Rational Software Corporation

Rational® PurifyPlus for Linux

REFERENCE MANUAL

VERSION: 2002 RELEASE 2 - SR1

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

Reference Manual Contents

About Online Documentation

1

The entire documentation set for PurifyPlus for Linux is provided as a full-featured online Help system.

This documentation was designed to be viewed with Netscape Navigator 4.5 or later on other operating systems.

Both environments provide contextual-Help from within the application, a full-text search facility, and direct navigation through the Table of Contents and Index panes on the left side of the Help window.

We welcome any feedback regarding this documentation.

Documentation Updates and Feedback

Documentation Updates

For the most recent documentation updates, please visit the Product Support section of the PurifyPlus for Linux Web site at:

http://www.rational.com/products/purifypluslinux/index.jsp

Feedback

We do our best to provide you with first-rate user documentation, so your feedback is essential for us to improve the quality of our products. If you have any comments or suggestions about our online documentation, feel free to contact us at techpubs@rational.com.

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

Command Line Reference 2

This section provides reference information to help you run the product's runtime analysis features from a command line. This can be useful in complex development environments to perform advanced software analysis in the command line interface.

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** 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** binary is a copy of the **perllauncher** (or **perllauncher.exe**) binary located in <*InstallDir*>**/bin/intel/linux**.

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**.

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/Jav
ic.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.

To install the Javic class for Ant:

1. Download and install Ant v1.4.1 from http://jakarta.apache.org/ant/
2. Set **ANT_HOME** to the installation directory, for example: **/usr/local/jakarta-ant-1.4.1**.
3. Add $ANT_HOME/bin in your PATH
4. 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
```

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
```

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

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

-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=<file>
```

The **-OPP** option allows you to specify an optional definition file. The <file> 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.

```
-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.

```
-JTEST | -NOJTEST
```

The **-JTEST** option provides UML sequence diagram output for Component Testing for Java with Test RealTime. **-NOJTEST** disables this output.

```
-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.

`-TRACE`

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

`-TSFNAME=`***`<file>`***

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

`-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=`***`<file>`***

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.

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 analysis. 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 C Instrumentor (**attolcc1**) supports preprocessed ANSI or K&R C standard source code without distinction.

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.

In C++, the following minor restrictions apply:

- **reinterpret_cast** does not allow casting a pointer to a member of one class to a pointer to a member of another class if the classes are unrelated.
- Template **template** parameters are not accepted.

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

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

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

-FILE specifies the only files that are to be explicitly instrumented, where <*file*> 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 <*file*> is a C source file. All other source files are instrumented. You cannot use this option with the option **-FILE**.

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

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

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

-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.

```
-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 when a trace dump must occur. The **-DUMPCALLING** function is for the C Instrumentor only.

```
-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.

```
-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 coverage report include the number of hits as well as the pass mode information.

Compact mode. 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.

```
-EXCALL=<file>
```

For C only. Excludes calls to the C functions whose names are listed in <file> 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.

`-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 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 elaborates 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>] -- <compilation_command>
```

where:

- <*compilation_command*> is the standard compiler command line that you would use to launch the compiler if you were not running PurifyPlus for Linux instrumentation.
- <*target_deployment_port*> lets you choose the Target Deployment Port to use. <*name*> must be the name of a subdirectory of the **ATLTGT** directory, and must contain object files with the same names as those in **ATLTGT**.
- "**--**" is the command separator preceded and followed by spaces.

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 restricted Target Deployment Ports do not have an **attolcc** binary. In this case, you cannot use Memory Profiling, Performance Profiling, Runtime Tracing and Code Coverage.

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.

```
-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 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.

Code Coverage Options

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

```
-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 coverage report include the number of hits as well as the pass mode information.

Compact mode. 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 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.

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.

Code Coverage Report Generator

Purpose

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

Syntax

```
attolcov {<fdc files>} {<traces>} [<options>]
```

where:

- <*fdc files*> The list of correspondence files for the application under analysis, 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.

```
-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.

```
-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.

TDF Splitter

Purpose

For use with Runtime Tracing. The **.tdf** splitter (**attsplit**) tool allows you to separate large **.tdf** dynamic trace files into smallermore manageablefiles.

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.

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.

Examples

The following examples 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
```

Graphical User Interface

Purpose

The Graphical User Interface (GUI) of the product 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 any text or report file that can be opened by the GUI.
- *<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
```

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 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 runtime analysis feature was 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:

- **.tio Code Coverage report files:** view with Code Coverage Viewer
- **.tdf Dynamic trace files**: view with UML/SD Viewer
- **.tpf Memory Profiling report files:** view with Memory Profiling Viewer
- **.tqf Performance Profiling report files:** view with Performance Profiling Viewer

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 analysis 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 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:

- 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:

- 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
```

Appendices

3

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.

Node variables are context-sensitive: the variable returned relates to the node selected in the Project Explorer. 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 analysis, 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 the section on **Partial Message Dump** in the **User Guide** 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 Linux.
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** 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_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.

Test Process Monitor

The Test Process Monitor uses the following environment variables.

Environment variable	Description
ATTOL_TPM_ROOT	This variable indicates the directory where Test Process Monitor databases are located for a project. **ATTOL_TPM_ROOT** is a mandatory variable and must be set

	when a project is created. It should be a shared directory accessible by all users who work on a project.
ATTOL_TPM_USER	This optional variable specifies the name of the user. If this variable is not set, the Test Process Monitor uses the current user, if possible.

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 Linux systems. By default, the allowed extension list is "**.ada:.ads:.adb**"

ATTOLALK_NOEXT	Specifies forbidden extensions separated by the ':' character on Linux systems. 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 on a Linux Platform

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, Bash, or Bourne shell:

1. Open a shell window
2. Type the following commands:

```
<variable>=<value>
export <variable>
```

File Types

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

File Type	**Default Extension**	**Generated By**	**Used By**
Code Coverage Correspondence File	**.fdc**	Instrumented application (Code Coverage)	Code Coverage Report Generator
Metrics File	**.met**	GUI	GUI Metrics Viewer
Project File	**.rtp**	GUI	GUI
Workspace File	**.rtw**	GUI	GUI
Target Output File	**.spt**	Target Deployment Port	GUI
UML/SD Dynamic Trace File	**.tdf**	Instrumented application (Runtime Tracing)	GUI UML/SD Viewer
Code Coverage Intermediate File	**.tio**	Instrumented application (Code Coverage)	Code Coverage Report Generator
Memory Profiling 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 and C++ Instrumentor	GUI UML/SD Viewer
Target Deployment Port Customization File	**.xdp**	TDP Editor	TDP Editor