JBossTS Test Orchestration Package

Programmer's Guide

Version 0.1

26 September 2008

Table of Contents

Introduction

The orchestration package is a tool designed primarily to support automation of tests for multi-threaded and multi-JVM Java applications. The package provides two means of modifying application program behaviour. Firstly, it allows side-effects to be introduced at specified points during execution. A suite of operations are supported which can be used to coordinate the activities of independent threads e.g. by introducing delays, waits and signals, countdowns, flag operations and so on. Trace statements are also supported so that test deployment scripts can track progress of a test run and identify successful or unsuccessful test completion. Further application-specific side-effects can also be implemented by introducing invocations of publicly accessible application or JVM methods at arbitrary points in the application code.

The second option supported by the tool is to subvert the behaviour of application code by modifying execution paths. The package allows early return from methods with the option of supplying a different return value to the one the application would normally compute. It also enables runtime exceptions to be thrown, effectively either aborting a thread or, at least, where a catch-all handler is employed, the portion of that thread's call tree bounded by the handler and the throw point. Finally, the package allows a machine crash to be simulated by configuring an immediate exit from the JVM when it is detected that a suitable point during the application life-cycle has been reached.

Event Condition Action Rules

The orchestration tool explicitly avoids the need for direct modification of application code. It employs an Event Condition Action (ECA) rule language. The three components of these rules, event, condition and action, are used, respectively, to define:

- *where* during application execution a side-effect should occur
- *whether* the side-effect should happen or not
- *what* the side effect should be

Each test is expected to define a set of rules specifying the side-effects it wishes to introduce into the application under test. Rules may be supplied in a single script or as a set of scripts, the latter allowing reuse of rules across related tests.

Agent Transformation

At JVM bootstrap the set of rules is read by a *Java agent* program. JVM class loaders provide agents with an opportunity to modify loaded bytecode just prior to compilation (see package `java.lang.Instrumentation` for details of how Java agents work). The orchestration package agent is responsible for monitoring method code as it is loaded and identifying *trigger points* where the rule engine code should be invoked. This allows the rule engine to introduce the desired side effects.

The agent inserts *trigger calls* into code at each point which matches a rule event (the *where* part of the rule). Trigger calls are calls to the rule execution engine which identify the *trigger method*, i.e. the method which contains the trigger point, and the rule which has been matched. If several rules match the same trigger point then there will be a sequence of trigger calls, one for each matching rule. When a trigger call occurs the rule execution engine locates the relevant rule, establishes bindings for variables mentioned in the rule event and then *tests* the rule condition. If the condition evaluates to true it *fires* the rule.

executing each of the rule actions in sequence.

Trigger calls pass the method recipient and arguments to the rule engine and these are made available to the condition and action via rule bindings. This allows data from the triggering context to be tested in the condition in order to decide whether to fire the rule and to be employed as a target or parameter for rule actions. The event specification includes a binding specification which can introduce bindings for additional variables by invoking methods or operations on the method bindings and/or static data. Note that the agent will eventually be updated to support binding of values for local variables which are in scope at the trigger point.

The agent also compiles exception handler code into the trigger method in order to deal with exceptions which might arise during rule processing. This is not intended to catch errors which arise during rule execution since these are dealt with internal to the rule execution engine. Exceptions are thrown by the execution engine to alter the flow of control through the triggering method. Normally, after returning from a trigger call the triggering thread continues to execute the original method code. However, it is possible for rule actions to specify that an early return or exception throw should be performed *from the trigger method*. The rule language implementation achieves this by throwing its own private, internal exceptions below the trigger call. The handler code compiled into the trigger method catches these internal exceptions and then either returns to the caller or recursively throws a runtime exception, thus avoiding normal execution of the remaining code in the body of the triggering method.

ECA Rule Engine

The rule execution engine consists of a rule parser, type checker and interpreter/compiler. The rule parser is invoked by the agent during bootstrap. This provides enough information to enable the agent to identify potential trigger points. Rules cannot be type checked and compiled until the class and method bytecode they refer to has been loaded. This is because type checking requires identifying properties of the trigger class and, potentially, of classes it mentions. The type checker needs to identify properties of loaded classes such as the types and accessibility of fields, method signatures etc. So, in order to ensure that the trigger class and all its dependent classes have been loaded before the type checker tries to access them, rules are type checked and compiled the first time they are triggered. This also avoids the cost of checking and compiling rules included in the rule set which do not actually get called.

A single rule may be associated with more than one trigger point. Firstly, depending upon how precisely the rule specifies its event, it may apply to more than one class or more than one method within a class. But secondly, even if a rule specifies a class and method unambiguously the same bytecode file may be loaded by different class loaders. So, the rule has to be type checked and compiled for each applicable trigger point.

If a type check or compile operation fails the rule engine prints an error and disables execution of the trigger call. Note that in cases where the event specification is ambiguous a rule may type check successfully against one trigger point but not against another. Rule execution is only disabled for cases where the type check fails.

Currently, trigger calls execute a rule by interpreting the rule parse tree. The rule engine creates an instance of a helper class (the inner class `Rule.Helper`) to provide a context for each trigger call. This helper instance stores the bindings for the method parameters and for variables introduced by the rule event. This ensures that concurrent triggers of the same rule from different threads do not interfere with each other. The default helper class implements methods which perform binding of the rule's event variables (method `bind`),

testing of the rule's condition (method `test`) and execution of the rule's actions (method `fire`). These implementations all work by interpreting the parse tree. In future this will be optimized by having the rule compiler generate a specialization of the helper class for each trigger point. This generated class will include runtime-generated code for the bind, test and fire methods derived from the type checked parse tree.

Rule Language

Rules are defined in scripts which consists of a sequence of rule definitions interleaved with comment lines. Comments may occur within the body of a rule definition as well as preceding or following a definition but must be on separate lines from the rule text. Comments are lines which begin with a # character:

```
########################################
# Example Rule Set
#
# a single rule definition
RULE example rule
# comment line in rule body
     . . .
ENDRULE
```

Rule Events

Currently, rule event specifications are limited either to invocations of a *target* method associated with a *target* class or execution of a particular source code line in a target method. Target methods can be either static or instance methods or constructors. In future, the rule language may be extended to include other events such as, for example, accesses to instance or static fields. So, at present, rule specifications have the following structure

```
# rule skeleton
RULE <rule name>
CLASS <class name>
METHOD <method name>
LINE <line number>
BIND <bindings>
IF   <condition>
DO   <actions>
ENDRULE
```

The name of the rule following the `RULE` keyword can be any free form text with the restriction that it must include at least one non-white space character. Rule names do not have to be unique but it obviously helps when debugging rule scripts if they clearly identify the rule. The rule name is printed whenever an error is encountered during parsing, type checking, compilation or execution.

The class and method names following the CLASS and METHOD keywords must be on the same line. The class name can identify a class either with or without the package qualification. The method name can identify a method with or without an argument list or return type. A constructor method is identified using the special name `<init>`. For example,

```
# class and method example
RULE any commit on any coordinator engine
CLASS CoordinatorEngine
METHOD commit
. . .
ENDRULE
```

matches the rule with any class whose name is `CoordinatorEngine`, irrespective of the package it belongs to. When any class with this name is loaded then the agent will insert a trigger point at the beginning of any method named `commit`. If there are several occurrences of this method, with different signatures then each method will have a trigger point inserted.

More precise matches can be guaranteed by adding more detail. For example,

```
# class and method example 2
RULE commit with no arguments on wst11 coordinator engine
CLASS com.arjuna.wst11.messaging.engines.CoordinatorEngine
METHOD State commit()
LINE 324
. . .
ENDRULE
```

This rule will only match the `CoordinatorEngine` class in package `com.arjuna.wst11.messaging.engines` and only match a method `commit` with no arguments and with a return type whose name is `State`. Note that the package for class `State` is left unspecified. The type checker infers this information from the matched method.

The previous example also employs the optional `LINE` number specification. The text following the line keyword must be able to be parsed to derive an integer line number. This directs the agent to insert the trigger call at the start of a particular line in the source code. If there is no executable code at the specified line or the line is not within the body of the selected method then the agent will not insert a trigger point (note that it does not currently print an error in such cases because this may merely indicate that the rule does not apply to this particular class or method – *perhaps this behaviour needs revising*?).

Note:

Currently the agent code will only transform code for classes which belong to the `com.arjuna` and `org.jboss` packages. It will not transform any classes in package `org.jboss.jbossts.orchestration` (the orchestration package itself).

Rule Bindings

The event specification also contains a binding specification which computes values for variables which can subsequently be referenced in the rule body. These values will be computed each time the rule is triggered before testing the rule condition. For example,

```
# binding example
RULE countdown at commit
CLASS com.arjuna.wst11.messaging.engines.CoordinatorEngine
METHOD commit
LINE 316
BIND engine:CoordinatorEngine = $0,
     recovered:boolean = engine.isRecovered(),
     identifier:String = engine.getId()
. . .
ENDRULE
```

creates a variable called `engine`. This variable is bound to the recipient of the `commit` method call which triggered the rule, identified by the parameter reference `$0` (if `commit` was a static method then reference to `$0` would result in a type check exception). Arguments to the trigger method can be identified using parameter references with successive indices, `$1`, `$2` etc. The declaration of engine specifies its type as being `CoordinatorEngine` though this is not strictly necessary since it can be inferred form the type of `$0`.

Similarly, variables `recovered` and `identifier` are bound by evaluating the expressions on the right of the = operator. Note that the binding for engine has been established before these variables are bound so it can be referenced in the evaluated expression. Once again, type specifications are provided but they could be inferred.

Rule Expressions

Expressions which occur on the right hand side of the = operator in event bindings can be simple expressions i.e.

- references to bound variables
- static data references
- primitive literals
- field accesses
- method invocations
- built-in operation invocations

n.b. built-in operations are explained in more detail below.

Expressions can also be complex expressions composed from other expressions using the usual Java operators: `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `&&`, `||`, `!`, `==`, `!=`, <. <=, >, >=, etc. The ternary conditional expression operator, `? :`, can also be employed. The type checker does its best to identify the types of simple and complex expressions wherever possible. So, for example, if it knows the type of bound variable `engine` then it will be able to employ reflection to infer the type of a field access `engine.recovered` or a method invocation `engine.isRecovered()`.

Note:

- The assignment operator is not available for use in expressions. It can only be employed at the top level in binding specifications.
- Use of the new operator is not currently allowed in expressions.
- The throw operation is not currently allowed in rule expressions. Specific exceptions

may be thrown by calling builtin operations.

- Also, the current parser is a little ropey (antlr is a bit of a crock to work with – bison, where art thou?) so you may need to use brackets to ensure the correct precedence.
- It should eventually be possible to allow expressions to make references to local variables which are in scope at the trigger point as well as to method arguments. So, for example, if line 316 of method commit was preceded by a declaration for an `int` variable with name `idx` then it should be possible to refer to this variable in expressions using a local variable reference of the form `$idx`. This reference would evaluate to the value of the local variable when the trigger point was reached.

Rule Conditions

Rule conditions are nothing more than rule expressions with boolean type. For example,

```
# condition example
RULE countdown at commit
CLASS com.arjuna.wst11.messaging.engines.CoordinatorEngine
METHOD commit
LINE 316
BIND engine:CoordinatorEngine = $0,
     recovered:boolean = engine.isRecovered(),
     identifier:String = engine.getId()
IF   recovered
. . .
ENDRULE
```

merely tests the value of bound variable `recovered`. The same effect could be achieved by using the following condition

```
# condition example 2
RULE countdown at commit
CLASS com.arjuna.wst11.messaging.engines.CoordinatorEngine
METHOD commit
LINE 316
BIND engine:CoordinatorEngine = $0,
     . . .
IF   engine.isRecovered()
. . .
ENDRULE
```

Alternatively, if, say, the instance employed a public field, `recovered`, to store the boolean value returned by method `isRecovered` then the same effect would be achieved by the following condition.

```
# condition example 3
RULE countdown at commit
CLASS com.arjuna.wst11.messaging.engines.CoordinatorEngine
METHOD commit
LINE 316
BIND engine:CoordinatorEngine = $0,
     . . .
IF   engine.recovered
. . .
ENDRULE
```

Note that the boolean literal `true` is available for use in expressions so a rule which should always fire can use this as the condition expression.

Rule Actions

Rule actions are either a rule expression or return expression or a comma-separated sequence of rule expressions, possibly ending with a return expression. Rule expressions occurring in an action list may have arbitrary type, including void type.

A return expressions is the `return` keyword possibly followed by a rule expression which is used to compute a return value. A return expression causes a return from the triggering method so it must supply a return value if and only if the method is non void. If a return value is employed then the type checker will ensure that it's type is assignable to the return type of the trigger method.

An empty action list may be specified using the keyword `NOTHING`.

Rule Language Built-Ins

The rule language provides a suite of built-in calls for use in rule expressions. Invocations of built-ins are written as simple method calls without a recipient (i.e. rather like static data calls). These are primarily intended for use in condition and action expressions but they may be called anywhere. They provide features which are designed to make it easy to perform complex tests, in particular to coordinate the actions of threads in multi-threaded applications. These features are not fixed in stone and the rule engine has been designed to ensure that new built-ins can easily be added as new requirements are identified.

Helpers and Built-In Operations

The reason for employing a helper class to handle rule execution is that it locates in one place all the execution context for the triggered rule. This includes the current set of bindings and a back-link to the original rule and, from there, the triggering class and method. This also provides an opportunity for the helper class to supply the implementation of any built-in operations which may be desired in rule event bindings, conditions or actions.

Built-in calls are written without a recipient as though they were invocations of a method on self. The rule engine identifies calls in this format and translates them to runtime invocations of helper class instance methods. So, for example, the helper class implements a method with signature

```
boolean trace(String message)
```

This method prints the supplied string to `System.out` and always returns true. It can be used in a rule action to display a trace message, for example:

```
DO   debug("killing JVM"), killJVM()
```

When the `debug` built-in is executed the rule engine calls the corresponding method of the current helper instance passing it the string `"killing JVM"`. Method `killJVM` is another built-in which is implemented by an instance method of class `Rule.Helper`.

Note that method debug has a boolean signature so that tracing can also be performed in rule conditions. This would normally occur in combination with a test of some bound variable or method parameter, for example:

```
IF   participant.isRecovered()
     AND
     debug("recovered participant " + participant.getId())
```

The rule language implementation automatically exposes all public instance methods of class `Rule.Helper` as built-in operations. So when the rule type checker encounters an invocation of `debug` with no recipient supplied it identifies that debug is a method of class and automatically type checks the call against this method. At execution time the call is executed by invoking the relevant implementation on the helper instance handling the rule trigger call.

This feature allows additional built-ins to be added automatically to the rule engine by adding new instance methods. No changes are required to the parser, type checker and compiler in order for this to work.

Countdowns

The rule engine provides countdowns which can be used to ensure that rule actions triggered in one thread only occur after other rules have been triggered or fired a certain number of times. The API defined by the helper class is

```
public boolean addCountDown(Object identifier, int count)
public boolean getCountDown(Object identifier)
public boolean countDown(Object identifier)
```

Countdowns are identified by an arbitrary object, allowing successive calls to the countdown API to apply to the same or different cases. This identification can be made across different rule and helper instances. For example, a pair of rules might use the recipient of their respective trigger methods as the identifier. A countdown created by the first rule would only be decremented when the second rule is triggered from a method call with the same recipient. Countdowns created by invocations with alternative recipients would also match up accordingly. However, if the countdown were identified from both rules using a specific `String` literal then the countdown created by the first rule would be decremented by the second rule irrespective of whether the trigger method calls were on related instances.

`addCountDown` is used to create a countdown. `count` specifies how many times the countdown will be decremented before a decrement operation fails i.e. if `count` is `1` then the countdown will decrement once and then fail at the next decrement. If count is supplied with a value less than 1 it will be replaced with value 1. `addCountDown` would normally be employed in a rule action. However, it is defined to return true if a new countdown is created and false if there is already a countdown associated with the identifier. This allows it to be used in rule conditions where several rules may be racing to create a countdown.

`getCountDown` is for use in a rule condition to test whether a countdown associated with a given identifier is present, returning `true` if so otherwise `false`.

`countDown` is for use in a rule condition to decrement a countdown. It returns `false` if the decrement succeeds or if there is no countdown associated with identifier. It returns `true` if the countdown fails i.e. it has count `0`. In the latter case the association between the identifier and the countdown is removed, allowing a new countdown to be started using the same identifier. Note that this behaviour ensures that a race between multiple threads to decrement a counter from a rule condition can only have one winner.

Waiters

The rule engine provides waiters used to suspend threads during rule execution and then have other threads wake them up. The wakeup can simply allow the suspended thread to resume execution of the rule which suspended it. Alternatively, it can force the waiting thread to exit from the triggering method with an exception. The API defined by the helper class is

```
public void waitFor(Object identifier)
public void waitFor(Object identifier, long millisecsWait)
public boolean waiting(Object identifier)
public boolean signal(Object identifier)
public boolean signalKill(Object identifier)
```

As with countdowns, waiters are identified by an arbitrary object. Note that the wait operation is not performed by invoking `Object.wait` on `identifier`. Doing so might interfere with locking and synchronization operations performed by the triggering method or its callers. The identifier is merely used by the rule engine determine the target of wait and signal operations. It is used to index private instances which manage the synchronization activity.

`waitFor` is intended for use in a rule action. It suspends the current thread until either a `signal` or a `signalKill` is called with the same identifier. In the former case the thread will continue processing any subsequent actions or will return from the trigger call. In the latter case the thread will throw a runtime exception from the triggering method call frame. The version without a wait parameter will never time out. The version which employs a wait parameter will time out after the specified number of milliseconds.

`waiting` is intended for use in rule conditions. it will return true if any threads are waiting for a signal associated with `identifier`. It returns false if there are no threads waiting.

`signal` is intended for use in rule conditions or actions. If there are threads waiting for a signal associated with `identifier` it wakes them and returns `true`. If not it returns `false`. Note this behaviour ensures that a race between multiple threads to signal waiting threads from a rule condition can only have one winner.

`signalKill` is identical to `signal` except that it does not just wake any waiting threads. It also causes them to throw a runtime exception of type `ExecuteException` from their triggering method call frame when they wake up.

Flags

The rule engine provides a simple mechanism for setting, testing and clearing global flags. The API defined by the helper class is

```
public boolean flag(Object identifier)
public boolean flagged(Object identifier)
public boolean clear(Object identifier)
```

As with countdowns, flags are identified by an arbitrary object. All three methods are designed to be used either in conditions or actions.

`flag` can be called to ensure that the flag identified by `identifier` is set. It returns `true` if the flag was previously clear otherwise `false`. Note that the API is designed to ensure that race conditions between multiple threads trying to set a flag from rule conditions can only have one winner.

`flagged` tests whether the flag identified by `identifier` is set. It returns `true` if the flag is set otherwise `false`.

`clear` can be called to ensure that the flag identified by `identifier` is clear. It returns `true` if the flag was previously set otherwise `false`. Note that the API is designed to ensure that race conditions between multiple threads trying to clear a flag from rule conditions can only have one winner.

Note a flag is semantically equivalent to a countdown created with count 0.

Tracing

The rule engine currently provides a single tracing call. The API defined by the helper class is

```
public boolean debug(String message)
```

`debug` prints the supplied message to `System.out`, prefixed with the name of the rule being executed. It always returns true, allowing debug messages to be used in conditions by ANDing them with other boolean expressions.

Aborting Execution

The rule engine provides two built-ins for use in rule actions which allow execution of the triggering method to be aborted. The API defined by the helper class is

```
public void killThread()
public void killJVM()
public void killJVM(int exitCode)
```

`killThread` causes a runtime exception of type `ExecuteException` to be thrown from the triggering method call frame. This will effectively kill the thread unless a catch-all exception handler is installed somewhere up the call stack.

`killJVM` results in a call to java.lang.Runtime.getRuntime().halt(). This effectively kills the JVM without any opportunity for any registered exit handlers to run, simulating a JVM crash. If `exitCode` is not supplied it is defaulted to -1

Other Builtins

It is intended to implement a rendezvous API along the following lines:

```
public boolean rendezvous(Object identifier, int count)
public int rendezvousCount(Object identifier)
public int rendezvousLimit(Object identifier)
```

`rendezvous` will cause the current thread to suspend until `count` other threads have arrived at the rendezvous identified by `identifier`, at which point it will return `true`. If count is less than `1` or if other threads are waiting for the rendezvous under a different count then it will return `false`. Note that it is legitimate (although pathological) to supply a

count of `1`. In this latter case the rendezvous call will return true without waiting.

`getRendezvous` will return the number of threads waiting at the rendezvous identified by `identifier` or 0 if no threads are currently waiting.

rendezvousLimit will return the count associated with the rendezvous identified by `identifier` or `0` if no threads are currently waiting.

Using The Orchestration Package

Obtaining the sources

The orchestration package sources are available from the JBossTS SVN repository located under directory workspace/adinn/orchestration (they will have to move to trunk at some point). The source tree includes an ext directory containing the external antlr and ObjectWeb asm jars needed by the agent and rule code.

The package includes some sample rule scripts located in directory dd/scripts.

Building the package

The package builds to produce a single jar in build/lib/orchestration.jar. This jar contains the Java agent and rule engine code. This jar currently also bundles in the contents of the external antlr and ObjectWeb asm package libraries for ease of use. These could be unbundled so long as they are available in the classpath of the JVM being used to run the test application and the rule code.

The top level directory contains a file build.xml with default target. 'jar' which builds the orchestration jar. Other useful targets include 'parser' which rebuilds the antlr parser from the garmmar rules in dd/grammar and 'TestScript', which compiels and runs program TestScript.java. This is an offline type checker which parses and then type checks the rules defined in file handler.txt in the top level directory. Note that the latter target requires any classes mentioned in the rules to be in the class path of the TestScript program so it may be necessary to edit the rules in build.xml to get this to work.

Using The Package

Using the package is refreshingly simple in that it only requires pointing the JVM at the agent code in the jar and at the script files containing the orchestration rules. This is specified using the `java` command flag

```
-javaagent:agentlib(=script:agentscript)+
```

This is a standard option for JDK 1.5 and upwards.

`agentlib` is a path to the orchestration jar. The build process inserts a metadata file in the jar which allows the JVM to identify the agent program entry point so eberthing else si shrink-wrapped.

`agentscript` is a path to a rule script to be used during the JVM run. Multiple scripts may be provided (the brackets and + sign are regexp syntax and do not actually appear in the command flag). The name of each script file is separated from the agentlib file or preceding script file names by an `=script:`separator string. For example, setting

```
export JAVA_OPTS="-javaagent:${HOME}/jboss/workspace/adinn/
orchestration/build/lib/orchestration.jar=script:${HOME}/j
boss/workspace/adinn/orchestration/dd/scripts/HeuristicSave
AndRecover.txt"
```

will cause the JVM (and indeed JBoss AS) to pick up the jar file from the build directory and use the script provided in the dd directory.