# TOAST

# Programmer's Guide

*Version 0.1.3*

*6th January 2009*

# Table of Contents

# Introduction to TOAST

TOAST is an execution scripting tool designed primarily to support automation of tests for multi-threaded and multi-JVM Java applications (TOAST stands for Thread Orchestration Automation Scripting Tool). It uses bytecode transformation to modify the runtime execution of the application without having to modify the source code. The possible modifications are essentially unlimited but the tool provides explicit support for test automation in three main areas:

- orchestrating the timing of activities performed by independent application threads
- tracing application execution
- subverting normal method execution

## *Event Condition Action Rules*

TOAST primarily operates by introducing side-effects at specified points during execution according to an orchestration script. A script comprises a sequence of Event Condition Action (ECA) rules language which define how the application behaviour should be transformed at runtime. 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

For example, in the following example rule the event specifies a trigger point in method `get()` of class `BoundedBuffer`. The precise location of the trigger point is just before method `get()` makes a call to method `Object.wait()` — the example assumes that get() suspends its caller by calling this method if the buffer is empty.

```
RULE throw on Nth empty get
CLASS org.my.BoundedBuffer
METHOD get()
AT INVOKE Object.wait()
BIND buffer = $0
IF countDown(buffer)
DO throw org.my.ClosedException(buffer)
ENDRULE
```

The event also establishes a binding for variable `buffer` assigning it with the value $0 which refers to the recipient (`this`) argument of the `get()` call which triggered the rule. The condition invokes the TOAST built-in `countdown(Object)` which decrements a CountDown associated with the buffer — the example assumes some other rule has called `addCountDown(buffer, N)` to create this CountDown and initialise it with value `N`. The `countdown` built-in returns true when the value of the CountDown decrements to zero.

So, in this example the condition will evaluate to `false` the first N-1 times that a getter attempts to wait. At the Nth triggering the condition will evaluate to `true` and the rule will fire, running the built-in action throw. This will cause the triggering thread to throw a `ClosedException` from the call to `get()`.

Each test run is expected to supply a set of rules specifying the side-effects it wishes to introduce into the application under test. These rules are applied to any code which contains a location matching the trigger point. Rules may be supplied in a single script or as a set of scripts, the latter allowing reuse of rules across related tests.

### *Rule Bindings*

Rule conditions and actions may be parameterised by *bindings*, contextual data obtained from the code location targeted by the rule. This allows actions associated with one rule to be correlated with actions performed by other rules. It also means that actions can be specific to the particular case in hand. For example, a rule attached to a database insertion method might bind the primary key of the record being created by the method and use this binding to create a CountDown labelled with the key. Another rule attached to a database read operation might bind the primary key of the record being read, test for the presence of a CountDown labelled with the key, decrement it and throw a read exception when the CountDown reached zero. So, with this example the second rule would only act on records created during the test run and each such record would be counted down independently.

### *Built-in Conditions and Actions*

TOAST provides a suite of built-in “orchestration” conditions and actions used to coordinate the activities of independent threads e.g. delays, waits and signals, countdowns, flag operations and so on. These are particularly useful for testing multi-threaded programs subject to arbitrary scheduling orders. Judicious insertion of orchestration actions can guarantee that thread interleavings in a given test run occur in a desired order, enabling test code to reliably exercise parallel execution paths which do not normally occur with synthetic workloads.

Tracing actions are provided so that test deployment scripts can track progress of a test run and identify successful or unsuccessful test completion. Trace output can also be used to debug rule execution. The use of local context data and orchestration data in rule conditions allows trace output to be quite finely tuned. Trace actions can insert this data into message strings, allowing detailed scrutiny of test execution paths.

A few special built-in actions can be used to subvert the behaviour of application code by modifying execution paths. This is particularly important in a test environment where it is often necessary to force application methods to generate dummy results or simulate an error. A *return* action forces an early return from the code location targeted by the rule. If the method is non-void then the return action supplies a value to use as the method result. A *throw* action enables runtime exceptions (i.e. instances of `RuntimeException` or its subclasses) to be thrown from any target location, effectively either aborting a thread or, at least, where a catch-all handler is employed, the portion of the thread's call tree between the rule location and the catch block. Other exceptions may be thrown so long as the method in which the rule is located declares the exception in its throws list. Finally, a *kill* action allows a machine crash to be simulated by configuring an immediate exit from the JVM.

Finally, rules are not just restricted to using built-in operations. Application-specific side-effects can also be introduced by invoking publicly accessible application or JVM methods in rule conditions or actions.

### *Agent Transformation*

The bytecode modifications performed by TOAST are implemented using 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 TOAST agent reads the rule script at JVM bootstrap. It then monitors method code as it is loaded looking for *trigger points*, locations in the method bytecode which match the locations specified in rule events.

The agent inserts *trigger calls* into code at each point which matches a rule event. Trigger calls are calls to the rule execution engine which identify:

- the *trigger method*, i.e. the method which contains the trigger point
- the rule which has been matched
- the arguments to the trigger method

If several rules match the same trigger point then there will be a sequence of trigger calls, one for each matching rule, and rules will be triggered in the order they appear in their script(s).

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 (this) and method arguments to the rule engine and these are made available to the condition and action as default bindings with a standard naming convention, `$0`, `$1` etc. The event binding specification can introduce bindings for additional variables which are initialized by invoking methods or operations on the method bindings and/or static data. This allows arbitrary 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. Note that the agent will eventually be updated to pass local variables which are in scope at the trigger point as arguments to the trigger call, making them available as default bindings.

The agent also compiles exception handler code around the trigger calls in order to deal with exceptions which might arise during rule processing. This is not intended to handle errors detected during operation of the rule execution engine (they should all be caught and dealt with internally). Exceptions are thrown out of 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, a rule can use the `return` and `throw` built-in 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 or application-specific exception. This avoids normal execution of the remaining code in the body of the triggering method. If there are other trigger calls pending at the trigger point then these are also bypassed when a `return` or `throw` action is executed.

### *ECA Rule Engine*

The TOAST 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 using reflection. 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.

Another point to note is that 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.

## The TOAST 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*

Rule event specifications identify a specific *location* in a *target* method associated with a *target* class. Target methods can be either static or instance methods or constructors. If no detailed location is specified the default location is entry to the target method. So, the basic schema for a single rule is as follows:

```
# rule skeleton
RULE <rule name>
CLASS <class name>
METHOD <method name>
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()
AT 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 location specifier `AT LINE`. 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.

Note:

The TOAST agent *will not* transform any classes in package `java.lang` nor classes in package `org.jboss.jbossts.orchestration` (the orchestration package itself).

## *Location Specifiers*

It is easy and convenient to use line numbers to specify locations in code which is not subject to change. However, this is less useful for automated testing because modifications to the code base can shift the line numbers of unmodified lines invalidating test scripts. Luckily there are several other ways of specifying where a trigger point should be inserted into a target method which relate to the structure of the code. For example,

```
# location specifier example
RULE countdown at commit
CLASS CoordinatorEngine
METHOD commit
AT READ state
. . .
ENDRULE
```

In this rule the trigger point will be inserted just before the first location in the bytecode where a `getField` operation is performed on field called `state`. This is effectively the same as saying that the trigger point will occur at the first point in the source code of the method where field state is accessed. By contracts, the following rule would locate the trigger point after the first write to field `recovered`:

```
# location specifier example 2
RULE add countdown at recreate
CLASS CoordinatorEngine
METHOD <init>
AFTER WRITE CoordinatorEngine.recovered
. . .
ENDRULE
```

Note that in the last example the field type is qualified to ensure that the write is to the field belonging to an instance of class `CoordinatorEngine`.

The full set of location specifiers is as follows

```
AT ENTRY
AT LINE number
AT READ [type .] field [count]
AFTER READ [ type .] field [count]
AT WRITE [ type .] field [count]
AFTER WRITE [ type .] field [count]
AT INVOKE [ type .] method [ ( argtypes ) ] [count]
AFTER INVOKE [ type .] method [ ( argtypes ) ][count]
AT SYNCHRONIZE [count]
AFTER SYNCHRONIZE [count]
```

If a location specifier is provided it must immediately follow the `METHOD` specifier. If no location specifier is provided it defaults to `AT ENTRY`.

An `AT ENTRY` specifier normally locates the trigger point before the first executable instruction in the trigger method. An exception to this occurs in the case of a constructor method in which case the trigger point is located before the first instruction following the call to the super constructor or redirection call to an alternative constructor. This is necessary to ensure that rules do not attempt to bind and operate on the instance before it is constructed

An `AT LINE` specifier locates the trigger point before the first executable bytecode instruction in the trigger method whose source line number is greater than or equal to the line number supplied as argument to the specifier. If there is no executable code at (or following) the specified line number 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?).

An `AT READ` specifier locates the trigger point before the first mention of an object field whose name matches the supplied field name i.e. it corresponds to the first occurred of a corresponding `getField` instruction in the bytecode. If a type is specified then the `getField` instruction will only be matched if the named field is declared by a class whose name matches the supplied type. If a count *N* is supplied then the *N*th matching `getField` will be used as the trigger point. Note that the count identifies to the *N*th textual occurence of the field access, not the *N*th field access in a particular execution path at runtime.

An `AFTER READ` specification is identical to an AT READ specification except that it locates the trigger point after the `getField` bytecode.

`AT WRITE` and `AFTER WRITE` specifiers are the same as the corresponding `READ` specifiers except that they correspond to assignments to the named field in the source code i.e. they identify `putField` instructions.

`AT INVOKE` and `AFTER INVOKE` specifiers are like `READ` and `WRITE` specifiers except that they identify invocations of methods or constructors within the trigger method as the trigger point. The method may be identified using a bare method name or the name may be qualified by a, possibly package-qualified, type or by a descriptor. A descriptor consists of a comma-separated list of type names within brackets. The type names identify the types of the method parameters and may be prefixed with package qualifiers and employ array bracket pairs as suffixes.

`AT SYNCHRONIZE` and `AFTER SYNCHRONIZE` specifiers identify synchronization blocks in the target method, i.e. they correspond to `MONITORENTER` instructions in the bytecode. Note that `AFTER SYNCHRONIZE` identifies the point immediately after entry to the synchronized block rather than the point immediately after exit from the block.

n.b. for hysterical reasons `CALL` may be used as a synonym for `INVOKE` and the `AT` in an `AT LINE` specifier is optional.

### *Rule Bindings*

The event specification includes 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
AT READ state
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.

The special syntax `BIND NOTHING` is available for cases where the rule does not need to employ any bindings.

### *Rule Expressions*

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

- references to previously 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`, a method invocation `engine.isRecovered()`, etc.

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.
- throw and return operations are only allowed as the last action in a sequence of rule actions (see below).
- Expressions should obey the normal rules regarding associativity and 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 the synchronization block of method commit was preceded by a declaration for an `int` variable with name `idx` then the example rule should be able to include references to this variable in expressions by employing 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
AT READ state
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
AT READ state
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
AT READ state
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 a return or throw expression or a comma-separated sequence of rule expressions, possibly ending with a return or throw 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 may omit a return value if and only if the method is 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. SO, for example, the following use of return is legitimate assuming method `commit` has return type `boolean`:

```
# return example
RULE countdown at commit
CLASS com.arjuna.wst11.messaging.engines.CoordinatorEngine
METHOD commit
AT READ state
     . . .
DO debug “returning early with failure”,
    return false
ENDRULE
```

A throw expression is the `throw` keyword followed by an exception constructor expression. An exception constructor expression is the class name of the exception which is to be thrown followed by an argument list. The argument list may be empty i.e. it may consist of an open and close bracket pair. Alternatively, the brackets may include a single rule expression or a sequence of rule expressions separated by commas. If no arguments are supplied the exception type must implement an empty constructor. If arguments are supplied then the exception type must implement a constructor whose signature is type-compatible.

A throw expression causes an exception of the type named in the exception constructor to be created and thrown from the triggering method. In order for this to be valid the expression type must either be assignable to `java.lang.RuntimeException` or be explicitly declared in the triggering method's `throws` list. The type checker will throw a type exception if either of these conditions is not met. So, for example, the following use of throw is legitimate assuming method `commit` includes WrongStateException in its throws list.

```
# throw example
RULE countdown at commit
CLASS com.arjuna.wst11.messaging.engines.CoordinatorEngine
METHOD commit
AT READ state
     . . .
DO debug “throwing wrong state”,
   throw WrongStateException()
ENDRULE
```

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

# TOAST Rule Language Built-Ins

The TOAST 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 `this`. 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 debug(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   debug(“checking for recovered participant”)
     AND
     participant.isRecovered()
     AND
     debug(“recovered participant “ + participant.getId())
```

n.b. `AND` is an alternative token for the Java && operator.

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 `Rule.Helper` 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 to the rule engine simply 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 firing of some given rule will 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, one rule might include action `addCountDown($0, 1)` and another rule might include condition `countDown($0)`. A CountDown created by the first rule would only be decremented if the second rule was triggered from a method call with the same value for `this`. CountDowns created by invocations with distinct values for `this` would match up accordingly.  However, if the CountDown was identified using a common `String` literal  (i.e. action and condition were `addCountDown(“counter”, 1) and countDown(“counter”), respectively`), then the CountDown created by the first rule would be decremented by the next firing of 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 one or more rule conditions 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 signalWake(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 to associate wait and signal operations. The Helper class employs it's own private Waiter object to manage the synchronization activity.

`waitFor` is intended for use in a rule action. It suspends the current thread on the Waiter associated with the identifier until either a `signalWake` or a `signalKill` is called with the same identifier. In the former case the thread will continue processing any subsequent actions and then 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 on the relevant Waiter for a signal. It returns false if there are no threads waiting.

`signalWake` is intended for use in rule conditions or actions. If there are threads waiting on the Waiter 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 `signalWake` 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 before, 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.

### *Counters*

The rule engine provides Counters which maintain global counts across independent rule triggerings. They can be created and initialised, read, incremented and decremented in order track and respond to the number of times various triggerings or firings have happened. Note that unlike CountDowns there are no special semantics associated with decrementing a Counter to zero. They may even have negative values. The API defined by the helper class is

```
public boolean createCounter(Object o)
public boolean createCounter(Object o, int count)
public boolean deleteCounter(Object o)
public boolean readCounter(Object o)
public boolean incrementCounter(Object o)
public boolean decrementCounter(Object o)
```

As before, Counters are  identified by an arbitrary object. All methods are designed to be used in rule conditions or actions.

`createCounter` can be called to create a new Counter associated with `identifier`. If argument `count` is not supplied then the value of the new Counter defaults to 0. `createCounter` returns `true` if a new Counter was created and `false` if a Counter associated with `identifier` already exists. Note that the API is designed to ensure that race conditions between multiple threads trying to create a Counter from rule conditions can only have one winner.

`deleteCounter` can be called to delete any existing Counter associated with `identifier`. It returns `true` if the Counter was deleted and `false` if no Counter was associated with `identifier`. Note that the API is designed to ensure that race conditions between multiple threads trying to delete a Counter from rule conditions can only have one winner.

`incrementCounter` can be called to increment the Counter associated with `identifier`. If no such Counter exists it will create one with value 0 before incrementing it. It returns the new value of the Counter.

`decrementCounter` can be called to decrement the Counter associated with `identifier`. If no such Counter exists it will create one with value 0 before decrementing it. It returns the new value of the Counter.

`readCounter` can be called to read the value of the Counter associated with `identifier`. If no such Counter exists it will create one with value 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 Built-ins

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 TOAST

## *Obtaining the sources*

TOAST 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 javacup, JFlex and ObjectWeb asm jars needed by the agent and rule code.

The sources include some sample rule scripts located in directory dd/scripts.

## *Building TOAST*

TOAST 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 Jflex, javacup runtime 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 javacup/JFlex parser from the grammar rules in dd/grammar and 'TestScript', which compiles and runs program TestScript.java. The latter 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 TOAST*

Using TOAST 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.

The transformations performed by the agent can be observed by setting several environment variables which cause the transformed bytecode to be dumped to disk.

If system property

```
org.jboss.jbossts.orchestration.dump.generated.classes
```

is set (with *any* value) the agent transformer code will dump a class file containing the bytecode of any class it has modified. The class file is dumped in a directory hierarchy derived from the package of the transformed class. So, for example, class `com.arjuna.Foo` will be dumped to file `com/arjuna/Foo.class`.

If system property

```
org.jboss.jbossts.orchestration.dump.generated.classes.direct
ory
```

is set to the name of a directory writeable by the JVM then class files will be dumped in a package directory hierarchy below this directory. For example, if this property is set with value `/tmp/dump` then class `com.arjuna.Foo` will be dumped to file /tmp/dump/`com/arjuna/Foo.class.` If this property is unset or does not identify a writeable directory then class files will be dumped below the current working directory of the JVM.