Parasoft Test Plugin For Maven and Ant

Parasoft Corporation
101 E. Huntington Drive, 2nd Floor
Monrovia, CA 91016
Phone: (888) 305-0041
Fax: (626) 305-9048
E-mail: info@parasoft.com
URL: www.parasoft.com

Visit www.parasoft.com/eula for licensing and usage information.

Table of Contents

Parasoft Test Maven Plugin

This topic covers the Parasoft Test Maven plugin that allows you to configure Parasoft testing and build analysis actions to be invoked from Maven. It assumes a basic familiarity with Maven usage.

Sections include:

- Prerequisites
- Setup
- Configuration
- Available Goals
- Environment Variables
- Usage Tips

For general help on configuring Maven plugins, see http://maven.apache.org/guides/mini/guide-configuring-plugins.html

Prerequisites

The Parasoft Test Maven plugin requires:

- Maven 2.0
- JDK 1.5

There are no minimum requirements for disk space or memory space.

Setup

Before you can use the Parasoft Test Maven plugin, you need to add the `http://build.parasoft.com/maven` repository to your POM or repository manager. For instance:

```
<project>
   ...
  <pluginRepositories>
      <pluginRepository>
          <id>Parasoft</id>
          <url>http://build.parasoft.com/maven</url>
      </pluginRepository>
  </pluginRepositories>
   ...
</project>
```

If you will be performing Jtest unit testing from Maven, you also need to add the build artifacts in the `http://build.parasoft.com/maven/Parasoft/jtest` repository to your POM or repository manager. For instance:

```
<project>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>Parasoft</groupId>
      <artifactId>jtest</artifactId>
      <version>9.5.0/version> <!-- See below for available versions -->
    </dependency>
    ...
  </dependencies>
   ...
```

```
  <repositories>
      <repository>
          <id>Parasoft</id>
          <url>http://build.parasoft.com/maven/</url>
      </repository>
  </repositories>
   ...
</project>
```

The Jtest versions you can specify in `<version>` are:

- 9.0.0.20100729 (for Jtest 9.0.x)
- 9.1.3 (for Jtest 9.1.x)
- 9.2.3 (for Jtest 9.2.x)
- 9.4.3 (for Jtest 9.4.x)
- 9.5.0 (for Jtest 9.5.x)

Configuration

There are three main ways to configure execution of the Parasoft plugin:

- Extend the POM
- Configure it from the command line
- Set a profile switch in the POM file, then use that switch at the command line

Option 1: Extend the POM

If you extend the POM, the specified goals will always be run at a particular phase of the build lifecycle. This allows you to specify your configuration once, and have it used every time that Maven builds the configured project.

First, specify the plugin version. For example:

```
<project>
  ...
  <build>
    <!-- To define the plugin version in your parent POM -->
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>Parasoft</groupId>
          <artifactId>maven-parasoft-plugin</artifactId>
          <version>3.0</version>
        </plugin>
        ...
      </plugins>
    </pluginManagement>
    <!-- To use the plugin goals in your POM or parent POM -->
    <plugins>
      <plugin>
        <groupId>Parasoft</groupId>
        <artifactId>maven-parasoft-plugin</artifactId>
        <version>3.0</version>
      </plugin>
      ...
    </plugins>
  </build>
  ...
```

```
</project>
```

Next, specify goals and configuration parameters. For example, the following runs the `eclipse` and `jtest` goals during the `test` lifecycle phase (Jtest is run with the built-in Static Analysis Test Configuration):

```
<project>
  ...
  <build>
      <plugins>
          <plugin>
              <groupId>Parasoft</groupId>
              <artifactId>maven-parasoft-plugin</artifactId>
              <version>3.0</version>
              <configuration>
                  <config>builtin://Static Analysis</config>
              </configuration>
              <executions>
                  <execution>
                     <phase>test</phase>
                      <goals>
                          <goal>eclipse</goal>
                          <goal>jtest</goal>
                      </goals>
                  </execution>
              </executions>
          </plugin>
  </build>
  ...
</project>
```

You can run multiple goals; for instance, a team using Jtest might want to generate a local settings file, then run static analysis and unit testing.

Option 2: Configure the Plugin from the Command Line

The most direct way to execute the plugin is to configure the plugin execution details completely from the command line. For example, to invoke the `jtest` goal from the sample POM (above), you would go to the project's command line and use

```
mvn ... Parasoft:maven-parasoft-plugin:jtest -Dparasoft.config="builtin://Static Analysis"
```

If you choose this approach, you will need to specify the appropriate options during every command line invocation.

Simplifying the Command Line

To simplify calling the plugin from the command line, we strongly recommend that you modify the `settings.xml` file (`~/.m2/settings.xml`) to include a Parasoft plugin group that specifies the appropriate groupids, artifactids, and goals. For example:

```
<settings>
    <pluginGroups>
          <pluginGroup>Parasoft</pluginGroup>
    </pluginGroups>
</settings>
```

This will tell Maven where to search for the plugin—allowing you to shorten the above command line invocation to

```
mvn ... parasoft:jtest -Dparasoft.config="builtin://Static Analysis"
```

For general help on using Maven plugin prefix mappings, see http://maven.apache.org/guides/introduction/introduction-to-plugin-prefix-mapping.html.

Another tip for simplifying your command line invocation is to specify some options in the POM. For instance, if you are invoking `jtest-global` from the command line, you could set the desired test configuration, localsettings file, etc. in the POM.

Option 3: Configure the Plugin as a Profile Switch

A third option is to add a switch within a profile, then run with the appropriate switch when you want to apply one of the Parasoft goals. You can configure any number of switches with different goals and parameters. This approach is useful for when you want to set plugin execution details in the POM, but you do not want it permanently attached to a lifecycle phase. It is also useful if you want configure a set of different plugin configurations, then select the appropriate one at the command line invocation.

For example, you can add a `jtest` profile to the build as follows:

```
<profiles>
     <profile>
     <id>jtest</id>
         <build>
             <plugins>
                 <plugin>
                     <groupId>Parasoft</groupId>
                     <artifactId>maven-parasoft-plugin</artifactId>
                     <configuration>
                         <config>builtin://Static Analysis</config>
                     </configuration>
                     <executions>
                         <execution>
                             <phase>test</phase>
                             <goals>
                                 <goal>eclipse</goal>
                                 <goal>jtest</goal>
                             </goals>
                         </execution>
                     </executions>
                 </plugin>
             </plugins>
         </build>
     </profile>
 </profiles>
```

You can then use `mvn install -Pjtest` to run the `jtest` profile.

For general help on using Maven profiles, see http://maven.apache.org/guides/introduction/introduction-to-profiles.html.

Available Goals

See the plugin reference page.

Environment Variables

Environment variables can be used for options that will remain constant for all builds on a specific machine.

Each Parasoft Test product has its own OPTS variable:

- CPPTEST_OPTS
- DOTTEST_OPTS
- JTEST_OPTS
- SOATEST_OPTS

For options that will be used for all Parasoft Test products, use PARASOFT_OPTS.

For options that will be used only in the project import phase, use IMPORT_OPTS.

These options are typically used when running the plugin via the command line—because some parameters can't be set in the command line (like those of type List). For example, the `vmargs` parameter of `parasoft:jtest i`s of type List and can be used for things like changing the user name or increasing Java memory.

For example:

1. Set the Jtest user name.
2. Create the environment variable JTEST_OPTS and set it to -J-Duser.name=user1

Usage Tips

This section provides tips for using the goals described in the plugin reference page.

Preparing for Jtest or SOAtest Execution

In order to execute Jtest, you need to have `.classpath` and `.project` files that work in the server environment. In order to execute SOAtest, you need to have a `.project` file that works in the server environment. To generate the files that Eclipse requires, use `parasoft:eclipse`.

If you have decided to store localsettings for Jtest and SOAtest execution on Concerto (as described in the Concerto documentation)—and/or if you are specifying localsettings options in the POM—`parasoft:localsettings` will generate a localsettings file with these options, then save it in the project so that it can easily be used for your Maven-driven tests.

Running Jtest Once for a Multi-Module Project

If you have a multi-module project (with modules that get built individually), the `parasoft:jtest` goal will run once for each project module. If you want a single Jtest run to analyze your entire project (for instance, for more accurate application of project-wide static analysis rules or to optimize execution time), use `parasoft:jtest-global` instead of `parasoft:jtest`.

Note that `jtest-global` must be invoked at the command line since it can’t be bound to any specific Maven lifecycle phase.

For Monitoring the Build

The `parasoft:compile` goal executes the Parasoft Build Monitor, which analyzes your build and sends Parasoft Concerto data regarding build warnings, errors, and the number of files successfully compiled. This is performed as part of the compile lifecycle phase.

When using this goal, be sure to specify the Concerto host, port, and project name.

To use this goal, go to your project’s POM file and add the following plugin instruction to your `<build><plugins>` node:

```
<plugin>
    <groupId>Parasoft</groupId>
```

```
        <artifactId>maven-parasoft-plugin/artifactId>
        <executions>
            <execution>
                <goals>
                    <goal>compile</goal>
                </goals>
            </execution>
        </executions>
        <configuration>
            <source>1.4</source>
            <target>1.4</target>
            <enableGRS>true</enableGRS>
            <showDeprecation>true</showDeprecation>
            <grsServerName>server.parasoft.com</grsServerName>
            <grsServerPort>32323</grsServerPort>
            <grsAttributes>
                <attribute>test1=value1</attribute>
                <attribute>test2=value2</attribute>
                <attribute>test3=value3</attribute>
            </grsAttributes>
        </configuration>
    </plugin>
```

To invoke Build Monitor with a compiler, run `mvn parasoft:compile`.

Notes:

- This extends the official compiler—enabling it to send build results to Parasoft Concerto.
- Currently, Build Monitor only supports one source root (`compileSourceRoots`). Therefore, all project components are related to the first SourceRoot entry.

Parasoft Test Ant Plugin

This topic covers the Parasoft Test Ant plugin that allows you to configure Parasoft testing and build analysis actions to be invoked from Ant. It assumes a basic familiarity with Ant usage.

Sections include:

Prerequisites

The Parasoft Test Ant plugin requires:

- JDK 1.5
- Ant 1.6.2

There are no minimum requirements for disk space or memory space.

Setup

Before you can use the Parasoft Test Ant plugin, you need to get the jar file from http://build.parasoft.com/docs/overview.html, then do one of the following:

- Add the jar file to your Ant library.
- Add the jar file to your Ant repository.
- Add the jar file's directory as a command line `-lib` argument.

Configuration

To configure execution of the Parasoft plugin:

1. Create a new file (e.g., `parasoft.xml` or `jtest.xml)`. In this file, you will import your original Ant build file as well as specify how you want Ant to run the Parasoft plugin.
2. Start that file by defining the project and calling the Parasoft Ant library. Add one project definition for each logical project in your build script. Each project defined here will be reported to Parasoft Concerto as a separate project. For example:
   ```
   <project name="Parasoft Test Build" default="parasoft-test" xmlns:parasoft="antlib:com.parasoft.antlib">
   ```
   If the `build.xml` being imported has its project definition basedir set to anything other than the current directory ".", add the same basedir to this configuration. For example:
   ```
   <project name="Parasoft Test Build" default="parasoft-test" xmlns:parasoft="antlib:com.parasoft.antlib" basedir="..">
   ```
   (If omitted, it defaults to ".")
3. Import the original Ant `build.xml` file. For example:
   ```
   <import file="build.xml"/>
   ```

This step is not required for running SOAtest.

4. Add a target for the Parasoft task you want executed and use dependencies to specify the task to integrate with (e.g., `build`). For example:
   ```
   <target name="parasoft-test" depends="build">
   ```

   For SOAtest, you do not need to use dependencies to specify the task to integrate with.

5. Configure the target for the task. Be sure to specify

   a. Any required or desired task options described in the plugin reference page
   b. (*Not required for SOAtest)* Either `javacref` or `monitorref` (described below) to specify which compilation targets you want to operate on.

   For example:

   ```
   <!-- To use and configure Parasoft Test Plugin -->
       <parasoft:jtest config="builtin://Static Analysis">
           <!-- javacref refers to the ID of the specific javac in build.xml -->
          <projectDescription basedir="." javacref="javacId" overwrite="true"/>
       </parasoft:jtest>
   ```

6. Repeat steps 4 and 5 until all of your desired targets are configured.

There are two main ways to specify the `javac` targets you want to operate on:

- To automatically record the project's `javac` targets, use the `parasoft:monitor` task. As your project builds, the monitor will record every call to the `javac` compiler. You can then give the monitor an ID then reference the monitor in your Parasoft script. If you are writing an Ant script in a very dynamic way (e.g., using `<antcall/>` to call the `javac` compiler multiple times with different arguments), using the monitor is highly recommended. See Using the Monitor below for more details.
- To have the plugin operate on `javac` calls specified in `build.xml`, add `refids` for those calls, then use `javacref` to specify the `refid`(s) you want to operate on. This option gives you more control over project configuration than the `monitor` option provides. See Using refids below for more details.

Using the Monitor

To use the monitor:

1. Modify your Parasoft Ant build script so that `parasoft:monitor` is called prior to compilation.
2. In the Parasoft task definition in your Parasoft Ant build script, specify `monitorref="your.monitor"`

For instance, the following configures a new `parasoft-init` target for the monitor, then uses a dependency to ensure that the main compilation task (`compiler`) executes only after the `monitor` task completes. Note how it defines, then references, the `my.monitor` id:

```
<target name="parasoft-init">
    <parasoft:monitor id="my.monitor"/>
</target>

<target name="jtest" depends="parasoft-init,clean, compiler">
   <echo>Running Jtest</echo>

    <!== build main sources -->
```

```
<parasoft:jtest config="builtin://Static Analysis" overwrite="true">
    <projectDescription monitorref="my.monitor" overwrite="true" /"
</parasoft:jtest>
```

Using refids

To use `refids`:

1. In your original Ant build script, go to the project definitions and define IDs for your `javac` targets.
2. In the Parasoft task definition in your Parasoft Ant build script, specify which `refid`s you want the task to operate on:
    - To operate on specific `javac` calls, use `javacref ="javacId"` or `javacref ="javacId, javacId2, javacId3, .."`.
    - To have the plugin operate on all `javac` calls specified in `build.xml`, use `javacref="all"`.
    - To have the plugin operate on all `javac` calls that have `refid`s, use `javacref="allrefs"`.

For example, you might update your `build.xml` file to include `refid`s as follows:

```
<javac target="${jdk.version.class}"
      source="${jdk.version.source}"
      ...
      id="main.javac"
      ...
      .../>
```

Then, you would add the following to the target description in your `parasoft.xml` file:

```
<target name="parasoft-test" depends="build">
     ...
     <!-- To use and configure Parasoft Test Plugin -->
     <parasoft:jtest config="builtin://Static Analysis">
         <!-- javacref refers to the ID of the specific javac in build.xml -->
         <projectDescription basedir="." javacref="main.javac" overwrite="true"/>
     </parasoft:jtest>
     ...
 </target>
```

Available Tasks

See the plugin reference page.

Environment Variables

Environment variables can be used for options that will remain constant for all builds on a specific machine.

Each Parasoft Test product has its own OPTS variable:

- CPPTEST_OPTS
- DOTTEST_OPTS
- JTEST_OPTS
- SOATEST_OPTS

For options that will be used for all Parasoft Test products, use PARASOFT_OPTS.

For options that will be used only in the project import phase, use IMPORT_OPTS.

These options are commonly used for tasks like changing user names or increasing Java memory.

For example:

1. Set the Jtest user name.
2. Create the environment variable JTEST_OPTS and set it to -J-Duser.name=user1

Usage Tips

This section provides tips for using the tasks described in the plugin reference page.

Preparing for Jtest or SOAtest Execution

In order to execute Jtest, you need to have `.classpath` and `.project` files that work in the server environment. In order to execute SOAtest, you need to have a `.project` file that works in the server environment. To generate the files that Eclipse requires, use the `parasoft:eclipse` task.

If you have decided to store localsettings for Jtest and SOAtest execution on Concerto (as described in the Concerto documentation)—and/or if you are specifying localsettings options in the Ant script—`parasoft:localsettings` will generate a localsettings file with these options, then save it in the project so that it can easily be used for your Ant-driven tests.

For Monitoring the Build

The `parasoft:monitor` task (described above in Using the Monitor) also executes the Parasoft Build Monitor, which analyzes your build and sends Parasoft Concerto data regarding build warnings, errors, and the number of files successfully compiled. *This must be performed before compilation.*

When using this task for Concerto build monitoring, be sure to specify the Concerto host, port, and project name.

For backwards compatibility, the Parasoft Test Ant plugin also supports the previous (now deprecated) mode of configuring Build Monitor. This is described in "Configuring Parasoft Build Monitor with Ant (Deprecated)", page 20.

Running Several Different Configurations

Here is a sample Ant script that can run several different Jtest configurations (as separate targets):

```
<?xml version="1.0"?>
<project name="Jtest Build" default="jtest" xmlns:parasoft="antlib:com.parasoft.antlib">
    <import file="build.xml"/>

    <property name="concerto" value="developers.parasoft.com"/>
    <property name="projectName" value="Default Project"/>
    <property name="config" value="builtin://Static Analysis"/>

    <!-- Installs the parasoft monitor to track the build -->
    <target name="parasoft-init">
       <parasoft:monitor id="parasoft.monitor" concertoHost="${concerto}" concertoPort="8080"
projectName="${projectName}"/>
    </target>

    <!-- Creates the localsettings file used by parasoft tools, can optionally retrieve set-
tings from Concerto (4.5 or higher) -->
    <target name="localsettings">
        <parasoft:localsettings concertoHost="${concerto}" concertoPort="8080" project-
Name="${projectName}">
            <additionalProperty key="scope.author" value="false"/>
```

```
            <additionalProperty key="scope.local" value="true"/>
            <additionalProperty key="report.mail.enabled" value="true"/>
        </parasoft:localsettings>
    </target>

    <!-- runs Jtest by statically analyzing the build.  id attributes must be added to javac
targets. A separate project is setup for each javac -->
    <target name="jtest-multiple-projects" depends="parasoft-init, clean, compile, localset-
tings">
        <echo>Running Jtest Using Multiple Projects</echo>
        <parasoft:jtest config="${config}">
            <projectDescription javacref="main.javac" overwrite="true"/>
            <projectDescription javacref="submodule.javac" overwrite="true"/>
        </parasoft:jtest>
    </target>

    <!-- runs Jtest by statically analyzing the build.  id attributes must be added to javac
targets. A single project is setup which combines the referenced javac targets -->
    <target name="jtest-single-project" depends="parasoft-init, clean, compile, localsettings">
        <echo>Running Jtest Using Single Project</echo>
        <parasoft:jtest config="${config}">
            <projectDescription javacref="main.javac, submodule.javac" overwrite="true"/>
        </parasoft:jtest>
    </target>

    <!-- runs Jtest by statically analyzing the build.  Collects all javac targets from build
file -->
    <target name="jtest-single-project-all" depends="parasoft-init, clean, compile, localset-
tings">
        <echo>Running Jtest Using With All javac targets</echo>
        <parasoft:jtest config="${config}">
            <projectDescription javacref="all" overwrite="true"/>
        </parasoft:jtest>
    </target>

    <!-- runs Jtest by statically analyzing the build.  Collects all javac references from
build file -->
    <target name="jtest-single-project-allrefs" depends="parasoft-init, clean, compile, local-
settings">
        <echo>Running Jtest Using With All javac targets that have references</echo>
        <parasoft:jtest config="${config}">
            <projectDescription javacref="allrefs" overwrite="true"/>
        </parasoft:jtest>
    </target>

    <!-- runs Jtest by analyzing the build dynamically.  The monitor will collect information
during the build to setup and run Jtest -->
    <target name="jtest-with-monitor" depends="parasoft-init, clean, compile, localsettings">
        <echo>Running Jtest With Monitor</echo>
        <parasoft:jtest config="${config}">
            <projectDescription monitorref="parasoft.monitor" overwrite="true"/>
        </parasoft:jtest>
    </target>

</project>
```

Examples

Automating Jtest with Ant

In the example below, the Parasoft Ant script imports an existing build.xml and calls Jtest to execute static analysis on the build.

```
<!-- Project definition and call to Parasoft Ant library -->
<project name="Parasoft Test Build" default="parasoft-test"
xmlns:parasoft="antlib:com.parasoft.antlib">
    <!-- To import the original Ant build.xml file and specify the task to integrate to -->
    <import file="build.xml"/>
    <target name="parasoft-test" depends="build">
        <!-- To run Jtest on the specified project directory in projectDescription, using the
configuration in config and Jtest in jtestHome -->
        <parasoft:jtest config="builtin://Static Analysis" jtestHome="C:\Program
Files\Parasoft\Jtest\9.5">
            <projectDescription basedir="."/>
        </parasoft:jtest>
    </target>
</project>
```

In the next example, the Parasoft Ant script imports an existing build.xml whose basedir is set to a different directory (for example, `basedir="../sources"`):

```
<!-- Project definition and call to Parasoft Ant library -->
<project name="Parasoft Test Build" default="parasoft-test"
xmlns:parasoft="antlib:com.parasoft.antlib">
    <!-- To import the original Ant build.xml file and specify the task to integrate to -->
    <import file="build.xml"/>
    <target name="parasoft-test" depends="build">
        <!-- To run Jtest on the specified project directory in projectDescription, using the
configuration in config and Jtest in jtestHome -->
        <parasoft:jtest config="builtin://Static Analysis" jtestHome="C:\Program
Files\Parasoft\Jtest\9.5">
            <projectDescription basedir="."/>
        </parasoft:jtest>
    </target>
</project>
```

Automating SOAtest with Ant

In the example below, the Parasoft Ant script calls SOAtest to run the .tst files with the "Run Web Functional Tests in Browser Specified by Test" Test Configuration.

```
<!-- Project definition and call to Parasoft Ant library -->
<project name="Parasoft Test Build" default="parasoft-test"
xmlns:parasoft="antlib:com.parasoft.antlib" basedir="../sources">
    <target name="soatest-test">
        <!-- To generate .project file needed to run SOAtest if it does not exist -->
        <parasoft:eclipse/>
        <!-- To run SOAtest with the selected configuration -->
        <parasoft:soatest config="builtin://Run Web Functional Tests in Browser Specified by
Test"
soatestHome="C:\Program Files\Parasoft\SOAtest\9.4"/>
    </target>
</project>
```

Note that this differs from a Jtest build file in that you do not need to:

- Import the original build.xml file.
- Use dependencies to specify the task to integrate with (e.g., `build`).
- Use either `javacref` or `monitorref` to specify which compilation targets you want to operate on.

Configuring Parasoft Build Monitor with Ant (Deprecated)

This topic covers how to configure the previous generation of Parasoft build monitoring with Ant. It is still supported for backwards compatibility, but the plugin described in "Parasoft Test Ant Plugin", page 14 is now the recommended method of build monitoring with Ant.

Sections include:

Setting Properties

Set all properties in the `ANT_OPT` environment variable so that the Ant script can detect all properties.

For Bash:

```
ANT_OPTS="-Dbuildmonitor.server.name=grsserver.example.com"
```

For Windows:

```
SET ANT_OPTS="-Dbuildmonitor.server.name=grsserver.example.com"
```

Build Monitor Logger

The Build Monitor Logger automatically detects the project name and project root from the Ant script.

Project Name

The project name must be set in the `<project>` node. If the project name is not defined in the option file or `buildmonitor.projectroot`, the application obtains the project name from the Ant script.

Build Monitor requires a name attribute from Ant. If Ant's `<project>` node does not have a name, it will be set to `"Default Project"`.

Although Ant allows for this, it is not a recommended practice. All Ant project nodes require a set name attribute, for example

```
<project name="XXXX" basedir=".">
...
</project>
```

Project Root

When Ant contains multiple `javac` build tasks, it is difficult to set the project root. If this is the case, do not set the project root in the option file or `-Dbuildmonitor.projectname`. `javac` tasks contain a `src` attribute and Build Monitor will use the `src` value to set the project root.

Running Ant

You can run Ant with the following command, which overrides the default logger for Ant and triggers Build Monitor to generate the build:

```
ant <user options> -logger proserve.tools.buildmon.BuildMonLogger
<build task command>
```

Configuring CruiseControl *(Optional)*

A logger is available in Build Monitor, which Ant builds within CruiseControl. Here's how it works:

The CruiseControl Ant builder overrides the default Ant logger and uses an XML logger to save output from Ant. Next, CruiseControl reads the XML output, and then displays it on the dashboard. By design, CruiseControl always expects `log.xml` to be in the Ant execution directory. Further configuration is needed for Build Monitor integration:

Important for Continuum users! *Continuum continuous build system created by Apache uses standard DefaultLogger for Ant. Normal proserve.tools.buildmon.BuildMonLogger would work just fine with Continuum.*

Note: *Build Monitor for Ant has been tested with CruiseControl 2.7.2 and above. Therefore, it is not recommended to use with previous versions of CruiseControl.*

To integrate Ant projects under CruiseControl with Build Monitor:

1. Modify the project configuration scheduler in config.xml from this:

   ```
   <schedule interval="300">
       <ant anthome="apache-ant-1.7.0" buildfile="projects/${project.name}/
   build.xml"/>
   </schedule>
   ```

   To this:

```
<cruisecontrol>
...
  <project name="j2ssh">
    <schedule showprogress="true">
      <ant uselogger="true" target="clean build"
              loggerclassname="proserve.tools.buildmon.BuildMonLogger"
              buildfile="jtest-build.xml"
              showprogress="true"
              antworkingdir="projects/${project.name}"
              showantoutput="false"
              anthome="${CC_HOME}\apache-ant-1.7.0"
              usedebug="false" />
    </schedule>
</cruisecontrol>
```

2. Set the `showantoutput` attribute to `false` to disable it when you are using Build Monitor.

   It is important to do this because the `showantoutput` (AntOutputLogger from CruiseControl) option triggers a failure when you use Build Monitor with it.

3. Turn ON the `showprogress` option.

   This option triggers CruiseControl to include antProgressXmlLogger. AntProgressXmlLogger will log the project build status for CruiseControl while BuildMonLogger performs its tasks.

   Note: *CruiseControl 2.7.1 and older do not support the* `showAntOutput` *attribute.* `Show-Progress` *is optional for CruiseControl 2.7.1.*

4. (Optional) Open and review the `cruisecontrol.log` file to check Build Monitor output.

   Tip! *It is good practice to start your Ant project under your project directory so that any log files can use your project root directory instead of the CruiseControl home directory. Be sure to set*

the `jvmarg` *node to optional and reflective of how to pass any property options for Build Monitor.*

Configuring Eclipse AntRunner *(Optional)*

You can integrate Build Monitor with Eclipse AntRunner.

To integrate Eclipse AntRunner with Build Monitor:

1. Copy the `proserve-base.jar` and `proserve-plugin.jar` files from the `$PROSERVE_HOME/jar` directory to:

   ```
   <eclipse install directory>/plugins/org.apache.ant_<version>/lib
   ```

2. Add `Build Monitor -logger` argument to the target after the AntRunner application:

   ```
   <arg value="-logger" />
   <arg value="proserve.tools.buildmon.BuildMonLogger" />
   ```

3. Add Build Monitor JVMArgs to specify Concerto Report Center properties:

   ```
   <jvmarg value="-Dbuildmonitor.grs.server=leech" />
   <jvmarg value="-Dbuildmonitor.grs.port=32323" />
   <jvmarg value="-Dbuildmonitor.attributes=\"ProjectName=samplePDE\"
   ```

   For details about Build Monitor command line options available to use in Ant, see “Command Line Options”, page 22.

Example

Following is a usage example that includes the steps above—showing how to integrate Eclipse AntRunner with Build Monitor:

```
<project name="Build Monitor Example" default="-build.internal">
  <target name="-build.internal">
    <java jar="C:\Program Files\eclipse-all-in-one-SDK-3.3.0\plugins\org.eclipse.equi-
nox.launcher_1.0.0.v20070606.jar"
fork="true" >
      <arg value="-application" />
      <arg value="org.eclipse.ant.core.antRunner" />
      <arg value="-logger" />
              <arg value="proserve.tools.buildmon.BuildMonLogger" />
      <arg value="-data"/>
      <arg value="C:\Documents and Settings\wes\workspace1"/>
      <arg value="-buildfile" />
      <arg value="C:\Documents and Settings\wes\workspace1\samplePDE\build.xml"/>
      <arg value="clean" />
      <arg value="build" />
      <jvmarg value="-Dbuilder=C:\Documents and Settings\wes\workspace1\samplePDE" />
      <jvmarg value="-Dbuildmonitor.grs.server=leech" />
              <jvmarg value="-Dbuildmonitor.grs.port=32323" />
              <jvmarg value="-Dbuildmonitor.attributes=\"ProjectName=samplePDE\"" />
    </java>
  </target>
</project>
```

Command Line Options

The following table lists and explains command line Build Monitor options that you can execute in Ant, if necessary:

Command	Task
`buildmonitor.grs.server`	*(Required)* Sets Report Center server name.
`buildmonitor.grs.port`	*(Required)* Sets Report Center server port. The default is set to 32323.
`buildmonitor.alternatelogger`	*(Optional)* Use to specify an alternate Ant BuildLogger for Build Monitor to use, such as `MailLogger`. By default, Build Monitor uses `org.apache.tools.ant.DefaultLogger`. Users can define their own loggers via typical passing of the `-logger` option to Ant, which is why Build Monitor supports loggers aside from its own. By adding this property to the Ant command line, Build Monitor loads the specified logger and uses it rather than the default logger; for example: `buildmonitor.alternatelogger=org.apache.tools.ant.listener.MailLogger <with other options for MailLogger>`

`buildmonitor.attributes`	*(Optional)* Sets Concerto Report Center user attributes. The format should be as follows:
	`name:value,name2:value2...`
	
	For example:
	
	`buildmonitor.attributes:ProjectName:buildmon,version:1.1`
	`buildmonitor.attributes:TestAttribute:test`

`buildmonitor.compiler`	*(Optional)* Name of the compile task. The javac Ant edition begins logging compile information during the task that is matched to the compiler. The default value for this option is `javac`.
`buildmonitor.nightly`	*(Optional)* If set to `true`, "Nightly" is added as an attribute to the **Test Group Properties Filter** in Concerto Report Center. If left blank or set to `false`, it is not included as an attribute.

Command	Task
`buildmonitor.projectname`	*(Optional)* Name of the project, which is used for the Concerto Report Center project attribute. If property is not defined, Build Monitor will pull the value from Ant's `${ant.project.name}` property setting.
`buildmonitor.projectroot`	*(Optional)* Root path of the project. If the property is not defined, Build Monitor will pull the value from Ant's `${basedir}` property setting.
`buildmonitor.verbose`	*(Optional)* `true` means verbosity is on. `false` means verbosity is off. This command is only useful for debugging Build Monitor issues. It provides more verbose output to the console so that you can see what is going on when Build Monitor is executed.
`eclipse.running`	*(Optional)* Set this system property if running Eclipse's JAVAC compiler for javac task. Eclipse's javac compiler prints warning and error messages in a different format—bypassing `eclipse.running=true` as the JVM argument (ANT_OPTS is the environment variable for the Ant script).

Example

Following is a usage example that includes the command line options listed in the table above.

Important! The `buildmonitor.attributes` option format has been changed beginning with Build Monitor 3.0.1:

- Commas (,) replace semi-colons (;) to separate attributes when multiple attributes are listed for Parasoft Concerto.
- Colons (:) replace equal signs (=) for key and value assignments.

Although the previous format style (semi-colons and equal signs) will still work, the new format style (commas and colons) is recommended. Mixing these format styles will NOT work!

***Note:** The compiler should be set as your compile task (by default, it is `javac`).*

```
<?xml version:"1.0"?>
<project name:"master project" basedir:"." default:"do-all">
    <property name:"src.dir" value:"."/>
    <property name:"buildmonitor.grs.server" value:"leech.parasoft.com"/>
         <property name:"buildmonitor.grs.port" value:"32323"/>
         <property name:"buildmonitor.verbose" value:"true"/>
         <property name:"buildmonitor.compiler" value:"javac"/>
    <target name:"clean">
        <delete dir:"j2ssh"/>
        <delete dir:"jEdit-4.3pre2"/>
        <delete dir:"WebApp"/>
    </target>
```

```
    <target name:"shadow">
        <exec dir:"${src.dir}" executable:"svn" failonerror:"true">
            <arg value:"checkout"/>
            <arg
value:"https://svn.parasoft.com/svn/proserve/demo/branches/aep/DummyCode/java/j2ssh/"/>
            <arg value:"j2ssh"/>
        </exec>
        <exec dir:"${src.dir}" executable:"svn" failonerror:"true">
            <arg value:"checkout"/>
            <arg
value:"https://svn.parasoft.com/svn/proserve/demo/trunk/DummyCode/java/jEdit-4.3pre2/"/>
            <arg value:"jEdit-4.3pre2"/>
        </exec>
    </target>

    <target name:"build-all">
        <ant antfile:"build.xml" dir:"j2ssh" target:"clean"/>
        <ant antfile:"build.xml" dir:"j2ssh" target:"build">
            <property name:"buildmonitor.attributes" value:"Test:BMTest,ProjectName:j2ssh"/>
        </ant>
        <ant antfile:"build.xml" dir:"jEdit-4.3pre2" target:"clean"/>
        <ant antfile:"build.xml" dir:"jEdit-4.3pre2" target:"compile">
            <property name:"buildmonitor.attributes"
value:"Test:BMTest,ProjectName:jEdit-4.3pre2"/>
        </ant>
    </target>

    <target name:"do-all" depends:"clean,shadow,build-all">
        <delete dir:"j2ssh"/>
        <delete dir:"jEdit-4.3pre2"/>
    </target>

</project>
```