Monitoring Java Applications

This topic explains how to configure monitoring for any Java application. When a properly-configured Event Monitor tool is placed in the beginning of a test suite that includes tests which invoke that Java application (directly or indirectly), it will receive and visualize the Java events that take place.

Sections include:

Parasoft Jtest Installation is Required

SOAtest provides Java runtime error detection and Java event monitoring capabilities via integration with Parasoft Jtest, which facilitates a broad range of test and analysis practices for Java.

Before you use SOAtest's runtime error detection and/or Java event monitoring capabilities, ensure that Jtest is installed on your machine and that the Jtest Connect license option is enabled. See Licensing.

Contact your Parasoft representative to access Jtest. This cross-product capability requires Jtest and SOAtest to be installed in one of the following ways:

- Jtest and SOAtest plugins installed into the same Eclipse instance, or
- Jtest and SOAtest installed via p2 updatesite archives (see Eclipse p2 Update Site Installation for details).

Why Monitor Java Applications?

By monitoring instrumented Java applications, you can gain visibility into the application's internal behavior as functional tests execute. This allows you to better identify the causes of test failures as well as enrich the use case validation criteria in your regression tests. In addition to validating the messages returned by the system and the intermediate messages/steps monitored via the ESB, you can also validate events within the Java application being invoked. For example, you can validate that an EJB method call or a remote call to another system is taking place with the expected parameters.

Application Configuration

To configure the application for monitoring, you need to instrument it with Parasoft's monitoring agent. To do this:

1. Copy `insure.jar` and `insureimpl.jar` from `[Jtest_install_dir]/plugins/com.parasoft.xtest.jtest_[version]/resources/` to a directory on the server with the application you wish to instrument.
2. If the server is running, stop it.
3. In your startup script, add the `-javaagent command` to the existing Java arguments. For details, see javaagent Command Details.
4. Restart the server.

The server will start as usual, but with the specified package classes instrumented. Now, whenever instances of objects are created or methods within the specified package prefixes are invoked, SOAtest (which can be running from another developer/QA desktop machine) will be able to receive event notifications in Event Monitor.

javaagent Command Details

Basics

The following invocation-time parameters are required in all situations:

Specifies which port should be used to communicate with the monitored program. Use 5050 to 5099.

Parameter	Description
soatest	Required for configuring monitoring.
port=[port_number]	Specifies which port should be used to communicate with the monitored program. Use 5050 to 5099.
instrument=[class_name.method_name (jni _sig)]	Specifies the prefixes of fully-qualified methods to check. For instance, given the `com.abc.util.IOUtil.method`, it will instrument all methods in `IOUtil.java` that start with `method`. If given `com.abc.`, it will also instrument those methods and all methods of classes whose fully qualified name starts with `com.abc.` You can provide specific class names, or use wildcards to monitor any class in the specified package. See the note below for more details.

trace=[class_name.method_name (jni_sig)]	Specifies the filter for method calls to trace. See the note below for more details.

Note - instrument and trace

Instrumentation of a class applies to all of its method bodies; it provides visibility, for example, into what methods that class calls and what values those methods return.

Tracing is implemented by also instrumenting the caller of the code you want to visibility into. The called code is not instrumented.

For example, assume you want to trace third-party methods called from your code, but not third-party methods called from other third-party code. In this case, you would instrument your own code, and trace the calls to the third-party code.

More, specifically....

`instrument=com.acme.util` configures instrumentation for all classes matching `com.acme.util`. All methods of those classes are instrumented. In the following code, the `writeData` method will be instrumented:

```
package com.acme.util;
class IOUtil {
       int writeData(DataOutputStream dos, Data data) {
           dos.write(data._int);
           dos.write(data._float);
       }
}
```

`instrument=com.acme.util,trace=`java.io provides visibility into the `java.io` calls made by `com.acme.util code`. Instrumentation adds calls to the `writeData()` method in order to check which calls to `java.io` are made by that monitored code.

The following parameters are optional:

Parameter	Description
trace_to_xml	Tells the monitor to serialize complex Java object into an XML representation. If this option is omitted, only primitive and "toString()" values of Objects will be returned. This parameter is strongly recommended for use with Event Monitor. It is not applicable if you are performing only runtime error detection.
xmlsizelimit=[integer_value]	Determines the maximum XML size limit in bytes. The default size is 100000 if this option is not specified. *Applies only when trace_to_xml is used.*
xmlcolsizelimit=[integer_value]	When generating an XML representation of Java objects, determines the maximum number of elements shown for collections/maps/arrays. The first 100 elements are shown by default. *Applies only when trace_to_xml is used.*
xmldeeplimit=[integer_value]	When generating an XML representation of Java objects, determines the maximum field depth included for data structures. Fields up to a depth of 8 are included by default. *Applies only when trace_to_xml is used.*
xmlexcl=[classes_or_fields]	':' separated classes or fields to exclude from xml serialization (i.e. xmlexcl=com.acme.A:com.acme.B._field). *Applies only when trace_to_xml is used.*
xmlinc=[classes_or_fields]	':' separated classes or fields to always include in xml serialization (i.e.xmlinc=com.acme.A:com.acme.B._field) Matches for xmlinc take preference over matches for xmlexcl: if something matches xmlinc, it will always be shown— even if xmlexcl also matches it. When the pattern matches a class name, 1) fields of that class type or a derived type are excluded from serialization and 2) method arguments and return values of that type or derived ones will not be serialized to xml. By default, the monitor excludes classes of the following types: ..... *Applies only when trace_to_xml is used.*

xmlsecondslimit=[seconds]	By default, if you are using trace_to_xml, the monitoring will spend only up to 10 seconds to convert a monitored complex Java object to an XML representation. This is to prevent significant slow-downs in the monitoring agent when monitoring very large objects. If that limit is reached, then the SOAtest event will show the following message instead of the XML representation of an object: SKIPPED: converting to XML takes too long: 10 seconds If you wish to change that 10 second threshold, use the xmlsecondslimit flag. example: xmlsecondslimit=20 For large objects, the recommended approach is to avoid reaching the threshold in the first place: reduce the XML size by excluding the fields you are not interested (using the xmlecl option).
trace_exceptions[=exception_c lass_prefix]	Shows a trace of events related to an exception that was created, thrown, or caught. `_details` can be added to get more detail of the events (i.e. the stack trace where the event happens).
terse	Configures terse output to the console (stack traces have only 1 element).

For Applications Running from Eclipse or Application Servers

Applications that define their own class loaders (i.e. Eclipse, JBoss and Tomcat) need `insure.jar` added to the boot classpath. To monitor those applications, add to the launch VM arguments:

```
-javaagent:"<path_to_jar>\insure.jar=soatest,port=<port>",trace_to_xml,instrument=<my.pack-age.prefix>,
trace=<my.package.prefix> -Xbootclasspath/a<path_to_jar>\insure.jar
```

For instance, you may use:

```
-javaagent:"/home/user/parasoft/insure.jar=soatest,port=5060",trace_to_xml,instru-
ment=com.mycompany.onlinestore,trace=com.mycompany.onlinestore -Xbootclasspath/a:/home/user/parasoft/insure.jar
```

For other (Standalone) Java Applications

For other (standalone) Java applications, you do NOT need to add `insure.jar` to the boot classpath. For instance, you may use:

```
java -javaagent:"C:\Program Files\Parasoft\insure.jar=soatest,port=5050",trace_to_xml,instru-ment=com.mycompany.
myapp,trace=com.mycompany.myapp
```

SOAtest Configuration

1. Double-click the **Event Monitor** tool to open up the tool configuration panel.
2. In the **Event Source** tab, select **Instrumented Java Application** as the platform, then specify the hostname where the server resides and the port number for the Parasoft agent runtime (5050 by default).

Tips

- When your application is first launched with the monitoring agent, we recommend that you per-form an initial run of the desired tests or scenarios before you start inspecting the reported events. This ensures that the various instrumentation and initialization processes have occurred. It also helps you obtain more realistic data. The first run's execution time may be significantly higher than subsequent runs, possibly resulting in timeouts or other side effects. Subsequent runs will better reflect the real behavior of the monitored system.
- Events are received asynchronously from the remote agent; there is no direct correlation between the events received and the test actions that trigger those events. Since SOAtest Event Monitor displays all the events it receives in chronological order, test events and Java monitoring events may sometimes be displayed out of order. To reduce the risk of this, you can increase the value for the **Event polling delay after each test finishes** execution setting. However, that will cause test execution time to increase (because the Event monitor will make test execution to pause after each invocation). Be sure that the **Maximum monitor execution duration** value is sufficient for the total test suite execution time. Different systems and environments may have different ideal thresholds for these parameters, so you will often need to tune them on a case-by-case basis.
- One best practice is to start instrumenting and tracing a broader set of packages/classes, then gradually reduce that set. More instrumentation results in slower application execution and more packages/classes being traced results in too many events being returned. You may adjust those two parameters and tune them to get the exact kinds of events you are interested in.
- When using the trace_to_xml option, you may get very large XML content representing your objects in the Event Monitor—possibly in megabytes and sometimes reaching the 100,000 bytes size limit (adjustable with `xmlsizelimit`). In these cases, consider using the various xml* parameters above to control the content you are interested in and limit the amount of data.