Rational® PurifyPlus RealTime

Reference Manual

VERSION: 2003.06.00

WINDOWS AND UNIX

Rational®
the software development company

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

Legal Notices

Version Number: 2003.06.00

Third Party Notices, Code, Licenses, and Acknowledgements

Reference Manual Contents

Preface

Welcome to Rational PurifyPlus RealTime.

This Reference Manual contains advanced information to help you use the product from the command line.

Rational PurifyPlus RealTime is a complete runtime analysis solution for real-time and embedded systems. It addresses all runtime analysis needs for the C, C++, Ada, and Java programming languages.

General information about using the product can be found in the *PurifyPlus RealTime User Guide*.

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

Audience

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

You should be familiar with the selected Windows or UNIX platform as well as both the native and target development environments.

Contacting Rational Technical Publications

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

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

Other Resources

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

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

http://www.rational.com/products/testrt/pplus_rt.jsp

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

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

For more information about Rational Software technical publications, see:

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

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

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

Customer Support

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

http://solutions.rational.com/solutions

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

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

- About you:
  Name, title, e-mail address, telephone number
- About your company:
  Company name and company address
- About the product:
  Product name and version number (from the **Help** menu, select **About**).
  What components of the product you are using
- About your development environment:
  Operating system and version number (for example, Linux RedHat 8.0), target

compiler, operating system and microprocessor. If necessary, send the Target Deployment Port **.xdp** file

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

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

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

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

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

Location	Contact
North America	Rational Software, 18880 Homestead Road, Cupertino, CA 95014 voice: (800) 433-5444 fax: (408) 863-4001 e-mail: support@rational.com
Europe, Middle East, and Africa	Rational Software, Beechavenue 30, 1119 PV Schiphol-Rijk, The Netherlands voice: +31 20 454 6200 fax: +31 20 454 6201 e-mail: support@europe.rational.com

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

Command Line Reference

This section provides reference information to help you use PurifyPlus RealTime runtime analysis features from a command line. This can be useful in complex development environments to perform most major tasks in the command line interface under UNIX or Windows operating systems.

Runtime Analysis for Ada

Ada Instrumentor

Purpose

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

Syntax

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

where:

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

Description

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

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

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

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

Options

-PROC [**=RET**]

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

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

-CALL

Instruments Ada functions or procedures.

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

This option specifies how blocks are to be instrumented.

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

By default, the Instrumentor instruments implicit blocks.

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

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

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

-NOPROC

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

-NOCALL

Disables instrumentation of calls.

-NOBLOCK

Disables instrumentation of simple, implicit, or logical blocks.

-NOCOND

Disables instrumentation of basic conditions.

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

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

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

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

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

Provides a set of link files to the Instrumentor.

-STDLINK=<*filename*>

Provides a standard link file to the Instrumentor.

-FDCDIR=<*directory*>

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

-FDCNAME=<*name*>

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

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

These options allow you to explicitly define upon which incoming or returning function(s) the trace dump must be performed. Please refer to **General Runtime Analysis Settings** in the **User Guide** for further details.

-COMMENT=<*comment*>

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

-NOMETRICS

Saves the metrics basic data calculation time.

-RESTRICTION =NOEXCEPTION|NOGENERIC|CSMART

Use this option to set a restriction.

- NOEXCEPTION deactivates instrumentation of exception block branches encountered in the source file. When this option is active, no coverage information is available on exception blocks or on instructions contained in exception blocks.
- NOGENERIC deactivates the instrumentation using a generic Target Deployment Port call. When this option is active, the generated source code may contain uninstrumentable calls. If used with the -CALL option, this can generate compilation errors depending on your application if, for example, you use private packages as well as private sub-packages.
- CSMART generates CSMART compliant code.

-NOSOURCE

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

-NOCVI

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

-METRICS

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

-GENERATEDNAME = CHECKSUM | *<filename>*
-USERNAME = <NAME>

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

- -GENERATEDNAME=CHECKSUM uses a checksum calculated on the instrumented file to create a package name under the form ATC_*<checksum>*, where *<checksum>* has a maximum of four letters.
- -GENERATEDNAME=*<filename>* uses the name of the file to be instrumented, this name is transformed into an Ada identifier and prefixed by ATC_.
- -USERNAME=*<username>*: A name you choose freely by the user and provide on the command line.

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

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

-PREFIX=*<prefix>*

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

By default, no prefix is used.

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

-SPECIFICATION

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

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

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

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

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

-ADA83 | **-ADA95**

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

-INSTRUMENTATION=[COUNT|**INLINE]**

Specifies the Instrumentation Mode:

- **COUNT:** Default Pass mode, each branch generates in 32 bits for profiling purposes. This offers the best compromise between code size and speed overhead.
- **INLINE:** Compact mode. functionally equivalent to *Pass* mode, except that each branch needs only one bit of storage instead of one byte. This implies a smaller requirement for data storage in memory, but produces a noticeable increase in code size (shift/bits masks) and execution time.

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

`-NOINFO`

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

Return Codes

After execution, the program exits with the following return codes

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

All messages are sent to the standard error output device.

Ada Link File Generator

Purpose

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

Syntax

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

where:

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

Description

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

File Extensions

A file extension is a character string of unconstrained positive length (greater than zero). A file name matches an extension if its length is greater than the length of extension, and if the N last characters of the file name are identical to the characters of the extension (N is the length of the extension). For example, **source.ada** matches the **.ada** extension but not **.1.ada** extension.

Permitted and Forbidden Extensions:

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

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

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

Link File Generation

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

Command Line Parameters

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

`-r`

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

`-s`

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

`-f`

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

Return Codes

After execution, the program exits with the following return codes

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

All messages are sent to the standard error output device.

Ada Unit Maker

Purpose

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

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

Syntax

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

where:

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

Description

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

Gnat Naming

The full compilation unit name is set to lower case and all dot characters (".") are replaced with hyphens ("-"). The feature appends the **.ads** extension to the name if the unit is an extension or the **.adb** extension if the unit is a body. The Krunch Gnat short name mode is not supported by the Ada Unit Maker. Please refer to your Gnat documentation for further information.

Rational Apex Naming

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

Options

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

`-l`

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

`-w`

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

`-n APEX|GNAT`

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

Return Codes

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

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

All messages are sent to the standard error output device.

Ada Metrics Calculator

Purpose

The Ada Metrics Calculator produces **.met** static metric files for the specified source files.

Syntax

metada *<source_file>* [**-output_dir=***<output_directory>*]
metada @*<options_file>*

where:

- *<source_file>* is the name of the source file to be analyzed.
- *<output directory>* is the absolute path of the location where the .met static metric file is to be generated.
- *<options_file>* points to a plain text file containing a list of options.

Description

The Ada Metrics Calculator analyzes a specified Ada source file and produces a .met static metric file, which can be opened with the PurifyPlus RealTime GUI.

> **Note** For other languages, the **.met** static metric files are produced by the C, C++ and Java Source Code Parsers.

Return Codes

After execution, the program exits with the following return codes

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

All messages are sent to the standard error output device.

Runtime Analysis for Java

Java Instrumentor

Purpose

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

- Performance Profiling
- Code Coverage
- Runtime Tracing

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

Syntax

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

where:

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

Description

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

The Runtime Analysis tools are activated by selecting the command line options:

- **-MEMPRO** for Memory Profiling
- **-PERFPRO** for Performance Profiling
- **-TRACE** for Runtime Tracing
- **-PROC** and **-BLOCK** for Code Coverage (code coverage levels).

Note that there is no **-COVERAGE** option; the following rules apply for the Code Coverage feature:

- If no code coverage level is specified, nor Runtime Tracing, Memory Profiling, or Performance Profiling, then the default is to have code coverage analysis at the **-PROC** and **-BLOCK=DECISION** level.
- If no code coverage level is specified while one or more of the aforementioned features are selected, then code coverage analysis is not performed.

Detailed information about command line options for each feature are available in the sections below.

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

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

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

Options

-FILE=<*filename*>[{,<*filename*>}] | -
EXFILE=<*filename*>[{,<*filename*>}]

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

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

<*filename*> may contain a path (absolute or relative from the current working directory). If no path is provided, the current working directory is used.

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

-CLASSPATH=<*classpath*>

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

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

-OPP=<*filename*>

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

-DESTDIR=<*directory*>

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

-PROC [=RET]

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

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

-BLOCK=IMPLICIT | DECISION | LOGICAL

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

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

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

By default, the Instrumentor instruments implicit blocks.

-NOTERNARY

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

-NOPROC

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

-NOBLOCK

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

-COUNT

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

-COMPACT

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

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

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

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

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

-DUMPINCOMING=<*service*>[{,<*service*>}]
-DUMPRETURNING=<*service*>[{,<*service*>}]
-MAIN=<*service*>

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

-COMMENT=<*comment*>

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

-NOCVI

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

-NOCLEAN

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

-FDCDIR=<*directory*>

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

specified, each **.fdc** file is generated in the directory of the corresponding source file. If you do not use this option, the default **.fdc** files directory is the current working directory. You cannot use this option with the **-FDCNAME** option.

-FDCNAME=*<name>*

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

-NO_UNNAMED_TRACE

With this option, anonymous classes are not instrumented.

-PERFPRO

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

-TRACE

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

-TSFDIR=*<directory>*

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

-TSFNAME=*<filename>*

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

-INSTRUMENTATION=[FLOW|COUNT|INLINE]

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

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

`-NOINFO`

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

Return Codes

After execution, the program exits with the following return codes

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

All messages are sent to the standard error output device.

Java Instrumentation Launcher

Purpose

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

Syntax

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

where:

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

Description

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

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

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

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

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

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

Customization

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

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

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

Options

The Launcher accepts the following settings:

--atl_threads_max=*<number>*

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

--atl_buffer_size=*<bytes>*

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

--address=*<IPaddress>*

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

--uploader_port=*<port number>*

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

--atl_run_gc_at_exit=0|1

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

--att_on_the_fly=0|1

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

--att_partial_dump=0|1

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

--att_timestamp=0|1

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

--att_heap_size=0|1

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

--att_thread_info=0|1

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

--att_record_max_stack=0|1

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

Example

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

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

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

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

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

Java Instrumentation Launcher for Ant

Purpose

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

Description

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

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

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

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

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

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

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

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

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

Options

The Launcher accepts the following settings:

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

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

--atl_buffer_size=*<bytes>*

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

--address=*<IPaddress>*

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

--uploader_port=*<port number>*

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

--atl_run_gc_at_exit=0|1

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

--att_on_the_fly=0|1

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

--att_partial_dump=0|1

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

--att_timestamp=0|1

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

--att_heap_size=0|1

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

--att_thread_info=0|1

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

--att_record_max_stack=0|1

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

To install the Javic class for Ant:

- Download and install Ant v1.4.1 from http://jakarta.apache.org/ant/
- Set **ANT_HOME** to the installation directory, for example: **/usr/local/jakarta-ant-1.4.1**.
- Add $ANT_HOME/bin in your PATH
- Compile and install the **Javic** class. In the ant directory, type:
  ```
  ant
  ```

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

Example

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

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

```
ant
```

This produces the following output:

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

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

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

This produces the following output:

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

JVMPI Agent

Purpose

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

Syntax

`java -Xint -Xrunpagent``[:`*`<options>`*`] `*`<configuration>`*

where:

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

Description

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

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

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

- Method entries and exits
- Object and primitive type allocations

The JVMPI Agent retrieves source code debug information during runtime. When the Agent receives a snapshot trigger request, it can either execute an instantaneous JVMPI dump of the JVM memory, or wait for the next garbage collection to be performed.

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

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

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

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

Options

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

-H_Cx=<*size*>

-H_Ox=<*size*>

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

-JVM <*prefix*>

By default, the Agent waits for the virtual machine (VM) to be fully initialized before it starts collecting data. This usually relates to the spawning of the first user thread. With the **-JVM** option, data collection starts on the first memory allocation, even if the VM is not fully initialized.

-N_O

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

-D_O_N

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

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

-D_GC

This option requests a JVMPI dump after each garbage collection

-D_PGC

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

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

Activates "Dump Method" mode.

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

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

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

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

Activates "Observe Method" mode.

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

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

User name

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

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

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

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

Filter mode.

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

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

Hostname.

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

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

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

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

Output filename.

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

Example

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

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

Runtime Analysis for C and C++

C and C++ Instrumentor

Purpose

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

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

Syntax

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

where:

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

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

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

Description

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

The Runtime Analysis tools are activated by selecting the command line options:

- **-MEMPRO** for Memory Profiling
- **-PERFPRO** for Performance Profiling
- **-TRACE** for Runtime Tracing

- **-PROC** , **-CALL**, **-COND** and **-BLOCK** for Code Coverage (code coverage levels).

Note that there is no **-COVERAGE** option; the following rules apply for the Code Coverage feature:

- If no code coverage level is specified, nor Runtime Tracing, Memory Profilingor Performance Profiling, then the default is to have code coverage analysis at the **-PROC** and **-BLOCK=DECISION** level.
- If no code coverage level is specified while one or more of the aforementioned features are selected, then code coverage analysis is not performed.

Detailed information about command line options for each feature are available in the sections below.

The C Instrumentor (**attolcc1**) supports preprocessed ANSI 89 or K&R C standard source code without distinction. The ANSI 99 standard is not supported.

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

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

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

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

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

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

General Options

-FILE=*<filename>*[{,*<filename>*}] | -
EXFILE=*<filename>*[{,*<filename>*}]

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

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

<filename> may contain a path (absolute or relative from the current working directory). If no path is provided, the current working directory is used.

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

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

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

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

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

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

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

For the C Instrumentor only. The **-RENAME** option allows you to change the name of C functions *<function>* defined in the file to be instrumented. Doing so, the *f* function will be changed to **_atw_stub_***f*. Only definitions are changed, not declarations (prototypes) or calls.

-REMOVE=*<name>*[**,***<name>*]

This option removes the definition of the function (or method) *<name>* in the instrumented source code. This allows you to replace one or several functions (or methods) with specialized custom functions (or methods) from the TDP.

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

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

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

-INSTANTIATIONMODE=ALL

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

-DUMPCALLING=<*name*>[{,<*name*>]]
-DUMPINCOMING=<*name*>[{,<*name*>}]
-DUMPRETURNING=<*name*>[{,<*name*>}]

These options allow you to explicitly define upon which incoming, returning or calling function(s) the trace dump must be performed. The **-DUMPCALLING** function is for the C language only. Pleaser refer to **General Runtime Analysis Settings** in the **User Guide** for further details.

-NOPATH

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

-NOINFO

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

-NODLINE

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

-TSFDIR[**=**<*directory*>]

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

-TSFNAME=<*name*>

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

-NOINCLUDE

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

Code Coverage Options

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

-PROC[=RET]

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

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

-CALL

Instruments C/C++ function calls.

-BLOCK=IMPLICIT | DECISION | LOGICAL

The -BLOCK option alone instruments simple blocks only.

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

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

By default, the Instrumentor instruments implicit blocks.

-NOTERNARY

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

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

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

FORCEEVALUATION instruments forced conditions.

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

-NOPROC

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

-NOCALL

Specifies no instrumentation of calls.

-NOBLOCK

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

-NOCOND

Specifies no instrumentation of basic conditions.

-COUNT

Specifies count mode.

-COMPACT

Specifies compact mode.

-EXCALL=*<filename>*

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

-FDCDIR=*<directory>*

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

-FDCNAME=*<name>*

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

-NOCVI

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

-METRICS

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

-NOSOURCE

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

-COMMENT=*<comment>*

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

Memory Profiling Specific Options

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

-MEMPRO

Activates instrumentation for the Runtime Tracing analysis feature.

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

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

Performance Profiling Specific Options

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

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

Activates instrumentation for the Runtime Tracing analysis feature.

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

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

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

Runtime Tracing Specific Options

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

`-TRACE`

Activates instrumentation for the Runtime Tracing analysis feature.

`-NO_UNNAMED_TRACE`

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

`-NO_TEMPLATE_NOTE`

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

`-BEFORE_RETURN_EXPR`

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

Return Codes

After execution, the program exits with the following return codes

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

All messages are sent to the standard error output device.

C and C++ Instrumentation Launcher

Purpose

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

Syntax

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

where:

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

Description

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

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

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

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

General Options

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

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

--HELP

Type **attolcc --help** to list a comprehensive list of options, including those of the instrumentor, for use with the instrumentation launcher.

-VERBOSE | -#

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

-TRACE
-MEMPRO
-PERFPRO

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

-FORCE_TDP_CC

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

-NOSTOP

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

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

Code Coverage Options

The following parameters are specific to the Code Coverage runtime analysis feature. These options do not activate Code Coverage. To activate Code Coverage, use the Code Coverage Level options (**-PROC**, **-CALL**, **-COND** and **-BLOCK**).

-PASS | -COUNT | -COMPACT

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

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

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

-COMMENT | -NOCOMMENT

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

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

Use **-NOCOMMENT** to disable the comment setting.

Instrumentation Settings

The instrumentation settings apply to the compilation of the Target Deployment Port Library.

The **0** or **1** values for many conditional settings mean false for 0 and 1 for true.

Compiler Settings

--cflags=*<compilation flags>*

--cppflags=*<preprocessing flags>*

--include_paths=*<comma separated list of include paths>*

--defines=*<comma separated list of defines>*

Enclose the flags with quotes ("") if you specify more than one. These flags are used while compiling the Target Deployment Port Library

By default, the corresponding **DEFAULT_CPPFLAGS**, **DEFAULT_CFLAGS**, **DEFAULT_INCLUDE_PATHS** and **DEFAULT_DEFINES** from the *<ATLTGT>***/tp.ini** or *<ATLTGT>***/tpcpp.ini** file are used

General Settings

--atl_multi_threads=0|1

To be set to 1 if your application is multi-threads (default **0**).

--atl_threads_max=*<number>*

Maximum number of threads at the same time (default **64**).

--atl_multi_process=0|1

To be set to 1 if your application uses fork/exec to run itself or another instrumented application (default **0**). Traces files are named atlout.<pid>.spt.

--atl_buffer_size=*<bytes>*

Size of the Dump Buffer in bytes (default **16384**).

--atl_traces_file=*<file-name>*

Name of the file that is flushed by execution and to be split (default **atlout.spt**).

Memory Profiling Settings

--atp_call_stack_size=*<number of frames>*

Number of functions from the stack associated to any tracked memory block or to any error (default 6).

--atp_reports_fiu=0|1

File In Use detection and reporting (default 1)

--atp_reports_sig=0|1

POSIX Signal detection and reporting (default 1).

--atp_reports_miu=0|1

Memory In Use detection and reporting, ie: not leaked memory blocks (default 0).

--atp_reports_ffm_fmwl=0|1

Freeing Freed Memory and Late Detect Free Memory Write detection and reporting (default 1).

--atp_max_freeq_length=*<number of tracked memory blocks>*

Free queue length, ie: maximum number of tracked memory blocks whom actual free is delayed (default 100).

--atp_max_freeq_size=*<bytes number>*

Sets the free queue size, ie: the maximum number of bytes actually unfreed (default 1048576 = 1Mb)

--atp_reports_abwl=0|1

Late Detect Array Bounds Write detection and reporting (default 1).

--atp_red_zone_size=*<bytes number>*

Size of each of the two Red Zones placed before and after the user space of the tracked memory blocks (default 16).

Performance Profiling Settings

--atq_dump_driver=0|1

Enable the Performance Profiling Dump Driver API **atqapi.h** (default 0).

Code Coverage Settings

--atc_dump_driver=0|1

Enablesthe Coverage Dump Driver API **apiatc.h** (default 0).

Runtime Tracing Settings

--att_on_the_fly=0|1

If set to 1, implies that each tdf lines are flushed immediatly in order to be read on-the-fly by the UML/SD Viewer in Studio (default 0).

--att_item_buffer=0|1

Enable Trace Buffer (not Dump Buffer) if set to 1 (default 0).

--att_item_buffer_size=*<bytes>*

Maximum number of recorded Trace elements before Trace Buffer flush (default 100).

--att_partial_dump=0|1

Partial Message Dump is on if set to 1 (default 0).

--att_signal_action=0|1|2

- 0 means no action when handling a signal (default)
- 1 means toggling dump of messages
- 2 means only flushing the current call stack

--att_record_max_stack=0|1

Display largest call stack length in a note (default 1).

--att_timestamp=0|1

If enabled, record and display time stamp (default 0).

--att_thread_info=0|1

If 1 record and display thread information (default 1).

Component Testing for C++ Contract Check Settings

--atk_stop_on_error=0|1

Call breakpoint function on assertion failure (default 0).

--atk_dump_success=0|1

By default (0), only failed assertions are reported. If enabled, both failed and passed assertions are reported.

--atk_report_reflexive_states=0|1

Trace unchanged states (default 1).

Example

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

Return codes

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

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

```
ERROR : Error during C preprocessing
```

All messages are sent to the standard error output device.

Generic Tools

Graphical User Interface

Purpose

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

Syntax

studio [**-r** *<node>*] [*<filename>*{ *<filename>*}]

where:

- *<filename>* can be an **.rtp** project or **.rtw** workspace file, as well as source files (.c, .cpp, .h, .ada, .java) or any report files that can be opened by the GUI, such as **.tdf**, **.tsf**, **.tpf**, **.tqf**, **.xrd** files.
- *<node>* is a project node to be executed.

Description

The studio command launches the GUI.

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

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

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

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

Example

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

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

The following example opens a UML sequence diagram created by Runtime Tracing.

```
studio my_app.tsf my_app.tdf
```

Trace Receiver

Purpose

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

Syntax

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

where:

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

Description

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

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

To use the Trace Receiver, you must:

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

The Runtime Trace Receptor uses the delimiter to find useful trace data and directs the output to the trace file formats. If no filenames are provided, the following files are produced:

Options

-p|**--port** *<number>*

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

-d|**--delimiter** *<delimiter-byte>*

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

-o|--oneshot

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

Example

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

```
...
^TU "ms"
SF 1 1dch 9527b66bh
TI 1 1 5
TM 726h
HS 95fch
ME 3 1
NI 6 1
SF 2 10edh 72cbacbch
TM b68h
HS bea4h
^  ...
```

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

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

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

```
trtpd
```

TDF Splitter

Purpose

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

Syntax

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

where:

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

Description

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

Options

-p *<prefix>*

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

-s *<bytes>*

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

Specifies

-v | **-vw**

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

-nt

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

-fopt *<filename>*

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

Code Coverage Report Generator

Purpose

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

Syntax

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

where:

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

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

Description

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

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

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

The Report Generator supports the following coverage type options:

-PROC[**=RET**]

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

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

-CALL

Reports call coverage.

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

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

- **IMPLICIT** or **DECISION** (equivalent) reports implicit blocks (unwritten else and default blocks), as well as statement blocks.

- **LOGICAL** Reports logical blocks (loops, as well as statement and implicit blocks.
- **ATC** Reports asynchronous transfer control (ATC) blocks, as well as statement blocks, implicit blocks, and logical blocks.

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

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

MODIFIED reports modified conditions as well as basic conditions.

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

Explicitly Excluded Options

Each coverage type can also be explicitly excluded.

-NOPROC

Procedure inputs, outputs, or returns are not reported.

-NOCALL

Calls are not reported.

-NOBLOCK

Simple, implicit, or logical blocks are not reported.

-NOCOND

Basic conditions are not reported.

Additional Options

The following options are also available:

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

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

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

Specifies which functions, methods, and procedures are to be reported or not. Use **-SERVICE** to report only the functions, methods and procedures that are explicitly

specified or **-EXSERVICE** to report all functions, methods, and procedures except those that are explicitly specified. Both **-SERVICE** and **-EXSERVICE** cannot be used together.

-TEST=<*test*>{ [**,**<*test*>] } | **-EXTEST=**<*test*>{ [**,**<*test*>] }

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

-OUTPUT=<*file*>

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

-LISTING[=<*directory*>]

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

-NOGLOBAL

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

-BRANCH=COV

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

-SUMMARY=CONCLUSION | **FILE** | **SERVICE**

This option sets the verbosity of the summary:

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

Return Codes

After execution, the program exits with the following return codes

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

All messages are sent to the standard error output device.

Test Process Monitor

Purpose

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

Syntax

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

where:

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

Description

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

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

- **Fields:** Each database contains a fixed number of fields. For example a typical Code Coverage database records.
- **Values:** Each field contains a series of values.
- **Filenames:** Values can be attributed to a filename, such as the name of the file under analysis. This way, the TPM Viewer can display result graphs for any single filename as well as for all files, allowing detailed statistical analysis.

Each field contains a set of values.

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

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

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

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

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

There are 3 steps to using TPM:

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

Creating a Database

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

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

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

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

To create a new metric database with tpm_add:

1. Type the following command:

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

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

Updating a Database

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

To add a new record to this database:

1. Type the **tpm_add** command:

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

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

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

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

Viewing TPM Reports

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

Examples

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

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

The new database is contained in the following files:

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

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

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

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

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

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

If the following command is issued:

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

A new database is created with four fields:

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

Dump File Splitter

Purpose

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

Syntax

atlsplit *<trace_file>*

where:

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

Description

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

Alternatively, you could use the following command line:

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

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

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

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

Uprint Localization Utility

Purpose

The Uprint is a utility that can prove useful if you are experiencing localization issues with PurifyPlus RealTime.

Syntax

uprint
uprint *<hex_min>..<hex_max>*
uprint --mimename
uprint --utf8 *<string>*

where:

- *<hex_min>* and *<hex_max>* specify a range of 16-bit unicode characters expressed in hexadecimal notation.
- *<string>* is a character string encoded in the current locale.

Description

When used with no argument, **uprint** returns the following information about the current locale:

- Mib name
- mimeName
- Locale name

When used with a *<hex_min>..<hex_max>* argument, **uprint** also returns a list of locale-encoded characters from *<hex_min>* to *<hex_max>*.

When used with the **--utf8** option, **uprint** translates a specified locale-encoded *<string>* into a UTF-8 compliant backslashed hexadecimal string for use in C or C++ source code.

When used with the **--mimename** option, **uprint** returns the name of the Unicode Mime encoding.

Examples

The following command returns information about the current locale:

```
>uprint
Mib:111 mimeName:"ISO-8859-15"  locale:"fr_FR@euro"
```

The following command translates the word "éric" into a UTF-8 compliant string:

```
>uprint --utf8 éric
\xc3\xa9\x72\x69\x63
```

Appendices

This section provides extra reference information that may be necessary when using the product.

GUI Macro Variables

Some parts of the graphical user interface (GUI) allow you to specify command lines, such as in the Tools menu or in User Command nodes.

To enhance the usability of this feature, the product includes a macro language, allowing you to pass system and application variables to the command line.

Usage

Macro variables are preceded by **$$** (for example: **$$WSPNAME**).

Macro functions are preceded by **@@** (for example: **@@PROMPT**).

Environment variables are also accessible, and start with **$** (for example: **$DISPLAY**).

When specifying a command line, variables and functions are replaced with their value.

In Windows, when long filenames are involved, it is necessary to add double quotes (" ") around filename variables. For example:

```
"C:\Program Files\Internet Explorer\IEXPLORE.EXE" "$$NODEPATH"
```

Node variables are context-sensitive: the variable returned relates to the node selected in the File or Test Browser. Multiple selections are supported. If a node variable is invoked when there is no selection, no value is returned by the variables.

Macro variables and functions are case-insensitive. For clarity, they are represented in this document in upper case characters.

Language Reference

- Global variables: not node-related, include Workspace and application parameters.

- Node attribute variables: general attributes of a node.
- Functions: return a value to the command line after an action has been performed.

Functions

Functions process an input value and return a result. Input values are typically a global or node variable.

Environment Variable	Description
@@PROMPT('*<message>***')**	Opens a prompt dialog box, allowing the user to enter a line of text. The optional <message> parameter allows you to define a prompt message, surrounded by single quotes (').
@@EDITOR(*<filename>***)**	Opens the product Text Editor.
@@OPEN(*<filename>***)**	Opens *<filename>*. *<filename>* must be a file type recognized by the product. This is the equivalent of selecting **Open** from the **File** menu.

Global Variables

Global variables always return the same value throughout the Workspace.

Environment Variable	Description
$$PRJNAME	Returns the name of the current **.rtp** Project file
$$PRJDIR	Returns the directory name of the current **.rtp** Project file
$$PRJPATH	Returns the absolute path of the current **.rtp** Project file
$$VCSDIR	Returns the local repository for files retrieved from Rational ClearCase, as specified in the ClearCase Preferences dialog box
$$CPPINCLUDES	Returns the directory of C and C++ include files, as specified in the Directories Preferences dialog box
$$PERL	Returns the full command-line to run the PERL interpreter included with the product
$$CLIPBOARD	Returns the text content of the clipboard
$$VCSITEMS	Returns a list of installed configuration management system (CMS) tools

Node Attribute Variables

These variables represent the attributes of a selected node. If no node is selected, these variables return an empty string.

Environment Variable	Description
$$NODENAME	Returns the name of the node. In the case of files, this is the node's short filename
$$NODEPATH	Returns the absolute path and filename of the selected node
$$CFLAGS	Returns the compilation flags
$$LDLIBS	Returns the filenames of link definition libraries
$$LDFLAGS	Returns the flags used for link definition
$$ARGS	Returns all arguments sent to the command line
$$OUTDIR	Returns the name of the product features output directory
$$REPORTDIR	Returns name of the text report output directory
$$TARGETDIR	Returns the absolute path to the current Target Deployment Port
$$BINDIR	Returns the binary directory where the product is installed
$$OBJECTS	Returns a list of **.o** or **.obj** object files generated by the compiler
$$TIO	Returns the name of the current **.tio** trace file generated by Code Coverage
$$TSF	Returns the name of the current UML/SD **.tsf** static file generated by Runtime Tracing
$$TDF	Returns the name of the current UML/SD **.tdf** dynamic file generated by Runtime Tracing
$$TDC	Returns the name of the current Code Coverage **.tdc** correspondence file
$$ROD	Returns the name of the current **.rod** report file
$$FDC	Returns the name of the current **.fdc** correspondence files for Code Coverage

Instrumentation Pragmas

The Runtime Tracing feature allows the user to add special directives to the source code under test, known as *pragma* directives. When the source code is instrumented, the Instrumentor replaces *pragma* directives with dedicated code.

Usage

#pragma attol *<pragma name> <directive>*

Example:

```
int f ( int a )
{
#pragma attol att_insert  if ( a == 0 ) _ATT_DUMP_STACK
  return a;
}
```

This code will be replaced, after instrumentation, with the following line:

```
/*#pragma attol att_insert*/  if ( a == 0 ) _ATT_DUMP_STACK
```

Note Pragma directives are implemented only if the routine in which it is used is instrumented.

Instrumentation Pragma Names

#pragma attol insert *<directive>*

replaced by the instrumentation to be:

```
/*#pragma attol insert*/ <directive>
```

if any of Code Coverage, Runtime Tracing, Memory Profiling or Performance Profiling is/are selected.

#pragma attol atc_insert *<directive>*

replaced by the instrumentation to be:

```
/*#pragma attol atc_insert*/ <directive>
```

if Code Coverage is selected.

#pragma attol att_insert *<directive>*

replaced by the instrumentation to be:

```
/*#pragma attol att_insert*/ <directive>
```

if Runtime Tracing is selected.

#pragma attol atp_insert *<directive>*

replaced by the instrumentation to be:

```
/*#pragma attol atp_insert*/ <directive>
```

if Memory Profiling is selected.

#pragma attol atq_insert *<directive>*

replaced by the instrumentation to be:

```
/*#pragma attol atq_insert*/ <directive>
```

if Performance Profiling is selected.

Code Coverage, Memory Profiling and Performance Profiling Directives

_ATCPQ_DUMP(*<reset>*)

where *<reset>* is 1 if internal tables reset is wanted or 0 if not.

This macro ATCPQ_DUMP does nothing if Code Coverage, Memory Profiling, or Performance Profiling are not selected.

Runtime Tracing Directives

When using this mode, the Target Deployment Package only sends messages related to instance creation and destruction, or user notes. All other events are ignored. See Partial message dump for more information about this feature.

_ATT_START_DUMP
_ATT_STOP_DUMP

These directives activate and deactivate the partial message dump mode.

_ATT_TOGGLE_DUMP

This directive toggles the dump mode on and off. _ATT_TOGGLE_DUMP can be used instead of _ATT_START_DUMP and _ATT_STOP_DUMP.

_ATT_DUMP_STACK

When invoked, this directive dumps the contents of the call stack at that moment.

_ATT_FLUSH_ITEMS

When in Target Deployment Package buffer mode, this directive flushes the buffer. All buffered trace information is dumped. Flushing the buffer be useful before entering a time-critical phase of the trace.

_ATT_USER_NOTE(*<text>***)**

This directive associates a text note to the function or method instance. *<text>* is a user-specified alphanumeric string containing the note text of type *char**. The length of *<text>* must be within the maximum note length specified in the Runtime Tracing Settings dialog box.

Environment Variables

Mandatory Environment Variables

The following environment variables MUST be set to run the product:

- **TESTRTDIR** for the graphical user interface
- **ATLTGT** in the command line interface

Environment Variable List

Environment Variable	Description
TESTRTDIR	A mandatory environment variable that points to the installation directory of the product.
ATTOLSTUDIO_VERBOSE	Setting this variable to **1** forces the product GUI to display verbose messages, including file paths, in the Build Message Window.

Runtime Analysis Features

The Runtime Analysis Features use the following environment variables:

Environment Variable	Description
ATLTGT	A mandatory environment variable that points to the Target Deployment Port directory when you are using the product in the command line interface. When you are using the Instrumentation Launcher or the product GUI, you do not need to set **ATLTGT** manually, as it is calculated automatically.
ATL_TMP_DIR	Indicates the location for temporary files. By default, they are placed in **/tmp** for UNIX or the current directory for Windows.
ATL_EXT_SRC	This variable allows you to instrument additional files with filename extensions other than the defaults (**.c** and **.i**). The **.c**

	extension is reserved for C source files that require preprocessing, while **.i** is for already preprocessed files. All other extensions supported by this variable are assumed to be of source files that need to be preprocessed.
ATL_EXT_OBJ	Lets you specify an alternative extension to **.o** (UNIX) or **.obj** (DOS) for object files.
ATL_EXT_ASM	Lets you specify more than **.s** extension for assembler source files when the compiler offers an option to generate an assembler listing without compiling it to the object file.
ATL_EXT_TMP_CMD	Windows only. Lets you specify an alternative extension to the Windows temporary options file. Defaults to **._@@**.
ATL_EXT_SRCCP	The variable lets you add C++ source file extensions (defaults are **.C**, **.cpp**, **.c++**, **.cxx**, **.cc**, and **.i**) to specify the C++ source files to be instrumented. Extensions **.C** to **.cc** in the list are reserved for source files under analysis. The **.i** extension is reserved for those to be processed, if the **ATL_FORCE_CPLUSPLUS** variable is set to **ON**. Any other extension implies that pre-processing is to be performed.
ATL_FORCE_CPLUSPLUS	If set to **ON**, this variable allows you to force C++ instrumentation whether the file extension is **.c**, **.i**, or any added extension.

C and C++ Instrumentation Launcher

The Instrumentation Launcher uses the following additional variables:

Environment Variable	Description
ATTOLBIN	If set, this variable must contain the path to the Instrumentor binaries. If not, this path is determined automatically from the **PATH** variable. This variable can be useful if the Target Deployment Port has been moved to a non-standard location.
ATTOLOBJ	If set, this variable points to a valid directory where the **products.h** file is generated and the Target Deployment Port (**TP.o** or **TDP.obj**) is compiled. By default, these files are generated in the current directory.
ATL_OVER_SET	This variable must indicate the path to a copy of the **BatchCCDefaults.pl** file if you want to change any Target Deployment Port compilation flags contained in that file.
ATL_EXT_LIB	Lets you specify additional alternative extensions for library files. By default **.a** or **.lib** are used.
ATL_FORCE_C_TDP	If set to **ON**, the **tp.ini** file is used instead of the **tpcpp.ini** file (used

	for C++ language). If the Target Deployment Port supports only C language, the **tp.ini** file is always used.
ATL_OVER_SET	As an alternative to using the --settings of the Instrumentation Launcher, you can copy and modify the *<InstallDir>***/lib/scripts/BatchCCDefaults.pl** file. In this case, set **ATL_OVER_SET** to the directory and filename of the new copy of this file.

Ada Tools

The Ada Link File Generator and Ada Unit Maker use the following additional variables:

Environment Variable	Description
ATTOLCHOP	Selects the default naming convention. The following values can be used: **ATTOLCHOP="APEX"** : for Rational Apex naming **ATTOLCHOP="GNAT"** : for Gnat naming All other values end with a fatal error. By default, Gnat naming is used.
ATTOLALK_EXT	Specifies allowed extensions separated by the semicolon (':') character on UNIX systems and (';') on Windows. By default, the allowed extension list is "**.ada:.ads:.adb**"
ATTOLALK_NOEXT	Specifies forbidden extensions separated by the ':' character on UNIX systems and ';' on Windows. By default, the forbidden extension list is empty.
LD_LIBRARY_PATH	Specifies the location of libraries required by the Ada Link File Generator. By default, these libraries are located in the **/lib** directory of the installation directory.

Setting Environment Variables

Solaris, Linux or HP-UX Platforms

To set an environment variable with a csh shell:

1. Open a shell window

2. Type the following command:
```
setenv <variable> <value>
```

To set an environment variable with a sh, ksh, or Bourne shell:

1. Open a shell window
2. Type the following commands:
```
<variable>=<value>
export <variable>
```

Windows Platforms

To set an environment variable under Windows NT, 2000 or XP:

1. From the **Start** menu, select **Parameters**, **Control Panel**, and double-click **System**.
2. Select the Advanced tab and click Environment variables.
3. Click the **New...** button to add the new environment variable.
4. Click **OK**.

File Types

This table summarizes all the file types generated and used by PurifyPlus RealTime.

File Type	Default Extension	Generated By	Used By
Code Coverage Correspondence File	.fdc	Instrumented application (Code Coverage)	Code Coverage Report Generator
Static Metrics File	.met	C++ Source code Parser C Source Code Parser Ada Source Code Parser Java Source Code Parser	GUI Metrics Viewer
Project File	.rtp	GUI	GUI
Workspace File	.rtw	GUI	GUI
System Testing for C Supervision Script	.spv	User (through CLI only) or Virtual Tester Deployment Wizard	System Testing for C Supervisor

Target Output File	.spt	Target Deployment Port	GUI
UML/SD Dynamic Trace File	.tdf	Instrumented application (Runtime Tracing, Component Testing for C++ and Java)	GUI UML/SD Viewer
Code Coverage Intermediate File	.tio	Instrumented application (Code Coverage)	Code Coverage Report Generator
Memory Profiling for C and C++ Dynamic Trace File	.tpf	Instrumented application (Memory Profiling)	GUI Memory Profiling Viewer
Performance Profiling Dynamic Trace File	.tqf	Instrumented application (Performance Profiling)	GUI Performance Profiling Viewer
Static Trace File	.tsf	C++ Test Script Compiler C and C++ Instrumentor Java Test Report Generator	GUI UML/SD Viewer
Memory Profiling for Java Dynamic Trace File	.txf	Java Instrumented application (Memory Profiling)	GUI Memory Profiling Viewer
Target Deployment Port Customization File	.xdp	TDP Editor	TDP Editor
XML Report File	.xrd	Code Coverage Report Generator	GUI Report Viewer