpt
     ```

Building with Ant and Not Instrumenting for Runtime Analysis

- Build the UMTS base station using ant

  1. Make sure the working directory is **(...)/BaseStation_Java/src**

  2. Execute **ant** (which uses the file **build.xml**, located in the working directory)
     ```
     ant
     ```

Building with Ant and Instrumenting for Runtime Analysis

- Build the UMTS base station using ant and instrument for runtime analysis

  1. Make sure the working directory is **(...)/BaseStation_Java/src**

  2. Execute **ant** (which uses the file **build.xml**, located in the working directory)

```
ant -DATLTGT=$ATLTGT -
Dbuild.compiler=com.rational.testrealtime.Javic -
Dbuild.actual.compiler=modern -Djavi.options="-perfpro -
trace -proc=ret -block=logical "
```

NOTE: For specific information about **javic**, **javi,** the JVMPI agent (**pagent**) and the **javic** Java class, see the Reference Manual.

Conclusion - with a Word about Process

Rational PurifyPlus for Linux has been built expressly with the development of mission and business-critical software in mind. Effort has been made to ensure that runtime analysis can be blended as seamlessly as possible into your current development process; minimal overhead stands between you and the use of a full complement of runtime analysis features.

So use them! It should be automatic - part of all your development and regression testing efforts. As you have seen, these features are only a mouse-click away so there is absolutely no drain on your time.

You may be concerned about the instrumentation - "But I don't want my final product to be an instrumented application. Doesn't it have to be if I'm testing instrumented code?" No, it does not have to be:

1. Using the code coverage feature, generate a series of tests that cover 100% of your code

2. Instrument that code for full runtime analysis

3. Uncover and address all reliability errors as you test (e.g. memory leaks, overly slow functions, improper function flow, untested code)
4. Now uninstrument your code - that is, simply shut off all runtime analysis features and rebuild your application
5. Run your regression suite of tests once more, this time looking only for functional errors
6. No errors? Time to move on to the next iteration or - even better - ship.

Make it part of your development process, just another step before you check in code for the night. Rational PurifyPlus for Linux simplifies runtime analysis to such an extent that there is no longer a reason not to do it.

Conclusion

3

Proactive Debugging

As software complexity increases, developers must become more responsible for their contribution to the overall development project. It is becoming harder and harder for the developer to consider robust, end-to-end testing of their code an unachievable luxury.

In fact, developers need to proactively debug - that is, treat testing and runtime analysis as an integral part of the development process, rather than waiting for defects to force their hand. Such practice is preached by agile movements such as Extreme Programming and Rational's own Rational Unified Process.

And why should this not be achievable? The advantage of proactive debugging is that it is manageable - testing and runtime analysis are only performed on the code known intimately well by the developer (barring the case of inherited code, where the runtime tracing feature plays such a crucial role). There is little chance for confusion, so the time spent developing and deploying your code are optimized. Defects are eliminated early, ensuring that any system level defects that have slipped through the nets won't find their origin deep in the code.

Matters improve further when one considers the built-in integration that PurifyPlus for Linux possesses with other products in Rational's software development arsenal. PurifyPlus for Linux is integrated with:

- **ClearCase** - Out-of-the-box integration with ClearCase, the industry's clear market leader for version control software. Click here for access to

the Rational ClearCase website.

- **ClearQuest** - Out-of-the-box integration to ClearQuest, the premier change management utility for diversified software teams. Submit context-sensitive defect reports directly from the PurifyPlus for Linux interface. Click here for access to the Rational ClearQuest website.

PurifyPlus for Linux can play a crucial role your in your responsibilities as a developer, helping you manage complexity and deliver optimized, quality code at all stages of the software development lifecycle. Code smarter and finish faster.

Questions?

Questions or comments? Want to share tips? Feel free to send us an e-mail at testrt-info@rational.com. Useful information will be shared on the Latest News and Updates page, accessible to PurifyPlus for Linux customers from the Help menu.

We hope you found this tutorial informative.

Technical Support

4

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

- Name, telephone number, and company name
- Product name and version number
- Operating system and version number (for example, RedHat 7.2)
- Computer make and model
- Your case id (if you're calling about a previously reported problem)
- A summary description of the problem, related errors, and how it was made to occur

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

You can obtain technical assistance by sending e-mail to the appropriate address. E-mail is acknowledged immediately and is usually answered within one working day of its arrival at Rational. When sending an e-mail, place "**PurifyPlus for Linux**" 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: "**CaseID:** <*number*>", where <*number*> is the CaseID number of the issue. For example:

```
CaseID: v0176528 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: Technical Support**" and add your fax number to the information requested above.

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