|  |  |
| --- | --- |
|  | Interoperable Trust Assurance Infrastructure |
|  | |
|  | |
| Specification of Testing&Monitoring-related Aspects | |

|  |  |
| --- | --- |
| ***Type (distribution level)*** | Restricted *(according to DoW)* |
| ***Contractual date of Delivery*** | 30-09-2013 |
| ***Actual date of delivery*** | 07-10-2013 |
| ***Deliverable number*** |  |
| ***Deliverable name*** |  |
| ***Version*** | V1.0 |
| ***Number of pages*** | 14 |
| ***WP/Task related to the deliverable*** | WP4/ T.4.5 |
| ***WP/Task responsible*** | SL |
| ***Author(s)*** | Vesztergombi, György (SL) |
| ***Partner(s) Contributing*** | SL |
| ***Document ID***  ****** |  |

|  |  |
| --- | --- |
| ***Abstract*** | *This document describes the interface and usage of the Notification and Injection Aspects. The Notification Aspect is a tool that generates a stream of log events. The purpose of the stream is to provide data about the internal state of the target application. The output stream can be analyzed by a diagnostic application that is responsible for monitoring the health of the target application (Montimage Monitoring Tool).*  *The Injection Aspect is a testing tool. It can reset the state of selected objects on demand.* |

Table of contents

Table of contents 3

1 Introduction 4

2 Scope of Measurement Points 5

3 Status information 6

4 Local Monitoring of System Internals 7

4.1 Method Monitor 7

4.2 Attribute Monitor 8

4.3 Class Monitor 8

4.4 Count Monitor 9

4.5 Ping Monitor 9

5 Recursive and Global Monitoring 11

5.1 Recursive Monitoring 11

5.2 Global Monitoring 12

5.3 Exclude – Monitoring Optimization 12

6 Setting System State 13

6.1 Injection – Runtime Method Calls 13

7 Summary 14

# Introduction

In this document we present the specification of the Notification Aspect and the Injection Aspect. The Notification Aspect continuously checks and monitors application health. It notifies the Monitoring module about measurements of execution internals and other internal conditions of the Inter-Trust-enabled programs. The Injection Aspect enables resetting application modules to pre-defined states, to facilitate repeated test trials to be carried out in a controlled manner.

The package targets the Java SE platform, and uses AspectJ technology, a statically weaved AOP approach. To use the Notification Aspect you must weave it into the target application and annotate the code based on this specification (standard Java annotation style). During operation of the target application the Notification Aspect produces a stream of log messages. You can specify measurement points that generate data which is sent to the logger. Measurement points can be attached to classes, methods and attributes. You can also select multiple code parts for monitoring with different scope.

The Injection Aspect provides an API to invoke the reset method of selected classes. Annotations identify classes along with their reset method. The Injector singleton stores annotated objects upon creation using weak references. The test environment can invoke the singleton’s resetAll() method, to call the stored methods. Injection Aspect relies on introspection – implemented by Java’s reflection API, to acquire and invoke reset methods. Therefore this functionality should not be disabled in Java’s security manager.

# Quick Start Guide

To start using the Notification Aspect, you have to initialize it in the start-up section of your code. The following snippet shows how to do this.

import sl.NotificationAspect;

**[4]** import static sl.Const.\*;

void main() {

...

**[1]** NotificationAspect.connectMonitoringTool("http://localhost:4567/");

**[2]** // NotificationAspect.enable\_log\_to\_stdout(true); --- Debugging feature

**[3]** NotificationAspect.log\_only(all ^ source\_line);

1. Connect to the MMT Monitoring Tool by supplying its URL;
2. Enable logging to stdout for debugging, if you need to. The Notification Aspect produces logs even in the absence of a connection to the MMT tool.
3. Restrict logging on the global level. In this example logging of source line information is disabled.
4. Constants used by log\_only() are defined in sl.Const. In production code, you probably do not want to use static import, but a regular import sl.Const and proper namespaces.

Log\_only() accepts filter masks that can be created using bit operations. The constant all, contains all valid options. Use XOR (^) to subtract, or alternatively use OR (|) to combine options.

Annotate your code to start producing monitoring data. The next example shows a class with annotations.

import sl.\*;

import static sl.Const.\*;

public class A1 {

@Monitor

int variableA;

@MonitorOnly(mask=all ^ source\_line ^ attribute\_value)

int variableB;

@Ping

public int ping() { return 12; }

**[1]** @MonitorOnly(mask=Const.all ^ Const.method\_params)

public int params(String str, int val) { return 12; }

@Monitor

public void outer() throws Exception { inner(); }

@MonitorOnly(mask=all ^ method\_params ^ source\_line)

public void outer() throws Exception { inner(); }

**[2]** @MonitorOnly(mask = log\_msg | name)

public int params(String str, int val) { return 12; }

1. The @MonitorOnly annotation is a variant of the @Monitor annotation, where you can override the output logging format. The mask bit-field controls which attributes are sent to the log.
2. The mask restricts the log entry to two fields: the log log message and the join point name.

The remaining sections present the features, mentioned briefly in this introduction, in more detail.

# Remote Logging to MMTConnector

Notification and Injection aspects use MMTConnector for remote logging. Use the following static function to connect.

import sl.NotificationAspect;

void main() {

...

NotificationAspect.connectMonitoringTool("http://localhost:4567/");

A sample output log message, sent to standard output:

Method Enter -- name:test.A.c, source:A.java:15

A sample output log message, sent to MMT Connector as name value pair:

msg:”Method Enter”, name:”test.A.c”, source:”A.java:15”

# Scope of Measurement Points

Measurement points, or meters measure different aspects of the code during runtime, like execution duration, thread count or function name.

Monitoring meters work on two different scope levels

* Local, and
* Recursive.

You can define local meters using Java annotations. Annotations are a part of the Java language, and provide a mechanism for annotating code with metadata. Annotations do not modify program execution by themselves, but they can be read during compile time or execution time.

Meters operating at the local scope level are always marked by an annotation. The annotation may be bound to a class, method or attribute declaration. Only annotated elements are impacted by local scope meters.

In the next scope level – recursive monitoring, annotations are used as well. Beside the annotated code, all code reachable through control flow is monitored, up to the available call depth. Call depth is limited by the available source code, because static aspects operate by modifying accessible source code. Our instrumentation therefore does not penetrate pre-compiled classes, like .class files or system libraries.

# Status information

The Notification Aspect appends status information to each logged event. The status information string contains the following data:

* Monitored object name, which can be a class name, method name or attribute name;
* Source file name and source line number;
* Thread id of current thread;
* Total number of threads;
* Total number of tracked objects;
* Time stamp.

Sample output:

name:B.x, source:Main.java:16, thread\_name:main, thread\_count:1,

time\_stamp[ms]:14994343

Output is made up of name-value pairs.

# Local Monitoring of System Internals

You can use three different annotations to monitor code at the local level:

* @Monitor
* @Count, and
* @Ping.

In addition, the @Monitor annotation can be used in three different contexts. It can be placed before

* classes,
* methods, and
* attributes.

## Method Monitor

Using this meter you can measure the execution time of a method and monitor its exit status.

|  |  |
| --- | --- |
| **Property** | **Description** |
| Name | Method monitor |
| Annotation | @Monitor |
| Scope | Local |
| Location | Before method declaration |
| Output | * Method start event * Method end event * Method duration * Exit status   + success – on normal return   + exception\_thrown – if an exception was raised   + exception text – if an exception was raised * Status information |
| Usage sample | @Monitor  **void** monitor\_nested() {  nested();  } |
| Output sample | Method Enter -- name:test.A.c, source:A.java:15, …  Method Leave -- exit\_status:success,  duration[µs]:3580.788, … |

In the usage sample, we annotated monitor\_nested() with the @Monitor annotation. First, method monitor logs the function start event. It produces a second log event on function return, which includes the run-time duration and exit status. Because of *local scope*, method monitor does not track the call to nested() in the method body.

## Attribute Monitor

Use the attribute monitor to capture changes made to attributes, and to log the new value.

|  |  |
| --- | --- |
| **Property** | **Description** |
| Name | Attribute monitor |
| Annotation | @Monitor |
| Scope | Local |
| Location | Before attribute declaration |
| Output | Attribute value, as returned by toString(), on each assignment operation |
| Usage sample | public class B {  @Monitor  public int x = -1;  @Monitor  private int y; |
| Output sample | Attribute changed -- value:-1, name:B.x, …  Attribute changed -- value:3, name:B.x, … |

In the usage sample, we have annotated variable x and y with the @Monitor annotation. Attribute monitor generates a log entry for the variable x, on object initialization. Later in the code, we assign y the value 3 (not shown in the usage sample), which is logged by attribute monitor. A log about y does not appear in the log either, since no value is assigned to it.

## Class Monitor

Class monitor combines Method monitor and Attribute monitor, and applies them to all methods and attributes of the class.

|  |  |
| --- | --- |
| **Property** | **Description** |
| Name | Class monitor |
| Annotation | @Monitor |
| Scope | Local |
| Location | Before class declaration |
| Output | Combines output of Method monitor and Attribute monitor, for all methods and attributes declared in class |
| Usage sample | @Monitor  public class A {  int a;  public void func() {  ... |
| Output sample | Method Enter -- name:test.A.c, source:A.java:15, …  Attribute changed -- value:3, name:A.a, …  Method Leave -- exit\_status:success,  duration[µs]:3580.788, … |

In the usage sample, we have annotated class A with the @Monitor annotation. Class monitor produces log entries when the value of variable a is set, and tracks the execution of method func().

## Count Monitor

Use the Count monitor to track the number of objects on the memory heap. Count monitor increments a counter when a new object is created, and decrements the counter an object is finalized. Finalization occurs right before the garbage collector reclaims the memory allocated to the object. Each annotated class receives an individual counter.

|  |  |
| --- | --- |
| **Property** | **Description** |
| Name | Object Counter |
| Annotation | @Count |
| Scope | Local |
| Location | Before class declaration |
| Output | The global object counter is increased and decreased by this meter. The object count total is part of the common status string, and is appended to every log output. |
| Usage sample | @Count  public class A {...} |
| Output sample | Object count is part of the status information that is appended to each log event.  count(test.B):1, count(test.A):1, tracked\_objects:2 |

We have added the @Count annotation to class A. On each call of the new operator the object counter is incremented. Eventually, when an object is no longer referenced, the garbage collector de-allocates it.

## Ping Monitor

The ping monitor generates a signal whenever a function is called. You can regard it as a lower impact alternative to the method monitor, because duration, exit status and returning from the function are not tracked. You can use it for example, to monitor application heartbeat by hooking it to a frequently called method.

|  |  |
| --- | --- |
| **Property** | **Description** |
| Name | Ping monitor |
| Annotation | @Ping |
| Scope | Local |
| Location | Before method declaration |
| Output | On each invocation of the method a ‘Ping’ message is added to the log. |
| Usage sample | @Ping  **void** receive\_packet() {  ...  } |
| Output sample | Method Enter -- name:test.A.c, source:A.java:15, … |

We have annotated the receive\_packet() function with the @Ping annotation. Whenever a new packet arrives, ping monitor produces a log event.

# Recursive Monitoring

Local monitors allow us to focus on a single class, method or attribute. Recursive and global monitors work on a larger scope and provide a higher coverage of the source code base. Recursive monitors factor in the execution control flow, while global monitors target methods and attributes within the entire source code base.

## Recursive Monitoring

You can use recursive monitoring, to implement taint analysis, in order to track the control flow of methods that process untrusted input. Taint monitor produces the same output as method monitor, but tracks all methods that are reachable by the control flow.

|  |  |
| --- | --- |
| **Property** | **Description** |
| Name | Taint Monitor |
| Annotation | @Taint |
| Scope | Recursive |
| Location | Before method declaration |
| Output | Outputs the same information as method monitor, for each method encountered by the control flow. Notably:   * duration, and * exit status. |
| Usage sample | @Taint  **void** taint\_test() {  nested();  }  **void** nested() {  inner();  }  **void** inner() {} |
| Output sample | Method Enter -- name:A.taint\_test, source:A.java:13, …  Method Enter -- name:A.nested, source:A.java:20, …  Method Enter -- name:A.inner, source:A.java:26, …  Method Leave -- name:A.inner, source:A.java:26, …  Method Leave -- name:A.nested, source:A.java:20, …  Method Leave -- name:A.taint\_test, source:A.java:13,… |

We have annotated taint\_test() with the @Taint annotation. Taint monitor tracks taint\_test(), nested() and inner, because they are reachable by control flow from taint\_test. Note that they are not annotated with @Taint.

## Exclude – Monitoring Optimization

The exclusion feature can be used to restrict the impact of global and recursive meters.

To improve the performance of the target application, after heavy instrumenting, the exclude feature can be used to remove code parts from the monitoring scope. Methods, Attributes and Classes annotated with @Exclude are excluded from monitoring. The exclusion property is recursively transferred to the reachable code set.

|  |  |
| --- | --- |
| **Property** | **Description** |
| Name | Exclude property |
| Annotation | @Exclude |
| Scope | Recursive |
| Location | Before class or method |
| Output | ‘Negative output’, curtails output of impacted code |
| Usage sample | @Monitor  public class A {  int a;  @Exclude  public void func() {  ... |
| Output sample | – |

We have added the @Monitor annotation to class A. This would generate log entries for attribute a and method func(). To omit func() from log generation, we have added the @Exclude annotation. Now, class monitor will only track attribute a, and no logs will be generated for func().

# Restricted Logging

Applications that are sensitive to disclosing information to the remote logger can configure their logging output. Applications can set a mask that controls the logging output. Valid options are specified in sl.Const.

public static final long

*attribute\_value* = 1<<0,

*class\_id* = 1<<1,

*class\_instances* = 1<<2,

*duration* = 1<<3,

*exception* = 1<<4,

*exit\_status* = 1<<5,

*log\_msg* = 1<<6,

*name* = 1<<7,

*method\_params* = 1<<8,

*return\_value* = 1<<9,

*source\_line* = 1<<10,

*thread\_count* = 1<<11,

*thread\_name* = 1<<12,

*time\_stamp* = 1<<13,

*tracked\_objects* = 1<<14,

*all* = (1<<15)-1;

Each field controls one attribute from a log entry. Set the field to 1 enable 0 to disable the entry. The entries are explained in more detail by the following table.

|  |  |
| --- | --- |
| **field** | **Description** |
| Name | Inject monitor |
| Annotation | @Inject |
| Scope | Local |
| Location | Before class and method declaration |
| Effect | The Injector class stores a weak reference on object creation. On a resetAll() command all reset methods are called. |
| class\_id | Id of classes monitored by @Count |
| class\_instances | Number of class instances monitored by @Count |
| duration | Method duration monitored by @Monitor |
| exception | String representation of exception, thrown during execution of a monitored function. |
| exit\_status | Exit status of method |
| log\_msg | Description of log message |
| name | Name of join point, such as function, variable, etc. … |
| method\_params | Method parameters, disable for sensitive methods |
| return\_value | Method return value |
| source\_line | Source line of join point |
| thread\_count | Number of threads in current process |
| thread\_name | Name of current thread |
| time\_stamp | Time stamp |
| tracked\_objects | Number of tracked objects |
| all | Union of all of the above options. Use it to subtract options with XOR |
|  |  |

To control logging globally use the following static function.

NotificationAspect.log\_only(all ^ source\_line);

For local control use the annotation variants suffixed with Only. Add the mask using a named parameter, as shown below.

@MonitorOnly(mask=Const.log\_msg | Const.method\_params)

public int params(String str, int val) { return 12; }

Restricted variants are listed by the table below.

|  |  |
| --- | --- |
| **Annotation** | **Description** |
| @PingOnly | Restricted variant of @Ping |
| @MonitorOnly | Restricted variant of @Monitor; Use for classes, methods and attributes |
| @TaintOnly | Restricted variant of @Taint; Use for classes and methods |

You can use local annotations to override global settings.

# Setting System State

In this chapter we present the features of the Injection Aspect, which is designed to support testing operations.

## Injection – Runtime Method Calls

Injection monitor helps us create test fixtures. To test with repeated trials, we need the capability to reset the target application to a known starting state. We assume that each relevant object implements the required reset functionality in a dedicated method (but we do not assume anything regarding the name of this method).

|  |  |
| --- | --- |
| **Property** | **Description** |
| Name | Inject monitor |
| Annotation | @Inject |
| Scope | Local |
| Location | Before class and method declaration |
| Effect | The Injector class stores a weak reference on object creation. On a resetAll() command all reset methods are called. |
| Usage sample | @Inject(reset=”reset\_me\_please”)  public class B {  **public** **void** reset\_me\_please() {  ...  }  }  // Invocation by the test logic:  **public** **static** **void** main(String[] arg) {  ...  Injector.*singleton*().resetAll();  ...  } |
| Output sample | – |

Class B must be reset before each test run. The @Inject annotation marks class B for the Injection aspect, and indicates the reset\_me\_please() method as the method implementing this reset functionality. Test logic invokes resetAll() on the Injector singleton, which forwards the call to reset\_me\_please() and all other acquired methods.

# Summary

In this section we summarize the annotations of the Notification and Injection aspect in table format.

|  |  |  |  |
| --- | --- | --- | --- |
| **Monitor name** | **@Annotation** | **Scope** | **Target** |
| Attribute monitor | Monitor | local | attribute |
| Method monitor | Monitor | Local | method |
| Class monitor | Monitor | local | class |
| Taint monitor | Taint | recursive | method |
| Count monitor | Count | local | class |
| Ping monitor | Ping | local | method |
| Injection | Inject(reset=”method name”) | local | class, method |
| Exclude | Exclude | local | class, method, attribute |

The table lists each monitor, along with its annotation mark, application scope, and targeted source code element. Global variants exist for several monitors that can be enabled by weaving in their corresponding aspect.