Byteman Programmer’s Guide, 4.0.13, Aug 10, 2020

Table of Contents

Legal Notices

The information contained in this documentation is subject to change without notice. Red Hat Middleware makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Red Hat Middleware shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Introduction to Byteman

Byteman is a bytecode manipulation tool which makes it simple to change the operation of Java applications either at load time or while the application is running. It works without the need to rewrite or recompile the original program. In fact, Byteman can even be used to modify Java code which forms part of the Java virtual machine, classes such as String, Thread etc. Byteman employs a clear, simple and easy-to-use Event Condition Action (ECA) rule language, based on Java. ECA rules are used to specify where, when and how the original Java code should be transformed to modify its operation.

Byteman was invented primarily to support automation of tests for multi-threaded and multi-JVM Java applications using a technique called fault injection. It includes features which have been designed to resolve problems which arise with this type of testing. Byteman provides explicit support for test automation in four main areas:

- tracing execution of specific code paths and displaying application or JVM state
- subverting normal execution by changing state, making unscheduled method calls or forcing an unexpected return or throw
- orchestrating the timing of activities performed by independent application threads
- monitoring and gathering statistics summarising application and JVM operation

Byteman is actually of much more general use than as a tool for testing. The core engine which underlies Byteman is a general purpose code injection program capable of injecting inline Java code into almost any location reachable during execution of a Java method. Byteman rule conditions and actions can employ all the normal Java built-in operations in order to test and modify program state. They can also invoke methods of application or JVM instances which are in scope at the injection point.

The Byteman rule language provides a standard set of built-in operations which support specific tasks from the categories outlined above. For example, a rule condition may enforce a wait on a thread rendezvous or an action may update a statistical counter. The available set of built-in calls is configured using a POJO plugin. This can easily be replaced with an alternative implementation either for a set of rules or on a piecemeal basis for individual rules. This makes it trivial to modify and/or extend the rule language to support ad-hoc program modifications which are specific to the application domain.

Running an Application with the Byteman Agent

In order to use Byteman to test a Java application the JVM must be configured to load and run the Byteman rule engine. At the most basic level this can be achieved using the `-javaagent` command line argument. The trailing text appended to this argument points the JVM at a jar containing the Byteman rule engine. It may also identify the location of one or more Byteman rule scripts, specifying side-effects to be injected into the application under test. The engine reads these scripts when the application is launched and applies the rules contained in the scripts to any matching classes and methods. Shell command scripts are provided to simplify the task of loading the agent and installing rules into your program.

In order to simplify test automation Byteman has been integrated with the two popular testing frameworks JUnit and TestNG, allowing Byteman tests to be driven from either ant or Maven. Using the integration modules fault injection testing involves no more than annotating your program with suitable rules and ensuring that the Byteman jars are located in the classpath.

If your Java application is a long-running program then you may want to load the rule scripts and, possibly even, the rule engine after the program has started running. For example, you may decide to install Byteman into your application server when it appears to be suffering from performance problems and only then upload rules which trace execution of the code you suspect is misbehaving. Once the rule engine is loaded it cannot be unloaded. However, rules can be added and removed ad lib, allowed you to focus in on the problem in hand using more and more specific trace or monitoring rules. Note that when rules are removed the methods they affect are guaranteed to return to their original behaviour.

Full details of how to install the Byteman agent and how to upload rules either at boot time or at runtime are provided in subsection Using Byteman. Full examples of how to use Byteman from the command line, plus examples of how to configure annotation based fault injection testing from ant or Maven project, are available in the online tutorials which are linked from the documentation page of the Byteman site http://www.jboss.org/byteman.

Event Condition Action Rules

The Byteman rule engine operates by introducing side-effects at specified points during execution. A Byteman script comprises a sequence of Event Condition Action (ECA) rules which specify precisely 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 is defined by the `CLASS`, `METHOD` and `AT INVOKE` clauses. It specifies a trigger point in method `get()` of class `BoundedBuffer`. The example assumes that the definition of method `get()` includes a call to `Object.wait()`. This would be appropriate if, say, the buffer were empty. The location specifier `AT INVOKE` places the trigger point just before the call to this method.

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

The event also includes a `BIND` clause which establishes a binding for local variable buffer assigning

it with the value $this (a rather artificial thing to do but this is an example). $this is an automatic binding which refers to the target of the the get() call which triggered the rule. buffer can be referred to later in the rule; in this case it is passed as an argument to the exception constructor in the DO clause.

The condition is defined by the IF clause. It invokes the standard Byteman built-in countDown(Object) which decrements a CountDown associated with the buffer the example assumes some other rule has called createCountDown(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.

The action is defined by the DO clause. It causes an instance of ClosedException to be created and thrown from the get() call which triggered the rule.

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 to the caller of method get().

Rule Bindings and Parameterization

The corresponding rule to set up the countDown might be defined as follows

```
RULE set up buffer countDown
CLASS org.my.BoundedBuffer
METHOD <init>(int)
AT EXIT
BIND buffer = $0;
size = $1
IF $1 < 100
DO createCountDown(buffer, size - 1)
ENDRULE
```

This rule is attached to the constructor for class BoundedBuffer which takes a single int parameter. The AT EXIT location means it is triggered just before the constructor returns. The BIND clause employs the indexed notation to refer to the method target, $0, and the first argument $1. The latter is assumed for the sake of the example to be the buffer size. If the buffer size is less than 100 then the rule creates a countDown with count size –1 using the buffer as a key to identify it.

Notice how the parameterization of the createCountDown and countDown calls with the bound variable buffer restricts the scope of these rules to specific cases. If there are two buffers, buffer1 and buffer2, but only buffer1 has size less than 100 then the condition for the set up rule will return false when buffer2 is created. This means the action will not run and a countDown will not be associated with buffer2. When a subsequent call is made to buffer2.get() the throw rule will be triggered but the condition will always return false and the rule will not fire.

If instead both buffers have size less than 100 then two countDowns will be created. The throw rule will be triggered when either buffer1.get() and buffer2.get() is called and in both cases will

eventually fire and throw an exception. However, each countDown will still decrement independently.

Built-in Conditions and Actions

Byteman provides a suite of built-in 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 byteman actions can guarantee that thread interleavings in a given test run will occur in a desired order, enabling test code to reliably exercise parallel execution paths which do not normally occur with synthetic workloads.

Tracing operations are also provided so that test 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. Trace output to be quite finely tuned simply by providing a condition which tests the state of local or parameter variable bindings. Trace actions can insert these bound values 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 exceptions to be thrown from the trigger method frame. A rule is always allowed to throw a runtime exception (i.e. instances of `RuntimeException` or its subclasses). If none of the caller methods up the stack from the trigger method include a a catch for `RuntimeException` or `Throwable` then effectively this aborts the current thread. Other exceptions may also be thrown so long as the trigger method declares the exception in its throws list. This restriction is necessary to ensure that the injected code does not break the method contract between the trigger method and its callers.

Finally, a call to the `killJVM()` builtin allows a machine crash to be simulated by configuring an immediate exit from the JVM.

It is worth noting that rules are not just restricted to using built-in operations. Application- specific side-effects can also be introduced by writing object fields or calling Java methods in rule events, conditions or actions. The obvious target for such field write or method call operations is objects supplied from the triggering method via local or parameter variable bindings. However, it is also possible to update static data and invoke static methods of any class accessible from the classloader of the triggering method. So, it is quite feasible to use Byteman rules to apply arbitrary modifications to the original program. Byteman rules have special access privileges which means that it is possible to read and write protected or private fields and call protected or private data.

Extending or Replacing the Byteman Language Built-ins

Another option to bear in mind is that the set of built-in operations available to Byteman rules is not fixed. The rule engine works by mapping built-in operations which occur in a given rule to public instance methods of a helper class associated with the rule. By default, this helper class is `org.jboss.byteman.rule.helper.Helper`. which provides the standard set of built-ins designed to simplify management of threads in a multi-threaded application. For example, the builtin operations `createCountDown()` and `countDown()` used in the example rules provided above are just public methods of class Helper. The set of built-ins available for use in a given rule can by changed merely by specifying an alternative helper class for that rule.

Any class may be specified as the helper so long as it is *non-abstract* and *non-final*. Its public instance methods automatically become available as built-in operations in the rule event, condition and action. For example, by specifying a helper class which extended the default class, Helper, a rule would be able to use any of the existing built-ins and/or also make rule-specific (or application-specific) built-in calls. So, although the default Byteman rule language is oriented towards orchestrating the behaviour of independent threads in multi-threaded tests, Byteman can easily be reconfigured to support a much wider range of application requirements.

Agent Transformation

The bytecode modifications performed by Byteman 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 Byteman 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. In this case rules are mostly triggered in the order they appear in their script(s).The only exception is rules which specify an `AFTER` location, such as `AFTER READ myField` or `AFTER INVOKE` someMethod, which are executed in reverse order of appearance.

When a trigger call occurs the rule execution engine locates the relevant rule and then executes it. The rule execution engine 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. These

values may be referred to in the condition and action with a standard naming convention, $0, $1 etc. The event specification can introduce bindings for additional variables. Bindings for these variables may be initialized using literal data or by invoking methods or operations on the method parameters and/or static data. Variables bound in the event can simply be referred to by name in the condition or action. Bindings allow 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 where the trigger code is compiled with the relevant debug options enabled the agent is able to pass local variables which are in scope at the trigger point as arguments to the trigger call, making them available as default bindings. Rules may refer to in-scope variables (including the method recipient and arguments) by prefixing their symbolic names with the $ character e.g. $this, $arg1, $i etc.

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.

Agent Retransformation

The agent also allows rules to be uploaded while the application is still running. This can be used to redefine previously loaded rules as well as to introduce new rules on the fly. In cases where no currently loaded class matches the uploaded rule the agent merely adds the new rule to the current rule set. This may possibly replace an earlier version of the rule (rules are equated if they have the same name). When a matching class is loaded the latest version of the rule will be used to transform it.

In cases where there are already loaded classes which match the rule the agent will retransform them, modifying the relevant target methods to include any necessary trigger calls. If an uploaded rule replaces an existing rule in this situation then when the previous rule is deleted all trigger calls associated with it are removed from the affected target methods. Note that retransforming a class does not associate a new class object with existing instances of the class. It merely installs a different implementation for their methods.

An important point where retransformation may occur automatically without an explicit upload is during bootstrap of the agent. The JVM needs to load various of its own bootstrap classes before it can start the agent and allow it to register a transformer. Once the agent has processed the initial rule set and registered a transformer it scans all currently loaded classes and identifies those which match rules in the rule set. It automatically retransforms these classes, causing subsequent calls to bootstrap code to trigger rule processing.

ECA Rule Engine

The Byteman 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 are not type checked and compiled during trigger injection. These steps are delayed until the class and method bytecode they refer to has been loaded. Type checking requires identifying properties of the trigger class and, potentially, of classes it mentions using reflection. To do this 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 class 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.

Interpreted or Compiled Execution

In the basic operating mode, trigger calls execute a rule by interpreting the rule parse tree. It is also possible to translate a rule's bindings, condition and actions to bytecode which can then be passed by the JIT compiler. Although the default behaviour is to use the interpreter, this default can be reset by setting a system property when the agent is installed. Whatever the global setting it is also possible to select compilation or interpretation for a given rule by specifying COMPILE or NOCOMPILE in the rule definition. Although compilation takes slightly more time to execute the first time a rule is run it provides a noticeable performance pay off where the trigger method gets called many times.

Whichever mode is chosen, execution is performed with the help of an auxiliary class generated at runtime by the Byteman agent called a helper adapter. This class is actually a subclass of the helper class associated with the rule (this explains why any user-defined helper class cannot be *final*). It inherits from the helper class so that it knows how to execute built-in operations defined by the helper class. A subclass is used to add extra functionality required by the rule system, most notably method `execute0` which gets called at the trigger point and a local bindings field which stores a hashmap mapping method parameters and event variables to their bound values.

When a rule is triggered the rule engine creates an instance of the rule's helper adapter class to provide a context for the trigger call (this explains why a user-defined helper class cannot be *abstract*). It uses setter methods generated by the Byteman agent to initialise the rule and bindings

fields and then it calls the adapter instance's execute method. Since each rule triggering is handled by its own adapter instance this ensures that concurrent triggers of the same rule from different threads do not interfere with each other (also that recursive triggerings of the same rule retain their own context).

The interpreted version of `execute0` locates the triggered rule and, from there, the parse tree for the event, condition and action. It traverses the parse trees of these three rule components evaluating each expression recursively. Bindings are looked up or assigned during rule execution when they are referred to from within the rule event, condition or action. When the execute method encounters a call to a built-in it can execute this call using reflection to invoke one of the methods inherited from its helper superclass. When compilation of rules is enabled the Byteman agent generates an execute method which contains inline bytecode derived from the rule event condition and action. This directly encodes all the operations and method invocations defined in the rule. This code accesses bindings and executes built-ins in the same way as the interpreted code except that calls to built-in are compiled as direct method invocations on this rather than relying on reflective invocation.

The Byteman 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> and a class initialization method is identified using the special name <clinit>. 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 a signature comprising a parameter type list and, optionally, a return type. 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 in this example the package for class State has been

left unspecified. The type checker will infer the package of the parameter or return type from the matched method where it is omitted. 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 Byteman agent *will* not normally transform any classes in package `java.lang` and will never transform classes in package `org.jboss.byteman`, the byteman package itself (it is possible to remove the first of these restrictions by setting a System property, but you need to be really sure you know what you are doing – see below for details).
- Inner classes can be specified by employing the (internal format) `$` separator to distinguish an inner class from its enclosing outer class e.g. `org.my.List$Cons`, `Map$Entry$Wrapper`.

Class Rules vs Interface Rules

Byteman rules can be attached to interfaces as well as classes. If the `CLASS` keyword is replaced with the keyword `INTERFACE` then the rule applies to any class which implements the specified interface. For example, the following rule

```
# interface rule example
RULE commit with no arguments on any engine
INTERFACE com.arjuna.wst11.messaging.engines.Engine
METHOD commit()
. . .
ENDRULE
```

is attached to method commit of interface `Engine`. If `Engine` is implemented by classes `CoordinatorEngine` and `ParticipantEngine` then the rule implies two trigger points, one at the start of method `CoordinatorEngine.commit()` and another at the start of method `ParticipantEngine.commit()`. The agent ensures that each implementing class is transformed to include a trigger call for the rule.

Overriding Rules

Normally, Byteman only injects rule code into methods which are defined by the class identified in the `CLASS` clause. This is sometimes not very helpful. For example, the following rule is not much use:

```
RULE trace Object.finalize
CLASS java.lang.Object
METHOD finalize
IF TRUE
DO System.out.println("Finalizing " + $0)
ENDRULE
```

The print statement gets inserted into method `Object.finalize()`. However, the JVM only calls finalize when an object's class overrides `Object.finalize()`. So, this rule will not do what is intended because overriding methods will not be modified. (n.b. this is not quite the full story –

method implementations which directly override Object.finalize *and* call super.finalize() *will* trigger the rule). There are many other situations where it might be desirable to inject code into overriding method implementations. For example, class `Socket` is specialised by various classes which provide their own implementation of methods bind, accept etc. So, a rule attached to `Socket.bind()` will not be triggered when the bind method of one of these subclasses is called (unless the subclass method calls `super.bind()`).

Of course, it is always possible to define a specific rule for each overriding class. However, this is tedious and may possibly miss some cases when the code base is changed. So, Byteman provides a simple bit of syntax for specifying that rules should also be injected into overriding implementations.

```
RULE trace Object.finalize
CLASS ^java.lang.Object
METHOD finalize
IF TRUE
DO System.out.println("Finalizing " + $0)
ENDRULE
```

The `^` prefix attached to the class name tells the agent that the rule should apply to implementations of finalize defined either by class `Object` or by any class which extends `Object`. This prefix can also be used with interface rules, requiring the agent to inject the rule code into methods of classes which implement the interface and also into overriding methods on subclasses of the implementing classes.

Note that if an overriding method invokes a super method then this style of injection may cause the injected rule code to be triggered more than once. In particular, injecting into constructors (which, inevitably, invoke some form of super constructor) will often result in multiple triggerings of the rule. This is easily avoided by adding a condition to the rule which checks the name of the caller method. So, for example, the rule above would be better rewritten as

```
RULE trace Object.finalize at initial call
CLASS ^java.lang.Object
METHOD finalize
IF NOT callerEquals("finalize")
DO System.out.println("Finalizing " + $0)
ENDRULE
```

This rule uses the built-in method `callerEquals` which can be called with a variety of alternative signatures (described in full below). This version calls `String.equals()` comparing the name of the method which called the trigger method to its String argument and returns the result. The condition negates this using the `NOT` operator (another way of writing the Java `!` Operator). So, when an implementation of `finalize` is called via the finalizer thread's `runFinalizer()` method this condition evaluates to true and the rule fires. When it gets called via `super.finalize()` the condition evaluates to false and the rule does not fire.

Overriding Interface Rules

The ^ prefix can also be used in combination with INTERFACE rules. Normally an interface rule is only injected into classes which directly implement the interface methods. This can mean that a plain INTERFACE rule does not always get injected into the classes you are interested in.

For example, class ArrayList extends class AbstractList which, in turn, implements interface List. A rule attached to INTERFACE List *will* be considered for injection into AbstractList but will *not* be considered for injection into ArrayList. This makes sense because AbstractList will contain an implementation of every method in List (some of these methods may be abstract). So, any methods in class ArrayList which re-implement the interface are considered to be overriding methods. However, the ^ prefix can be used to achieve the desired effect. If the rule is attached to INTERFACE ^List then it *will* be considered for injection into both AbstractList and ArrayList.

Note that there is a subtle difference between these cases where a class extends a superclass and those where an interface extends a superinterface. The same class hierarchy can be used as an example to explain how interface extension is treated.

Let's look at the interface Collection which is extended by interface List. When a rule is attached to INTERFACE Collection then it is considered for injection into any class which implements Collection and also any class which implements an extension of Collection. Since List extends Collection this means that an implementation class like AbstractList will be a candidate for the rule. This is because AbstractList is the first class reached down the chain from Collection via List so it is the first point in the class hierarchy where an implementation can be found for methods of Collection (even if it is only an abstract method). Class ArrayList willl not be a candidate for injection because any of it's methods which re-implement a method declared by Collection will still only override a method implemented in AbstractList. If you want the rule to be injected into these overriding methods defined in class ArrayList then you can do so by attaching the rule to INTERFACE ^Collection.

Location Specifiers

The examples above either specified the precise location of the trigger point within the target method to a specific line number using AT LINE or defaulted it to the start of the method. Clearly, line numbers can be used to specify almost any point during execution and are easy and convenient to use in code which is not subject to change. However, this approach is not very useful for test automation where the code under test may well get modified. Obviously when code is edited the associated tests need to be revised. But modifications to the code base can easily shift the line numbers of unmodified code invalidating test scripts unrelated to the edits. Luckily, there are several other ways of specifying where a trigger point should be inserted into a target method. For example,

```
# location specifier example
RULE countdown at commit
CLASS CoordinatorEngine
METHOD commit
AFTER WRITE $current
. . .
ENDRULE
```

The name current prefixed with a `$` sign identifies a local variable,or possibly a method parameter. In this case, current happens to be a local variable declared and initialised at the start of method `CoordinatorEngine.commit` whose type is the enum `State`.

```
public State commit()
{
  final State current ;
  synchronized(this)
  {
    current = this.state ;
    if (current == State.STATE_PREPARED_SUCCESS) {
      . . .
```

So, the trigger point will be inserted immediately after the first write operation in the bytecode (istore) which updates the stack location used to store current. This is effectively the same as saying that the trigger point will occur at the point in the source code where local variable current is initialised i.e. the first line inside the synchronized block.

By contrast, the following rule would locate the trigger point after the first read from field recovered:

```
# location specifier example 2
RULE add countdown at recreate
CLASS CoordinatorEngine
METHOD <init>
AT READ 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`. Without the type the rule would match any read from a field with name `recovered`.

The full set of location specifiers is as follows:

```
AT ENTRY
AT EXIT
AT LINE number
AT READ [type .] field [count | ALL ]
```

```
AT READ $var-or-idx [count | ALL ]
AFTER READ [ type .] field [count | ALL ]
AFTER READ $var-or-idx [count | ALL ]
AT WRITE [ type .] field [count | ALL ]
AT WRITE $var-or-idx [count | ALL ]
AFTER WRITE [ type .] field [count | ALL ]
AFTER WRITE $var-or-idx [count | ALL ]
AT INVOKE [ type .] method [ ( argtypes ) ] [count | ALL ]
AFTER INVOKE [ type .] method [ ( argtypes ) ][count | ALL ]
AT NEW [ type ] [ [] ] * [count | ALL ]
AFTER NEW [ type ] [ [] ] * [count | ALL ]
AT SYNCHRONIZE [ count | ALL ]
AFTER SYNCHRONIZE [ count | ALL ]
AT THROW [count | ALL ]
AT EXCEPTION EXIT
```

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

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.

AT EXIT

An `AT EXIT` specifier locates a trigger point at each location in the trigger method where a normal return of control occurs (i.e. wherever there is an implicit or explicit return but not where a throw exits the method).

AT LINE

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 print an error in such cases because this may merely indicate that the rule does not apply to this particular class or method).

AT READ

An `AT READ` specifier followed by a field name 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 Nth matching getField will be used as the trigger point. Note that the count identifies to the Nth textual occurence of the field access, not the Nth field

access in a particular execution path at runtime. If the keyword ALL is specified in place of a count then the rule will be triggered at all matching getField calls.

An `AT READ` specifier followed by a $-prefixed local variable name, method parameter name or method parameter index locates the trigger point before the first instruction which reads the corresponding local or method parameter variable i.e. it corresponds to an iload, dload, aload etc instruction in the bytecode. If a count N is supplied then the Nth matching read will be used as the trigger point. Note that the count identifies to the Nth textual occurrence of a read of the variable, not the Nth access in a particular execution path at runtime. If the keyword ALL is specified in place of a count then the rule will be triggered before every read of the variable.

Note that it is only possible to use local or parameter variable names such as $i, $this or $arg1 if the trigger method bytecode includes a local variable table, e.g. if it has been compiled with the -g flag. By contrast, it is always possible to refer to parameter variable read operations using the index notation `$0`, `$1` etc (however, note that location `AT READ $0` will only match where the trigger method is an instance method).

AFTER READ

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

AT WRITE, AFTER WRITE

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

Note that location `AT WRITE $0` or, equivalently, `AT WRITE $this` will never match any candidate trigger method because the target object for an instance method invocation is never assigned.

Note also that for a given local variable, localvar, location `AT WRITE` $localvar or, equivalently, `AT WRITE` $localvar 1 identifies the location immediately after the local variable is initialised i.e. it is treated as if it were specified as `AFTER WRITE` $localvar. This is necessary because the variable is not in scope until after it is initialised. This also ensures that the local variable which has been written can be safely accessed in the rule body.

AT INVOKE, AFTER INVOKE

`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 NEW, AFTER NEW

`AT NEW` and `AFTER NEW` specifiers identify locations in the target method where a `new` operation creates a Java object class or array class. An `AT NEW` rule is triggered before the object or array is

allocated. An `AFTER NEW` rule is triggered after creation and initialization of the object or array.

Selection of the `NEW` trigger location may be constrained by supplying a variety of optional arguments, a type name, one or more pairs of square braces and either an integer count or the keyword `ALL`. These arguments may all be specified independently and they each serve to select a more or less precise set of matches for points where the rule may be considered for injection into the target method.

If a type name is supplied injection is limited to points where an instance (or array) of the named type is created. The type name can be supplied without a package qualifier, in which case any new operation with a type sharing the same non-package qualified name will match.

If the type name is omitted then injection can occur at any point where an instance (or array) is created.

Note that `extends` and `implements` relationships are ignored when matching. For example, if a rule specifies `AT NEW Foo` then the location will not be matched against operation `new Foobar` even if `FooBar extends Foo`. Similarly, when `Foo implements IFoo` specifying location `AT NEW IFoo` will not be matched. Indeed specifying any interface is a mistake. new operations always instantiate a specific class and never an interface. So, locations specifying an interface name will never match.

If one or more brace pairs are included then injection is limited to points in the method where an array with the equivalent number of dimensions is created. So, for example specifying `AT NEW [][]` will match any new operation where a 2d array is created, irrespective of what the array base type is, By contrast, specifying `AT NEW int[]` will only match a new operation where a 1d int array is created. If no braces are supplied then matches will be restricted to new operations where a Java object class (i.e. a non-array class) is instantiated.

When there are multiple canidate injection points in a method an integer count may be supplied to pick a specific injection point (count defaults to 1 if it is left unspecified). Keyword `ALL` can be supplied to request injection at *all* matching injection points.

AT SYNCHRONIZE, AFTER SYNCHRONIZE

`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.

AT THROW

An `AT THROW` specifier identifies a throw operation within the trigger method as the trigger point. The throw operation may be qualified by a, possibly package-qualified, typename identifying the lexical type of the thrown exception. If a count N is supplied then the location specifies the Nth textual occurrence of a throw. If the keyword ALL is specified in place of a count then the rule will be triggered at all matching occurrences of a throw.

AT EXCEPTION EXIT

An `AT EXCEPTION EXIT` specifier identifies the point where a method returns control back to its caller

via unhandled exceptional control flow. This can happen either because the method itself has thrown an exception or because it has called out to some other method which has thrown an exception. It can also happen when the method executes certain operations in the Java language, for example dereferencing a null object value or indexing beyond the end of an array.

A rule injected with this location is triggered at the point where the exception would normally propagate back to the caller. Once rule execution completes then normally the exception flow resumes. However, the rule may subvert this resumed flow by executing a RETURN. It may also explicitly rethrow the original exception or throw some newly created exception by executing a THROW (n.b. if the latter is a checked exception then it must be declared as a possible exception by the trigger method).

n.b. when several rules specify the same location the order of injection of trigger calls usually follows the order of the rules in their respective scripts. The exception to this is AFTER locations where the the order of injection is the reverse to the order of occurrence.

n.b.b. when a location specifier (other than ENTRY or EXIT) is used with an overriding rule the rule code is only injected into the original method or overriding methods if the location matches the method in question. So, for example, if location AT READ myField 2 is employed then the rule will only be injected into implementations of the method which include two loads of field myField. Methods which do not match the location are ignored.

n.b.b.b. for historical reasons `CALL` may be used as a synonym for `INVOKE`, `RETURN` may be used as a synonym for `EXIT` 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. Alternatively, the `BIND` clause may be omitted.

Downcasts At Rule Variable Initialization

A binding initialization can do more than simply introduce a place holder for the value computed in the initializer expression and in this respect it differs significantly from an assignment that occurs elsewhere in the rule body. It is possible to perform a 'downcast' in a binding initialization i.e. assigning a value of some generic class type to a rule variable whose type is a compatible subclass type.

For example, in the following rule

```
# downcast example
RULE countdown at commit
CLASS com.arjuna.wst11.messaging.engines.CoordinatorEngine
METHOD commit
AT READ state
BIND engine:CoordinatorEngine = $0;
     endpoint : javax.xml.ws.EndpointReference = engine.participant;
     w3cEndpoint javax.xml.ws.wsaddressing.W3CEndpointReference = endpoint;
. . .
ENDRULE
```

the reference stored in the `CoordinatorEngine` field `participant` is used to initialize rule variable `endpoint` whose type is the generic JaxWS class `EndpointReference`. The second binding for rule variable `w3cEndpoint` uses the value stored in `endpoint`. The type of this second variable `w3cEdpoint` is subclass `W3CEndpointReference` of the type `EndpointReference` of the initializing expression. In an assignment anywhere else in a rule this would lead to a type error. The Byteman type checker ignores the type mismatch in this initializing assignment, but only because it knows that `W3CEndpointReference` is a subclass of the type for the intiializer expression, `endpoint`. It assumes that the 'downcast' at this point is deliberate, i.e. that the rule author knows that the value returned by the initializer expression will in fact belongs to the subtype.

Byteman still performs a type check when it executes the initialization to ensure that the value is indeed of the required type, throwing an exception if the test fails. n.b. in this case the assignment will never fail because `CoordinatorEngine` field `participant` is actually declared as a `W3CEndpointReference`.

Downcasting is particularly useful when rules need to handle generic types like `List` etc. The Byteman type checker cannot identify information about generic types from bytecode because it is erased at compile time. So, for example, a list `get` operation will always be typed as returning an Object. If you know that a specific list available at the injection point stores values of some given type then a value retrieved from the list can be downcast to the desired type in the BIND clause.

Rule Expressions

Expressions which occur on the right hand side of the = operator in event bindings can be any of the Java expressions supported by Byteman. This includes all the usual simple expressions found in plus a few extra special cases i.e.

- references to previously bound variables
- references to the trigger method recipient or parameters
- references to the local variables in scope at the trigger point
- references to special variables `$!`, `$^`, `$#`, `$*`, `$@`, `$CLASS` and `$METHOD`
- static field references
- primitive literals
- array literals
- field accesses
- static or instance method invocations
- built-in operation invocations

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

Expressions can also be constructed as complex expressions composed from other expressions using the usual Java operators: `+`, `-`, `*`, `/`, `%`, `&`, `|`, `^`, `&&`, `||`, `!`, `=`, `==`, `!=`, `<`, `<=`, `>`, `>=`, `new`, 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:

- `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.
- The trigger method recipient and parameters may be referred to by index using the symbols `$0` (invalid for a static method), `$1` etc. If the method has been compiled with the relevant debug options then symbolic references may also be used. So, for example, `$this` may be used as an alias for `$0` and `$myArg` may be used as an alias for `$1` if the method first parameter is declared with name `myArg`,
- If the trigger method has been compiled with the relevant debug options then local variables may be referenced symbolically using the same syntax to method parameters. So, for example, if variable `idx` is in scope at the trigger point then `$idx` can be used to obtain its value.
- Special variables provide access to other trigger method data. There are currently 7 such special variables:
  - `$!` is valid in at `AT EXIT` rule and is bound to the return value on the stack at the point where the rule is triggered. Its type is the same as the trigger method return type. The rule will fail

to inject if the trigger method return type is void.

- `$!` is also valid in an `AFTER INVOKE` rule and is bound to the return value on the stack at the point where the rule is triggered. Its type is the same as the invoked method return type. The rule will fail to inject if the invoked method return type is void.
- `$!` is also valid in an `AFTER NEW` rule and is bound the instance or array created by the new operation which triggered the rule. Its type is that of the corresponding new expression in the trigger method.
- `$^` is valid in an `AT THROW` rule and is bound to the throwable on the stack at the point where the rule is triggered. Its type is Throwable.
- `$^` is also valid in an `AT EXCEPTION EXIT` rule and is bound to the throwable being returned from the method via exceptional control flow. Its type is Throwable.
- `$#` has type int and identifies the number of parameters supplied to the trigger method.
- `$*` is bound to an `Object[]` array containing the trigger method recipient, `$this`, in slot 0 and the trigger method parameter values, `$1`, `$2` etc in slots 1, 2 etc (for a static trigger method the value in slot 0 is `null`).
- `$@` is only valid in an `AT INVOKE` rule and is bound to an `Object[]` array containing the `AT INVOKE` target method recipient in slot 0 and the call arguments for the target method installed in slots 1 upwards in call order (if the target method is static the value in slot 0 is `null`). Note that this variable is not valid in `AFTER INVOKE` rules. The array contains the call arguments located on the stack just before the trigger method calls the AT INVOKE target method. These values are no longer available after the call has completed.
- `$CLASS` is valid in all rules and is bound to a `String` whose value is the full package qualified name of the trigger class for the rule. The trigger class is the class whose method the rule has been injected into. Note that this is normally the same as the target class mentioned in the `CLASS` clause of the rule. However, when injecting into interfaces or using overriding injection the trigger class may be an implementation or subclass, respectively, of the target class. So there may be more than one trigger class for any given target class.
- `$METHOD` is valid in all rules and is bound to a `String` whose value is the full name of the trigger method into which the rule has been injected, qualified with signature and return type. Note that this is normally the same as the target method mentioned in the `METHOD` clause of the rule. However, the target method may omit the signature and return type. So there may be more than one trigger method for any given target method.
- `$NEWCLASS` is only valid in `AT NEW` and `AFER NEW` rules. It is bound to a String which is the canonical name of the object or array created by the new operation e.g `org.my.Foo`, `int[]`, `org.my.Bar[][]`.

- Array literal expressions are a comma-separated sequence of expressions enclosed in braces such as `{}` , `{ "foo", "bar" }`. Array literals may only be used to define the initial value for either: an array variable declared in the `BIND` clause e.g.
  `x:int[] = {1, 2, 3};`
  or: an array created via a new expression e.g.
  `names = new Object[][] {{$0, $0.name()},{$1, $1.name()}};`
  n.b. Byteman does not restrict the type of expressions embedded in the initializer to other literals. As you can see in the second example above embedded values can be computed using

arbitrary Java expressions. Byteman also allows subordinate bracketed terms to have different numbers of items so long as the expressions are type compatible; it simply creates the relevant sub-array using the number of values provided.
n.b.b. when an initializer with mixed value types is used to initialize an untyped variable in a `BIND` clause the values must have a type uniform with that of the first element, which can be used consistently to infer the corresponding array type e.g given the following binding `x = { $1, "foo", $2 };`
type `Object[]` will be inferred for `x` if `$1` is of type `Object`, type `String[]` will be inferred if `$1` and `$2` are both of of type `String`. With any other types a type error will occur.

- Assignments may update the bindings for rule variables introduced in the `BIND` clause, parameter or local variables, instance fields, static fields or the return value special variable `$!`. Assignments are not currently allowed to update any of the the other special variables.
- Assignments to parameter variables or local variables are visible on resumption of the trigger method. For example, assume that a rule includes an assignment such as `$name = "Ernie"`, where name is either a parameter variable or a local variable in scope at the trigger point. If name has value `"Bert"` when the rule is triggered and the assignment actually gets executed then on resumption of the trigger method `name` will have value `"Ernie"`. Note that assignments cannot be made to `$this` (or, equivalently, `$0`); the recipient argument for an instance method is always `final`.
- Assignment to `$!` in an AT RETURN, AFTER INVOKE or AFTER NEW rule updates the value on top of the stack at the trigger point. For an AT RETURN rule this causes the trigger method to return this updated value. The same effect can be achieved in an AT RETURN rule by executing a `RETURN` expression. For an AFTER INVOKE or AFTER NEW rule this substitutes an alternative result for the just completed call or new operation.
- Byteman provides the English language keywords listed below which can be used in place of the related standard Java operators (in brackets):
`OR (||)`, `AND (&&)`, `NOT (!)`, `LE (< =)`, `LT (<)`, `EQ (==)`, `NE (!=)`, `GE (>=)`, `GT (>)`, `TIMES (*)`, `DIVIDE (/)`, `PLUS (+)`, `MINUS (-)`, `MOD (%)`, Keywords are recognised in either upper or lower (but not mixed) case.
Keywords may clash with the same names where they they occur as legal Java identifiers in the target classes and methods specified in Byteman rules

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 = $this;
     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 = $this,
     . . .
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 = $this,
. . .
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 action or a sequence of rule

expressions separated by semi-colons, possibly ending with a return or throw action. Rule expressions occurring in an action list may have arbitrary type, including void type.

A return action is the `return` keyword possibly followed by a rule expression which is used to compute a return value. A return action 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 action is the `throw` keyword followed by an throwable constructor expression. A throwable constructor expression is the keyword `new` followed by the class name of the throwable 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 throwable type must implement an empty constructor. If arguments are supplied then the throwable type must implement a constructor whose signature is type-compatible. n.b. for hysterical reasons the new keyword may be omitted from the throwable constructor expression which follows the throw keyword.

A throw action causes a throwable 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 `java.lang.Error` or be explicitly declared as a checked exception 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 new WrongStateException()
ENDRULE
```

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

Built-In Calls

Built-in calls are calls to a family of useful methods which implement a family operations that are often useful in rule conditions or actions. They 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 instance methods of a helper class, by default the class Helper provided by Byteman itself. So, referring back to the last few examples, it is apparent that the class Helper implements a debugging method with signature

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

This method prints the supplied string to `System.out` but only when property `org.jboss.byteman.debug` has been set. It can be used in a rule action to display a message when you wish to debug rule execution, for example:

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

So, in this example 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 implemented by the corresponding instance method of `Helper`. It can be used to perform an immediate halt of the JVM, simulating a JVM crash.

Note that method `debug` has a `boolean` return type. It always returns true. This is to allow tracing of rule IF clauses by `AND` in a `debug` call with the rest of the condition. 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 `Helper` as built-in operations. So when the rule type checker encounters an invocation of `debug` with no recipient supplied it verifies that `debug` is a method of class Helper and automatically type checks the call against this method. At execution time the call is executed by invoking the implementation of `debug` on a helper instance created when rule execution is triggered at the injection point.

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

User-Defined Rule Helpers

A rule can specify it's own helper class if it wants to extend, override or replace the set of built-in calls available for use in its event, condition or action. For example, in the following rule, class `FailureTester` is used as the helper class. Its boolean instance method `doWrongState(CoordinatorEngine)` is called from the condition to decide whether or not to throw a `WrongStateException`.

```
# helper example
RULE help yourself
CLASS com.arjuna.wst11.messaging.engines.CoordinatorEngine
METHOD commit
HELPER com.arjuna.wst11.messaging.engines.FailureTester
AT EXIT
IF doWrongState($0)
DO throw new WrongStateException()
ENDRULE
```

A helper class does not need to implement any special interface or inherit from any pre-defined class. It merely needs to provide instance methods to resolve the built-in calls which occur in the rule. The only limitations are

- your helper class must not be final
  - Byteman needs to be able to subclass your helper in order to interface it to the rule execution engine
- your helper class must not be abstract
  - Byteman needs to be able to instantiate your helper when the rule is triggered
- you must provide a suitable public constructor for your helper class
  - by default Byteman will instantiate it using the empty constructor (i.e. the one with signature `()`)
  - if you provide a constructor that accepts the rule as argument (i.e. with signature `(org.jboss.byteman.agent.rule.Rule)`) Byteman will use that for preference

By sub-classing the default helper it is possible to extend or override the default set of methods. For example, the following rule employs a helper which adds emphasis to the debug messages printed by the rule.

```
# helper example 2
RULE help yourself but rely on others
CLASS com.arjuna.wst11.messaging.engines.CoordinatorEngine
METHOD commit
HELPER HelperSub
AT ENTRY
IF NOT flagged($this)
DO debug("throwing wrong state");
   flag($this);
   throw new WrongStateException()
ENDRULE
```

```
class HelperSub extends Helper
{
    public HelperSub(Rule rule)
    {
      super(rule);
    }
    public boolean debug(String message)
    {
        super("!!! IMPORTANT EVENT !!! " + message);
    }
}
```

The rule is still able to employ the built-in methods flag and flagged defined by the default helper class.

The examples above use a `HELPER` line in the rule body to reset the helper for a specific rule. It is also possible to reset the helper for all subsequent rules in a file by adding a `HELPER` line outside of the scope of a rule. So, in the following example the first two rules use class `HelperSub` while the third one uses class `YellowSub`.

```
HELPER HelperSub
# helper example 3
RULE helping hand
. . .
RULE I can't help myself
. . .
RULE help, I need somebody
CLASS . . .
METHOD . . .
HELPER YellowSub
. . .
```

Rule Helper Lifecycle Methods

It is occasionally useful to be able to perform some sort of setup activity when rules are loaded or a teardown activity when rules are unloaded. For example, if tracing rules are loaded to collect statistics on program execution it would be convenient to add a background thread which wakes up at regular intervals in order to print and then zero the value of the various counters incremented by the rules. Similarly, it would be helpful to be able to detect that all the tracing rules had been unloaded so that the thread can be shutdown, avoiding wasted CPU time. The rule engine supports a lifecycle model for loading and unloading which makes this sort of setup and teardown simple to achieve.

There are four lifecyle events in the model: activate, install, uninstall and deactivate. Although these lifecycle events are associated with loading and unloading of rules, the focus of the event is the helper class associated with the rule being loaded/unloaded. It is the helper class which provides a callback method to handle the lifecycle event. The four lifecycle events are generated according to the following model.

Assume that we have a helper class *H* and a set of *installed* rules *R(H)* employing *H* as their helper. Obviously *R(H)* is empty at bootstrap. When a rule *r(H)* with helper H has been loaded (either during agent bootstrap or via the dynamic listener), injected and typechecked it is installed into the set *R(H)*. When an installed rule is unloaded via the dynamic listener it is uninstalled from the set *R(H)*.

- an activate event occurs when an install cause *R(H)* to transition from empty to non-empty
- an install event occurs when *r(H)* is installed in *R(H)*
- an uninstall event occurs when *r(H)* is uninstalled from *R(H)*
- a deactivate event occurs an uninstall causes *R(H)* to transition from non-empty to empty

Note that an install always generates an activate event before the associated install event. An uninstall always generates an uninstall event before any associated deactivate event.

The helper class *H* is notified of these events if it implements any of the corresponding static methods:

```
public static void activated()
public static void installed(Rule rule)
public static void installed(String ruleName)
public static void uninstalled(Rule rule)
public static void uninstalled(String ruleName)
public static void deactivated()
```

`activated()` is called when an activate event occurs. It can perform a one-off set up operation on behalf of all rules employing the helper.

`deactivated()` is called when a deactivate event occurs. It can perform a one-off tear down operation on behalf of all rules employing the helper.

`installed(Rule)` is called when an install event occurs. It can perform a set up operation specific to the supplied rule.

`deinstalled(Rule)` is called when an install event occurs. It can perform a tear down operation specific to the supplied rule.

`installed(String)` and `uninstalled(String)` can also be implemented as alternatives to `installed(Rule)` and `uninstalled(Rule)` if the helper can make do with the rule name rather than using the `Byteman Rule` instance. Note that if both flavours are implemented only the method which takes a `Rule` will be called.

Note that the default helper class, *Helper*, implements these lifecycle methods. In particular, its implementation of *deactivated* clears any resources allocated during rule execution, such as counters, flags, trace streams, etc, since it can be sure that there are no longer any installed rules relying on them.

It is important to understand that loading and unloading of a rule does not always initiate lifecycle processing. If a rule does not parse or typecheck correctly it will not be installed so it will not generate activate or install events. If later this rule is unloaded it will not generate uninstall or deactivate events since it was never installed. It is also possible for a valid rule to be loaded and unloaded without initiating lifecycle processing. For example, the rule may never get injected because no matching trigger class has been loaded into the JVM. Finally, when a rule is resubmitted and, hence, redefined the agent will normally elide an uninstall and reinstall associated with removing the old version of the rule and injecting the new version. Of course, if the new version of a rule fails to parse, inject or type check correctly then an uninstall will be performed (assuming that the old version of the rule was actually installed).

Helper Lifecycle Method Chaining

Although lifecycle methods are static (i.e. associated with the class rather than with an instance) it is necessary to propagate lifecycle events up the superclass hierarchy, chaining calls to lifecycle methods where present not just on the immediate helper class but also on its parent classes. Byteman ensures that all available implementations of methods activated and installed are called in response to the associated lifecyle events, searching for implementations starting with the immediate helper class of the rule and working up the super hierarchy. Similarly, it ensures that all available implementations of methods uninstalled and deactivated are called in response to the associated lifecyle events searching for implementations starting at the top of the helper super class hierarchy and working down to the rule's immediate helper class.

This is necessary to ensure that each superclass of the helper that is interested in tracking rule lifecyle events is aware of the state of all rules that may be relying on it as a helper. If a rule *r* has helper *H* which inherits from helpers *H'*, *H''*, etc then at install time *r* has effectively been installed into all the associated sets *R(H)*, *R(H')*, *R(H'')*, etc and at deinstall time it has effectively been removed from these sets. In consequence, Byteman must perform lifecycle processing for each of the helpers *H*, *H'*, *H''*, etc that implements lifecycle callbacks.

Two examples may help clarify why this is needed. First, consider a rule *r* which uses helper *H* where *H* specialises some other helper class *H'*. Let's also assume that *H* does not implement any lifecycle methods while *H'* implements both *activated* and *deactivated*, respectively, to create and

destroy a hashmap used by some of its builtin methods. If *r* gets injected before any other rules using *H'* then it is no good just counting it as installed into *R(H)* and, finding no lifecycle methods, ignoring the install. This will fail to run method *H'.activated*. If *r* executes a builtin of *H'* that uses the hashmap it will suffer a null pointer exception.

Conversely, assume that a rule *r'* using *H'* is installed first, followed by rule *r* and that *r'* is then uninstalled. If *r* is only included in *R(H)* and not *R(H')* then deinstall of *r'* will transition *R(H')* to empty, deactivating *H'* and deleting the hashmap. Once again, if rule *r* calls a builtin of *H'* which uses the hashmap it will suffer a null pointer exception.

It is not enough for Byteman simply to find the first helper or super which implements activated and expect the helper to perform the call chaining. This can be seen if the above example is changed so that *H* implements *activated* and *deactivated* to call the corresponding method of *H'*. The problem is that the installed and uninstalled counts will still not be correctly updated. In the second scenario, the *activated* callback for *H'* will be called when *r'* is installed (because *R(H')* transitions from empty to non-empty) and called again, this time indirectly from *H.activate*, when *r* is installed (because *R(H)* transitions from empty to non-empty). Also, when *r'* is uninstalled *H'.deactivated* will be called (as *R(H')* transitions back to empty), deleting the hashmap even though rule *r* is still depending on it.

Rule Compilation

By default rules are executed by Byteman's own interpreter which interprets the rule's parse tree. However, it is also possible to translate a rule's bindings, condition and actions to bytecode which can then be executed directly by the JVM and, potentially, optimized by the JIT compiler. This is very useful when rule code is injected into a method which is called very frequently. The default execution mode can be switched from interpreted to compiled by setting system property org.jboss.byteman.compile.to.bytecode when the agent is loaded. However, resetting the default for all rules may not be helpful as it is nor always worth incurring the overhead of compiling rules which only get executed once. So, it is also possible to request compilation for a group of rules within a rule script or even for individual rules.

The following rule uses a COMPILE clause to ensure that it is always translated to bytecode even when the default mode is interpreted.

```
# this rule will always be compiled
RULE compile example
CLASS com.arjuna.wst11.messaging.engines.CoordinatorEngine
METHOD prepare
COMPILE
AT ENTRY
. . .
ENDRULE
```

If you have reset the default to compiled and wish to disable compilation then a NOCOMPILE clause can be used. The same two clauses can also appear outside a rule at the top level in a script (just as with HELPER clauses). This resets the default for subsequent rules in the same script.

```
# set default execute mode to compile
COMPILE
# this rule will be compiled
RULE compile example 2
CLASS com.arjuna.wst11.messaging.engines.Engine
METHOD prepare
AT ENTRY
. . .
ENDRULE
# this rule will never be compiled
RULE compile example 3
CLASS com.arjuna.wst11.messaging.engines.CoordinatorEngine
METHOD prepare
NOCOMPILE
AT ENTRY
. . .
ENDRULE
# set default execute mode to interpreted
NOCOMPILE
# this rule will be interpreted
RULE compile example 4
CLASS com.arjuna.wst11.messaging.engines.ParticipantEngine
METHOD commit
AT ENTRY
. . .
ENDRULE
```

Module Imports

NOTE	This section documents an early-preview Byteman feature which is incomplete and may be subject to change.

When a Byteman rule is injected into a method the injected code needs to be resolved against the values and types available in the injection context. For example, if the rule is injected into method `charAt` of class `String` then a mention of parameter variable `$1` is typed by checking the type signature of the trigger method and noting that it is an `int`. The injected code loads the integer value from slot 1 of the local area of a trigger method invocation and passes it to the rule execution engine.

Similarly, a call to `Thread.currentThread()` is type checked by noting that the name preceding the method invocation is a reference to a class named `java.lang.Thread` and looking up first the class and then the method in order to identify that the return type is also of type `Thread`. The rule execution engine executes the resolved method in order to compute the value of this expression.

Resolution of type names to classes requires a classloader lookup to see whether a class with the given name is `in scope'. Byteman interprets what it means for a name to be `in scope' by looking up class names using the classloader of the *trigger* class i.e. the class which owns the method into which the rule is being injected.

In normal Java SE deployments this normally presents no issues because application and JDK runtime classes are all visible via the classpath. So, in the following example, the code injected into application class `ThreadPool` can directly refer to application class `Logger` and invoke one of its static methods

```
RULE call out to logging method
CLASS org.my.ThreadPool
METHOD schedule(Runnable)
BIND runnableKlazz = $1.getClass().getName()
IF TRUE
DO org.my.Logger.log(runnableKlazz, "scheduled: " + System.currentTimeMillis())
```

Since both classes are deployed on the classpath they will both be loaded by the system classloader. When Byteman tries to inject the rule into class `org.my.ThreadPool` it uses that loader to lookup class `org.my.Logger` and finds the desired class. Note that package and class privacy imposes no barriers to Byteman here. The rule could call Logger.log even if it were private.

In Java EE deployments this sort of cross-jar or cross-deployment reference may not work. If the two classes above are deployed in separate war files then the classloader for `org.my.ThreadPool` may not be able to resolve references to class `org.my.Logger`. Indeed, if you employ a module system like JBoss Modules or OSGi then you may not even be able to resolve references to classes on the system or bootstrap classpath when you are injecting to code which is deployed via a JBoss or OSGi module loader.

Byteman provides a way to overcome this problem using an `IMPORT` declaration. Note that this declaration is *not* to be confused with Java's `import` statement. `IMPORT` is only appropriate when injecting into a class which belongs to a module. Its purpose is to ensure that classes from other modules are `in scope' for rule code. Type references which are not visible from the target class's loader may still be resolved against types provided by the module whose name follows the `IMPORT` keyword.

So, for example, let's assume class `org.my.ThreadPool` is deployed in a JBoss EE depoyment which does not have access to the transactions API. This is a reasonable assumption given that the thread pool has no reason to import the TX module nor to use any of the transaction annotations which would lead to the TX module being auto-imported. The following rule explicitly imports the JBoss Module bringing class `TransactionManager` into scope. This allows the trace call to include details of any active transaction in the record of schedule operations.

```
RULE log thread schedule operations with details of current TX
CLASS org.my.ThreadPool
METHOD schedule(Runnable)
IMPORT javax.transaction.api
BIND runnableKlazz = $1.getClass().getName()
IF TRUE
DO traceln(runnableKlazz + "scheduled at " +
           System.currentTimeMillis() + in TX " +
           javax.Transaction.TransactionManager.getTransaction())
```

Note that the format of the module name following the IMPORT keyword is specific to the module system being used. In this case the name identifies the module used by JBoss EAP to deploy the Java EE Transactions API classes.

It is possible to import more than one module by adding repeated `IMPORT` declarations. Also, when writing a script imports may be written at top level (outside of rule scope) accumulating a list of imports that apply to subsequent rules. An empty IMPORT statement at top level will clear the current accumulated imports list. An empty IMPORT statement in the body of a RULE will clear any script level imports just for that rule.

```
# import the TX and JPA APIs
IMPORT javax.transaction.api
IMPORT javax.persistence.api

RULE resolve TX and JPA classes
CLASS . . .
METHOD . . .
AT ENTRY
IMPORT javax.transaction.api
. . .
ENDRULE

# cancel script level APIs and use hibernate APIs
RULE resolve Hibernate classes
CLASS  . . .
METHOD . . .
AT ENTRY
IMPORT
IMPORT org.hibernate
. . .
ENDRULE

# cancel all script level APIs
IMPORT

RULE resolve only trigger class scope
. . .
ENDRULE
```

In order for module imports to work Byteman has to be able to type check the rule using a class loader which looks up type by name first via the target class's loader and then via the classloaders for any imported modules. Since this is a module system-specific task it requires the use of a module system-specific extension jar.

Byteman supports a plugin architecture to install an extension which will handle `IMPORT` declarations. You need to configure an appropiate module system plugin when you install the Byteman agent if you want to be able to use imports. Details of how to configure this plugin are provided in chapter *Using Byteman* below.

Currently, Byteman ships with only the one JBoss Modules plugin, for use with JBoss EAP and related products that use the JBoss Modules module system. Eventually, Byteman will provide plugins for other module systems such as the most popular OSGi implementations and, perhaps, the JDK's own Jigsaw module system

Byteman Rule Language Standard Built-Ins

The default helper class provides the following standard suite of built-in calls for use in rule expressions. These are primarily intended for use in condition and action expressions but they may also be called in event bindings. 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. Built-in operations divide into three categories, thread coordination operations, rule state management operations and trace and debug operations

Thread Coordination Operations

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 signalWake(Object identifier, boolean mustMeet)
public boolean signalThrow(Object identifier)
public boolean signalThrow(Object identifier, boolean mustMeet)
```

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 its 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 `signalThrow` 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.

`signalWake` takes an optional argument `mustMeet` which is useful in situations where it cannot be guaranteed that the waiting thread will reach its trigger point before the signalling thread arrives at its trigger point. If this argument is supplied as `true` then the signalling thread will not deliver its signal until another thread is waiting. If necessary the signalling thread will suspend until a waiting thread arrives. Supplying value false is equivalent to omitting the optional argument.

`signalThrow` 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.

`signalThrow` also takes an optional argument `mustMeet` which enables the same behaviour as for signalWake.

Rendezvous

Waiters are useful in situations where there is an asymmetrical relationship between threads: one or more threads need to wait for an event which will be signalled by the thread in which the event happens. A rendezvous provides a way of synchronizing where there is no such asymmetry. A rendezvous also provides a way of introducing asymmetry since it sorts threads by order of arrival. The value returned from the rendezvous built-in can be checked to identify, say, the first (or last) thread to arrive and that thread can be the one whose action is triggered.

```
public boolean createRendezvous(Object identifier,
                                 int expected)
public boolean createRendezvous(Object identifier,
                                 int expected,
                                 boolean rejoinable)
public boolean rendezvous(Object identifier)
public boolean rendezvous(Object identifier, long timeout)
public boolean isRendezvous(Object identifier, int expected)
public int getRendezvous(Object identifier, int expected)
public int deleteRendezvous(Object identifier, int expected)
```

`createRendezvous` creates a rendezvous identified by `identifier`. `count` identifies the number of threads which must meet at the rendezvous before any one of them is allowed to continue execution. The optional argument `rejoinable` defaults to false in which case any attempt to meet once the first count threads have arrived will fail. If it is supplied as true then once count threads have arrived the rendezvous will be reset, enabling another round of meetings to occur. `createRendezvous` returns `true` if the rendezvous is created. If a rendezvous identified by identifier already exists it returns false. Note that it is legitimate (although pathological) to supply a count of 1.

`rendezvous` is called to meet other threads at a `rendezvous` identified by `identifier`. If the number of

threads (*including the calling thread*) arrived at the rendezvous is less than the expected count then the calling thread is suspended. If the number of threads equals the expected count then all suspended threads are awoken. A rejoinable rendezvous has its arrived count reset to 0 at this point. If the rendezvous is not rejoinable then it is deleted and any subsequent call to rendezvous using the original identifier will return -1.

`rendezvous` may also be passed a timeout identifying the number of milliseconds which the caller should wait for the all threads to arrive. If the timeout interval is exceeded without the desired number of threads reaching the rendezvous a (runtime) exception will be thrown from the call. A zero or negative timeout value means do not time out.

`isRendezvous` will return true if a rendezvous identified by identifier with the expected count is currently active. If there is no active rendezvous identified by identifier or it exists but has a different expected count then `getRendezvous` will return false.

`getRendezvous` will return the number of threads waiting at the rendezvous identified by `identifier` or 0 if no threads are currently waiting. If there is no rendezvous identified by `identifier` or it exists but has a different expected count then `getRendezvous` will return -1.

`deleteRendezvous` deletes a rendezvous, breaking the association between `identifier` and the rendezvous and forcing any threads waiting under a call to rendezvous to return immediately with result -1. If a rendezvous with the correct expected count is found and successfully deleted it returns true. If there is no such rendezvous or if it is deleted either by another concurrent call to `deleteRendezvous()` or because a concurrent call to rendezvous() completes the rendezvous it returns `false`.

Joiners

Joiners are useful in situations where it is necessary to ensure that a thread does not proceed until one or more related threads have exited. This is not always a requirement for an application to execute correctly but may be necessary to validate a test scenario. For example, a socket listener thread may create connection manager threads to handle incoming connection requests. The listener might use the connection object to notify connection manager threads of a forced exit. It does not necessarily have to retain a handle on the connection thread and explicitly call `Thread.join()` be sure the therad will exit when notified. However, a test may want to check the thread pool to be sure all activity has completed. This means the test needs to be able to accumulate a list of managed threads and then subsequently join them either from the manager thread or from a test thread.

```
public boolean createJoin(Object identifier, int expected)
public boolean isJoin(Object identifier, int expected)
public boolean joinEnlist(Object identifier)
public boolean joinWait(Object identifier, int expected)
public boolean joinWait(Object identifier,
                        int expected,long timeout)
```

`createJoin` creates a Joiner which can subsequently be referenced by identifier. expected identifies the number of threads which are to be joined. If a Joiner is created it returns `true`. If a Joiner is

currently `identified` by identifier it returns false.

`isJoin` tests whether `identifier` identifies a Joiner with the given expected count. If a Joiner with the given expected count is currently identified by `identifier` it returns `true` otherwise it returns `false`.

`joinEnlist` adds the calling thread to the list of threads associated with a Joiner and returns `true`, allowing the thread to proceed towards exit. If `identifier` does not identify a Joiner it returns `false`. It also returns `false` if the calling thread is already contained in the Joiner's thread list or if the number of threads added to the list has reached the `expected` count provided when the Joiner was created.

`joinWait` suspends the calling thread until the number of threads in the list associated with the Joiner reaches the expected count. It then joins each thread in the list and returns `true`. If `identifier` does not identify a Joiner or identifies a Joiner with the wrong `expected` count it returns `false`.

`joinWait` may also be passed a timeout identifying the number of milliseconds which the caller should wait for the thread count to reach the expected count and for the subsequent join operations to complete. If the timeout interval is exceeded without the desired number of threads reaching the expected count a (runtime) exception will be thrown from the call. A zero or negative timeout value means do not time out.

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 the following:

```
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

Rule State Management Operations

LinkMaps

The rule engine provides LinkMaps which can be used to record information available when a rule fires for retrieval later on by other rules or, perhaps, at the end of a test run. A link map is basically a named `Map` which associates one `Object` with another. The API defined by the helper class is

```
boolean createLinkMap(Object mapName)
boolean deleteLinkMap(Object mapName)
Object link(Object mapName, Object name, Object value)
Object linked(Object mapName, Object name)
Object unlink(Object mapName, Object name)
List<Object> linkNames(Object mapName)
List<Object> linkValues(Object mapName)
boolean clearLinks(Object mapName)
Object link(Object name, Object value)
Object linked(Object name)
Object unlink(Object name)
List<Object> linkNames()
List<Object> linkValues()
boolean clearLinks()
```

The API methods which omit a `mapName` parameter operate on the default map which is a predefined map labelled using the global `String` name `"default"`. However, you can have as many maps as you want, labelled using whatever `Object` is handy when a rule fires. When running a multi-threaded program it is often useful to use the current thread to name a LinkMap as this ensures that values saved by one thread do not get overwritten by other threads.

`createLinkMap` can be used to create a LinkMap before using it. However, it is not really necessary to call this since the other API functions will create a map if needed where it does not already exist. It returns `true` if the map does not already exist or `false` otherwise.

`deleteLinkMap` is used to delete a LinkMap. This is useful because it ensures all references to the objects linked in the map are removed. It returns `true` if a map with labelled by `mapName` was found and deleted or `false` otherwise.

`link` is used to add a link from `name` to `value` to a map. The version with no `mapName` argument adds a link to the default map i.e. the one labelled with name `"default"`. The return value from the call is any previous value that was linked to `name` or null if no link was previously present in the map.

`linked` is used to retrieve the value linked via `name` from a map. The version with no `mapName` argument retrieves the value of the link from the default map i.e. the one labelled with name `"default"`. The return value from the call is whatever value is linked to `name` or null if no link is present in the map or if a map labelled `mapname` cannot be found.

`unlink` is used to remove any link from `name` from a map. The version with no `mapName` argument removes any link to the default map i.e. the one labelled with name `"default"`.The return value from the call is whatever value was linked to `name` or null if no link was present in the map.

`linkNames` is used to retrieve a list of all Objects used as names for links in a map. The version with no `mapName` argument retrieves all names for links in the default map i.e. the one labelled with name `"default"`. The return value will be a, possibly empty, list if the map is found when the call is made. It will be null if a map labelled `mapname` cannot be found.

`linkValues` is used to retrieve a list of all Objects occuring as values for links in a map. The version with no `mapName` argument retrieves all values for links in the default map i.e. the one labelled with

name `"default"`. The return value will be a, possibly empty, list if the map is found when the call is made. It will be null if a map labelled `mapname` cannot be found.

`clearLinks` is used to atomically clear all links from a map. The version with no `mapName` argument clears all links in the default map i.e. the one labelled with name `"default"`. The return value will be true if a non-empty map labelled `mapname` is found and cleared or false if no map is found or an empty map is found.

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 createCountDown(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 `createCountDown($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 `createCountDown("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.

`createCountDown` 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. `createCountDown` 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.

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 int incrementCounter(Object o, int amount)
public int incrementCounter(Object o)
public int decrementCounter(Object o)
public int readCounter(Object o)
public int readCounter(Object o, boolean zero)
```

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 `o`. If argument count is not supplied then the value of the new Counter defaults to `o`. `createCounter` returns `true` if a new Counter was created and false if a Counter associated with `o` 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 `o`. It returns true if the Counter was deleted and false if no Counter was associated with `o`. 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 `o`. If no such Counter exists it will create one with value 0 before incrementing it. `incrementCounter` returns the new value of the Counter. If amount is omitted it defaults to 1.

`decrementCounter` is equivalent to calling `incrementCounter(o, -1)` i.e. it adds -1 to the value of the counter.

`readCounter` can be called to read the value of the Counter associated with `o`. If no such Counter exists it will create one with value 0. If the optional flag argument `zero` is passed as `true` the counter is atomically read and zeroed. `zero` defaults to `false`.

Timers

The rule engine provides Timers which allow measurement of elapsed time between triggerings. Timers can be created, read, reset and deleted via the following API

```
public boolean createTimer(Object o)
public long getElapsedTimeFromTimer(Object o)
public long resetTimer(Object o)
public boolean deleteTimer(Object o)
```

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

`createTimer` can be called to create a new Timer associated with `o`. createTimer returns `true` if a new Timer was created and `false` if a Timer associated with `o` already exists.

`getElapsedTimeFromTimer` can be called to obtain the number of elapsed milliseconds since the Timer associated with `o` was created or since the last call to resetTimer. If no timer associated with `o` exists a new timer is created before returning the elapsed time.

`resetTimer` can be called to zero the Timer associated with `o`. It returns the number of seconds since the Timer was created or since the last previous call to `resetTimer` If no timer associated with `o` exists a new timer is created before returning the elapsed time.

`deleteTimer` can be called to delete the Timer associated with `o`. `deleteTimer` returns true if a new Timer was deleted and false if no Timer associated with `o` exists.

Recursive Triggering

When a rule is triggered it executes the Java code in the event, condition and action and this may include calls to Helper methods or methods defined by the application under test or by the JVM runtime. If any of these methods match Byteman rules then this may result in a recursive entry to the rule execution engine. In some cases this may be desirable. However, in other cases this

recursive entry may cause an infinite triggering chain and it is necessary to disable triggering while the rule executes. For example, the following rule will fail because of this problem:

```
RULE infinite triggering chain
CLASS java.io.FileOutputStream
METHOD open(String, int)
AT EXIT
BIND filename = $1
IF TRUE
DO traceln("openlog", "Opened " + $1 + " for write")
ENDRULE
```

The problem is that on the first call to builtin method `traceln(Object, String)` the default helper class attempts to open a trace file which it will then associate with key `"openlog"`. In doing so it calls `FileOutputStream.open` and retriggers the rule.

One way round this is to specify a condition which will break the chain. The trace file will have a name of the form "trace *NNN*.txt" so the following version of the rule works as desired:

```
RULE infinite triggering chain broken using IF test
CLASS java.io.FileOutputStream
METHOD open(String, int)
AT EXIT
BIND filename = $1
IF !filename.matches("trace.*")
DO traceln("openlog", "Opened " + $1 + " for write")
ENDRULE
```

With this version the rule is triggered recursively under the call to `traceln` but the condition stops it being fired, breaking the recursion.

Of course in other cases it may not be so simple to come up with a condition which avoids recursive firing. So, the default helper provides the following method which allows triggering to be disabled or re-enabled while the rule is executing

```
public boolean setTriggering(boolean enabled)
```

If enabled is `false` then triggering is disabled during execution of subsequent expressions in the rule body. If it is `true` then triggering is re-enabled.

This can be used to implement the behaviour shown in the example above without the need to identify a suitable conditional

```
RULE infinite triggering chain broken using IF test
CLASS java.io.FileOutputStream
METHOD open(String, int)
AT EXIT
BIND filename = $1
IF TRUE
DO setTriggering(false);
 traceln("openlog", "Opened " + $1 + " for write")
ENDRULE
```

Note that once execution of the rule has completed triggering is automatically re-enabled so, in this case, there is no need to call `setTriggering(true)` at the end of the `DO` clause.

Method `setTriggering` always returns boolean value `true`, allowing it to be ANDed into the condition of an `IF` clause or used to initialise a rule variable declared in a `BIND` clause. This is sometimes necessary to ensure that triggering is disabled early, before other expressions in the `IF` or `BIND` clause are evaluated.

Trace and Debug Operations

Debugging

The rule engine provides a simple built-in dbug method to support conditional display of messages during rule execution. 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 `AND` ing them with other boolean expressions.

Generation of debug messages can be switched on by setting the following system property on the JVM command line:

```
org.jboss.byteman.debug
```

Tracing

The rule engine provides a set of built-in methods to support logging of trace messages during execution. Messages may be logged to `System.out`, `Sytem.err` or to a named file. The API defined by the helper class is the following:

```
public boolean traceOpen(Object identifier, String filename)
public boolean traceOpen(Object identifier)
public boolean traceClose(Object identifier)
public boolean trace(Object identifier, String message)
public boolean traceln(Object identifier, String message)
public boolean trace(String message)
public boolean traceln(String message)
```

`traceOpen` opens the file identified by `fileName` and associates it with identifier, returning `true`. `filename` can be either a relative or absolute path. Relative file names are located relative to the current working directory of the JVM. If there is already a file associated with identifier then `traceOpen` immediately returns `false`. If a file with the given name already exists it is opened in append mode. If filename is omitted then a unique name is generated for the file which is guaranteed not to match any existing trace file in the current working directory.

`traceClose` closes the file associated with identifier and removes the association, returning `true`. If no open file is associated with identifier it returns false.

`trace` prints message to file associated with identifier, returning `true`. If no open file is associated with identifier then a file will be opened and associated with identifier as if a call to trace had been made with no file name supplied. If identifier is omitted then the output is written to `System.out`.

`traceln` prints message to file associated with identifier and appends a newline to the file, returning true. If no open file is associated with identifier then a file will be opened and associated with identifier as if a call to trace had been made with no file name supplied. If identifier is omitted then the output is written to `System.out`.

A caveat applies to the above descriptions for three special cases. If identifier is `null` or the string "out", then `trace` and `traceln` write to `System.out`. If identifier is the string "err", then `trace` and `traceln` write to `System.err`. `traceOpen` and `traceClose` always return `false` immediately if identifier has any of these values. Calls to `trace(message)` and `traceln(message)` which omit identifier are implemented by calling, respectively, `trace("out", message)` and `traceln("out", message)`.

Stack Management Operations

Checking The Call Tree

The rule engine provides a set built-in methods which can be used to check the caller stack at the point where the rule was triggered. Obviously, the rule will only be triggered from a method which matches the name in its `METHOD` clause. However, sometimes it is useful to be able to know which method called the trigger rule. For example, the following rule will only fire when method `MyClass.getData()` is called from method `handleIncoming` of class `MyOtherClass`:

```
RULE trace getData call under handleIncoming
CLASS MyClass
METHOD myGetData
IF callerEquals("MyOtherClass.handleIncoming", true)
DO traceStack("found the caller!\n", 10)
ENDRULE
```

The API defined by the helper class is

```
public boolean callerEquals(String name)
public boolean callerEquals(String name,
                            int frameCount)
public boolean callerEquals(String name,
                            int startFrame,
                            int frameCount)
public boolean callerEquals(String name,
                            boolean includeClass)
public boolean callerEquals(String name,
                            boolean includeClass,
                            int frameCount)
public boolean callerEquals(String name,
                            boolean includeClass,
                            int startFrame,
                            int frameCount)
public boolean callerEquals(String name,
                            boolean includeClass,
                            boolean includePackage)
public boolean callerEquals(String name,
                            boolean includeClass,
                            boolean includePackage,
                            int frameCount)
public boolean callerEquals(String name,
                            boolean includeClass,
                            boolean includePackage,
                            int startFrame,
                            int frameCount)

public boolean callerMatches(String regExp)
public boolean callerMatches(String regExp,
                             int frameCount)
public boolean callerMatches(String regExp,
                             int startFrame,
                             int frameCount)
public boolean callerMatches(String regExp,
                             boolean includeClass)
public boolean callerMatches(String regExp,
                             boolean includeClass,
                             int frameCount)
public boolean callerMatches(String regExp,
```

```
                              boolean includeClass,
                              int startFrame,
                              int frameCount)
public boolean callerMatches(String regExp,
                              boolean includeClass,
                              boolean includePackage)
public boolean callerMatches(String regExp,
                              boolean includeClass,
                              boolean includePackage,
                              int frameCount)
public boolean callerMatches(String regExp,
                              boolean includeClass,
                              int startFrame,
                              int frameCount)

public boolean callerCheck(String match, boolean isRegExp,
                           boolean includeClass,
                           boolean includePackage,
                           int startFrame,
                           int frameCount)
```

The real action happens in method `callerCheck(String, boolean, boolean, boolean, int, int)`. All the other methods call each other defaulting the various missing arguments until they bottom out in a call to this method.

`callerCheck` tests `frameCount` call frames starting from `startFrame` and returns `true` if any of them matches match.

`startFrame` defaults to 1 which identifies the stack frame for the caller of the trigger method (0 can be used to identify the trigger method itself). `framecount` also defaults to 1 which means that when `startFrame` and `frameCount` are defaulted the call only checks the frame for the caller of the trigger method.

`includeClass` and `includePackage` default to `false`. If `includeClass` is false then match is compared against the bare name of the method associated with each selected stack frame. If `includeClass` is true and `includePackage` is false then match is compared to the class qualified method name. If both are `true` then match is compared against the full package and class qualified method name.

If `isRegExp` is true then match is compared as a regular expression compared using `String.matches()` otherwise it compared using `String.equals()`. The `callerEquals` methods pass this argument to `callerCheck` as false and the `callerMatches` methods pass this argument as true.

Tracing the Caller Stack

The rule engine provides a set built-in methods which can be used to obtain a string representation of a stack trace or to print a stack trace to a trace file. The API defined by the helper class is

```
public void traceStack()
public void traceStack(String prefix)
public void traceStack(String prefix, Object key)
public void traceStack(int maxFrames)
public void traceStack(String prefix, int maxFrames)
public void traceStack(String prefix,
                       Object key,
                       int maxFrames)

public String formatStack()
public String formatStack(String prefix)
public String formatStack(int maxFrames)
public String formatStack(String prefix, int maxFrames)
```

The real action happens in methods `traceStack(String, Object, int)` and `formatStack(String, int)`. All the other methods call each other defaulting the various missing arguments until they bottom out in a call to one of these two methods.

`formatStack(String prefix, int maxFrames)` constructs a printable String representation of the stack starting from the trigger frame, including the fully qualified method name, file and line number for each frame followed by a new line.

If `prefix` is non-null it prepended to the generated text. It defaults to `null` resulting in the prefix "Stack trace for thread <*current*>\n" being used as the prefix where <*current*> is substituted with the value of `Thread.currentThread().getName()`.

If `maxFrames` is positive and less than the number of frames in the stack then it is used to limit the number of frames printed and the text "...\n" is appended to the returned value. Otherwise all frames in the stack are included. `maxFrames` defaults to 0.

`traceStack(String prefix, Object key, int maxFrames)` constructs a stack trace by calling `formatStack(key, maxFrames)`. It then prints this to a trace file by calling `trace(key, <value>)`. As before, `prefix` defaults to `null` and `maxFrames` to 0. key defaults to "out" so this means that where it is omitted the trace printout will go to `System.out`.

Selective Stack Tracing Using a Regular Expression Filter

It is useful to be able to selectively filter a stack trace, limiting it, say, to include only frames from a given package or set of packages. The rule engine provides an alternative set of built-in methods which can be used to obtain or print a string representation of some subset of the stack filtered using a regular expression match. The API defined by the helper class is

```
public void traceStackMatching(String regExp)
public void traceStackMatching(String regExp, String prefix)
public void traceStackMatching(String regExp,
                               String prefix,
                               Object key)
public void traceStackMatching(String regExp,
                               boolean includeClass)
public void traceStackMatching(String regExp,
                               boolean includeClass,
                               String prefix)
public void traceStackMatching(String regExp,
                               boolean includeClass,
                               String prefix,
                               Object key)
public void traceStackMatching(String regExp,
                               boolean includeClass,
                               boolean includePackage)
public void traceStackMatching(String regExp,
                               boolean includeClass,
                               boolean includePackage,
                               String prefix)
public void traceStackMatching(String regExp,
                               boolean includeClass,
                               boolean includePackage,
                               String prefix,
                               Object key)

public void formatStackMatching(String regExp)
public void formatStackMatching(String regExp, String prefix)
public void formatStackMatching(String regExp,
                                boolean includeClass)
public void formatStackMatching(String regExp,
                                boolean includeClass,
                                String prefix)
public void formatStackMatching(String regExp,
                                boolean includeClass,
                                boolean includePackage)
public void formatStackMatching(String regExp,
                                boolean includeClass,
                                boolean includePackage,
                                String prefix)
```

Once again the action happens in the methods with the full set of parameters and the others merely call these methods defaulting the omitted arguments.

`formatStackMatching(String regExp, boolean includeClass, boolean includePackage, String prefix)` constructs a printable String representation of the stack prefixed by prefix as per `formatStack` with the difference that frames are only included if they match the regular expression `regExp`. `includeClass` and `includePackage` are defaulted and interpreted exactly as described in the

`callerMatches` API. If `prefix` is `null` (the default) then the string "Stack trace for thread <*current*> matching regExp\n" is used as the prefix where <*current*> is substituted with the value of `Thread.currentThread().getName()` and `regExp` is substituted with the value of `regExp`.

`traceStackMatching(regExp, includeClass, includePackage, prefix, key)` calls `formatStackMatching` to obtain a stack trace and then calls `trace(String, Object)` to print it to the trace stream identified by `key`. `key` defaults as described in the `traceStack` API listed above.

Stack Range Tracing

Another option for selective stack tracing is to specify a matching expression to select the start and end frame for the trace. The rule engine provides another set of built-in methods which can be used to obtain or print a string representation of a segment of the stack in this manner. The API defined by the helper class is

```
public void traceStackBetween(String from, String to)
public void traceStackBetween(String from, String to,
                              String prefix)
public void traceStackBetween(String from, String to,
                              String prefix, Object key)
public void traceStackBetween(String from, String to,
                              boolean includeClass)
public void traceStackBetween(String from, String to,
                              boolean includeClass,
                              String prefix)
public void traceStackBetween(String from, String to,
                              boolean includeClass,
                              String prefix, Object key)
public void traceStackBetween(String from, String to,
                              boolean includeClass,
                              boolean includePackage)
public void traceStackBetween(String from, String to,
                              boolean includeClass,
                              boolean includePackage,
                              String prefix)
public void traceStackBetween(String from, String to,
                              boolean includeClass,
                              boolean includePackage,
                              String prefix, Object key)

public void formatStackBetween(String from, String to)
public void formatStackBetween(String from, String to,
                               String prefix)
public void formatStackBetween(String from, String to,
                               boolean includeClass)
public void formatStackBetween(String from, String to,
                               boolean includeClass,
                               String prefix)
public void formatStackBetween(String from, String to,
                               boolean includeClass,
```

```
                                boolean includePackage)
public void formatStackBetween(String from, String to,
                                boolean includeClass,
                                boolean includePackage,
                                String prefix)

public void traceStackBetweenMatches(String from, String to)
public void traceStackBetweenMatches(String from, String to,
                                     String prefix)
public void traceStackBetweenMatches(String from,String to,
                                     String prefix,
                                     Object key)
public void traceStackBetweenMatches(String from, String to,
                                     boolean includeClass)
public void traceStackBetweenMatches(String from, String to,
                                     boolean includeClass,
                                     String prefix)
public void traceStackBetweenMatches(String from, String to,
                                     boolean includeClass,
                                     String prefix,
                                     Object key)
public void traceStackBetweenMatches(String from, String to,
                                     boolean includeClass,
                                     boolean includePackage)
public void traceStackBetweenMatches(String from, String to,
                                     boolean includeClass,
                                     boolean includePackage,
                                     String prefix)
public void traceStackBetweenMatches(String from, String to,
                                     boolean includeClass,
                                     boolean includePackage,
                                     String prefix,
                                     Object key)

public void formatStackBetweenMatches(String from, String to)
public void formatStackBetweenMatches(String from, String to,
                                      String prefix)
public void formatStackBetweenMatches(String from, String to,
                                      boolean includeClass)
public void formatStackBetweenMatches(String from, String to,
                                      boolean includeClass,
                                      String prefix)
public void formatStackBetweenMatches(String from, String to,
                                      boolean includeClass,
                                      boolean includePackage)
public void formatStackBetweenMatches(String from, String to,
                                      boolean includeClass,
                                      boolean includePackage,
                                      String prefix)

public void traceStackRange(String from, String to,
```

```
                              boolean isRegExp,
                              boolean includeClass,
                              boolean includePackage,
                              String prefix, Object key)
public String formatStackRange(String from, String to,
                              boolean isRegExp,
                              boolean includeClass,
                              boolean includePackage,
                              String prefix)
```

Once again the action happens in the last two methods and all the other methods merely provide a way of calling them with default values for various of the parameters. The `BetweenMatches` methods pass `true` for parameter `isRegExp` whereas the plain Matches methods pass `false`.

`formatStackRange` searches the stack starting from the trigger frame for a stack frame which matches `from`. If no match is found then "" is returned. If `from` is `null` then the trigger frame is taken to be the start frame. It then searches the frames above the start frame for a frame which matches to. If no match is found or if to is null then all frames above the start frame are selected. Details of each frame in the matching range are appended to the supplied prefix to construct the return value. If `isRegExp` is `true` then the start and end frame are matched using `String.matches()` otherwise `String.equals()` is used. `includeClass` and `includePackage` are defaulted and interpreted as per method `formatStackMatching`. If `prefix` is `null` (the default) then the string “Stack trace (restricted) for thread <*current*>\n” is used as the prefix where <*current*> is substituted with the value of `Thread.currentThread().getName()`.

`traceStackRange` calls `formatStackRange` to obtain a trace of a stack range and then calls `trace(Object, String)` to print it to a trace file. `key` defaults to "out" as for the other stack trace APIs described above.

Tracing Named Thread Stacks

The default helper class also provides methods which can be used to trace or format the stack frames of a specific, named thread:

```
public void traceThreadStack(String threadName)
public void traceThreadStack(String threadName,
                             String prefix)
public void traceThreadStack(String threadName,
                             String prefix,
                             Object key)
public void traceThreadStack(String threadName,
                             int maxFrames)
public void traceThreadStack(String threadName,
                             String prefix,
                             int maxFrames)
public void traceThreadStack(String threadName,
                             String prefix,
                             Object key,
                             int maxFrames)

public void formatThreadStack(String threadName)
public void formatThreadStack(String threadName,
                             String prefix)

public void traceThreadStack(String threadName,
                             int maxFrames)
public void traceThreadStack(String threadName,
                             String prefix,
                             int maxFrames)
```

or to trace or format the stacks of all threads in the runtime:

```
public void traceAllStacks()
public void traceAllStacks(String prefix)
public void traceAllStacks(String prefix, Object key)
public void traceAllStacks(int maxFrames)
public void traceAllStacks(String prefix, int maxFrames)
public void traceAllStacks(String prefix, Object key, int maxFrames)
public void formatAllStacks()
public void formatAllStacks(String prefix)
public void formatAllStacks(int maxFrames)
public void formatAllStacks(String prefix , int maxFrames)
```

Default Helper Lifecycle Methods

The default helper provides an implementation of the four helper lifecycle methods which generate simple debug messages to `System.out`. So, with debug enabled you will see messages like the following as rules are loaded and then unloaded:

```
Default helper activated
Installed rule using default helper : my test rule
. . .
Installed rule using default helper : my second test rule
. . .
Uninstalled rule using default helper : my test rule
Uninstalled rule using default helper : my second test rule
Default helper deactivated
```

Using Byteman

Using Byteman from java or ant

If you are using Byteman from the java command line or from ant then you will need to download a Byteman release and install it locally. The latest Byteman release is available as a zip file from the Byteman project downloads page at

- http://www.jboss.org/byteman/downloads

You need to download either the binary release or the full release and install it in a directory accessible on the machine on which you want to run Byteman. The rest of this document assumes that environment variable `BYTEMAN_HOME` identifies the directory into which the binary release is unzipped.

The binary release contains all the binary (classfile) jars and command scripts you will need to use Byteman, a copy of the programmers guide plus some sample scripts and an associated helper jar. The full release also provides source and javadoc jars.

Details of how to use Byteman from the command line are included in the Byteman Command Line Tutorial.

If you are using Byteman from ant you need to use the BMUnit package which integrates Byteman with JUnit and TestNG. Details of how to configure an ant build script to reference the required jars from your Byteman download are provided in the Byteman BMUnit Tutorial. A more complex example showing the possibilities offered by BMUnit is provided in the Byteman Fault Injection Tutorial.

The downloads page also contains a link to downloads for older binary release versions. Note that from Byteman 1.1 onwards the agent can only be run in JDK 6 or 7. Previous releases can also be run in JDK 5.

Using Byteman from maven

If you are using Byteman from maven, in particular, if you are using the BMUnit package which integrates Byteman with JUnit and TestNG, then you normally only need to declare dependencies on the Byteman jars. maven will download these jars automaticaly from the Maven Central repository.

Details of how to configure your pom to depend on the required jars are provided in the Byteman BMUnit Tutorial. A more complex example showing the possibilities offered by BMUnit is provided in the Byteman Fault Injection Tutorial.

Byteman also provides a maven plugin which can be used to parse and type-check your test rule scripts as part of the maven test cycle. Details of how to configure the maven plugin are provided in the Byteman Rulecheck Plugin Tutorial.

Obtaining the source build tree

If you want to understand how Byteman works and, possibly, contribute to the Byteman project you will need to download the latest Byteman source tree. The sources are available from the master git repository at

- https://github.com/bytemanproject/byteman

The source tree is structured as a maven project..

Building Byteman from the sources

Byteman can be built by executing command mvn package in the top level directory of the source tree. The build process should produce binary, source and javadoc jars in the `target` directory of each of the dependent sub-modules (`agent/target`, `submit/target`, etc.) and assemble these into both a binary and a full release zip file in subdirectory `download/target`.

If you are using Byteman from ant or the java command line then you should unzip one of the download zip files into your local `BYTEMAN_HOME` directory as you would for any of the official downloads.

If you want to use Byteman from maven then you need to install the byteman jars in your local machine's maven repository by executing command mvn install. In this case before performing the install process you should modify the project version in the root pom and the parent version in each of the dependent sub-module poms so that the version you install locally does not overwrite a version obtained form the JBoss master repository.

Running Applications with Byteman

The Byteman tutorials explain how to use Byteman to trace and test the behaviour of a variety of programs ranging from simple examples to sophisticated test cases.

- The Byteman Command Line Tutorial covers the basics of how to run Byteman from the command line.
- The Byteman BMUnit Tutorial explains how to integrate Byteman into your JUnit or TestNG tests drievn either from ant or maven.
- The Byteman Fault Injection Tutorial provides an example of how to use Byteman to do more sophisticated fault injection testing.

The tutorials are probably the best thing to look at if you are new to Byteman. The rest of this guide

provides complete details of how to use Byteman from the Java command line and explains the effect of all the configuration options which are available when using Byteman. Many of these options can also be configured when using BMUnit from ant or maven.

Configuring a java agent

Using Byteman from the java command line is refreshingly simple. Installing Byteman at JVM startup requires only one extra argument. This argument points the JVM at the agent code in the byteman jar and at the script files containing your byteman rules. This is specified using the java command flag

```
-javaagent:agentlib=options
```

This is a standard option for JDK 1.6 and upwards.

- *agentlib* is a path to the byteman jar. The build process inserts a metadata file in the jar which allows the JVM to identify the agent program entry point so everything else is shrink-wrapped.
- *options* is a comma-separated sequence of options which are passed to the agent.

For example, setting

```
export JAVA_OPTS="-javaagent:$BYTEMAN_HOME/lib/byteman.jar=
script:$SCRIPT_HOME/HeuristicSaveAndRecover.btm,
script:$SCRIPT_HOME/JBossTSTrace.btm"
```

will cause the JVM to pick up the byteman jar file from the install root, `BYTEMAN_HOME` and inject all rules found in the two scripts `HeuristicSaveAndRecover.btm` and `JBossTSTrace.btm` located in directory `SCRIPT_HOME`. (n.b. the command should actually be all on one line without any line breaks – these have been inserted merely as an aid to readability)

Normally, the script option is all that is required . However there are several other options provided to configure more sophisticated features of the agent such as dynamic rule upload. The `bmjava` command script (see below) can be used to simplify the task of concatenating these options into a valid `-javaagent` argument.

Installing Byteman in a Running JVM using Script bminstall

If your Java application is already running you may still be able to install the Byteman agent into the JVM. The installed `bin` directory contains a script `bminstall` [1: There is a Windows script called `bminstall.bat` which you can execute using the command `bminstall`. The Linux script is called `bminstall.sh` and you must supply this full name to execute it.] which can be used to contact your application's JVM and install the agent. Once installed you can then use script `bmsubmit` described below to upload or unload rules via the Byteman agent listener. The command line syntax for the `bminstall` script is

```
bminstall [-p port] [-h host] [-b] [-s] [-m] [-Dproperty[=value]]* pid
```

where terms between *[* and *]* are optional and * means zero or more repetitions.

- `pid` is the process id of the JVM running your application or, alternatively, the name of the application as displayed by running command `jps -l`.
- `-h` host means use host as the host name when opening the listener socket (the default is localhost)
- `-p` port means use port as the port when opening the listener socket (the default is 9091)
- `-b` means add the byteman jar to the bootstrap class path
- `-s` sets an access-all-areas security policy for the Byteman agent code. This may be necessary when the JVM is running an application employing a security manager which imposes access restrictions (this includes recent versions of JBoss Wildfly/EAP).
- `-m` activates the byteman JBoss modules plugin
- `-Dproperty=value` means call `System.setProperty(property, value)` before starting the agent. More than one such property setting may be provided. If value is omitted then it defaults to "". Note that property must start with prefix "`org.jboss.byteman.`".

bminstall expects to find the byteman agent jar (`byteman.jar`) in `$BYTEMAN_HOME/lib`. If `-m` is specified it also expects to find the byteman JBoss Modules plugin jar (`byteman-jboss-modules-plugin.jar`) in `$BYTEMAN_HOME/contrib/jboss-modules-system`.

Note that when you identify the JVM using an application name instead of a process id you do not have to provide the full name but you must be sure the name is unambiguous. For example, if you want to install the agent into the JBoss Application Server you can use the full name `org.jboss.Main` or the abbreviated names `jboss.Main` or `Main`. Clearly, `Main` will often be a bad choice as many applications will use that name. The install will be done using the first matching JVM.

Note also that loading the agent at runtime may not work with certain JVMs. It will definitely not work unless your Java release includes a jar named lib/tools.jar below your top-level Java install directory and this jar must contain a class called `com.sun.tools.attach.VirtualMachine`. However, even with this jar in place the upload may fail.

Available -javaagent Options

Each option comprises a keyword prefix, identifying the type of the option, and a suffix, identifying the value of the option. Prefix and suffix are separated by a colon, :. Multiple options are separated by commas. Valid options include

script:scriptfile where *scriptfile* is a path to a file containing Byteman rules. This option causes the agent to read the rules in *scriptFile* and apply them to subsequently loaded classes. Multiple script arguments may be provided to ensure that more than one rule set is installed. It is possible to start the agent with no initial script arguments but this only makes sense if the listener option is supplied with value `true`.

resourcescript:scriptfile where *scriptfile* is a path to a resource file on the CLASSPATH containing Byteman rules. This option is like the script option except that the file containing the Byteman rules is located as a class loader resource (the *scriptfile* argument is passed directly to `ClassLoader.getSystemResourceAsStream()`).

listener:boolean where *boolean* is either `true` or `false`. When set to `true` this option causes the agent to start a listener thread at startup. The listener can be talked to using the `bmsubmit script, either to provide listings of rule applications performed by the agent or to dynamically load, reload or unload rules. Loading or reloading of a rule causes any matching classes which have already been loaded into the JVM to be retransformed, inserting a trigger call for the newly loaded rules into their target methods. Unloading a rule causes trigger code to be removed from any matching classes.

n.b. this option is actually an alias for option manager (see below)

port:portnum where *portnum* is a positive whole number. This option selects the port used by the agent listener when opening a server socket to listen on. If not supplied the port defaults to 9091. Supplying this option defaults the listener option to `true`.

address:host where *host* is a host name. This option selects the address used by the agent listener when opening a server socket to listen on. If not supplied the address defaults to 9091. Supplying this option defaults the listener option to `true`.

manager:classname where *classname* is the full, package qualified name of a class which is to manage loading and unloading of rules. The manager option allows you to provide your own plugin class to manage the Byteman agent's installed rule base. The plugin class implements an initialize method with one of the following signatures:

```
void initialize(org.jboss.byteman.agent.Retransfromer, String, int)

void initialize(org.jboss.byteman.agent.Retransfromer)
```

The first argument is the retransformer instance which regulates operation of the Byteman agent's class file transformer. It provides methods for installing and deinstalling rules and for querying the installed rule base. The second and third arguments allow a hostname and port to be passed to the manager class.

An instance of the named manager class is created and its initialize method called when the agent is first loaded. The first variant of initialize is called by preference (passing `null` or -1 if, respectively, no hostname or port was provided as a Byteman agent option). Alternatively, the second variant is invoked. At least one implementation of initialize must be provided.

Option **listener:true** is an alias for **manager:org.jboss.byteman.agent.TransformListener**, e.g. it requests the agent to use the standard Byteman manager class to manage the agent's rule base. `TransformListener` simply opens a socket listener and passes requests from the `bmsubmit` script on to the `Retransformer` then posts results back via the socket. If you wish to implement your own manager then you should study class `TransformListener` in order to understand how the `Retransformer` provides access to the agent rule base.

NOTE	The `-javaagent modules:` option is an early-preview Byteman feature which is incomplete and may be subject to change

modules:classname where *classname* is the full, package qualified name of a class which is to manage module imports. This option allows you to configure a plugin class to handle IMPORT statements and manage the associated class resolution in a modular runtime.

Byteman ships a single plugin implementation for JBoss Modules, which is deployed as `byteman-jboss-modules-plugin.jar`. This is available in Byteman zip downloads under directory `contrib/jboss-modules-system`. It is also available from the Maven Central repo with coordinates: group `org.jboss.byteman`, artifact `byteman-jboss-modules-plugin` and version `3.0.4` or above. The manager class name for the JBoss Modules plugin is `org.jboss.byteman.modules.jbossmodules.JBossModulesSystem`.

Note that when you configure this option you also need to ensure the jar containing the module plugin manager class is added to the system classpath of the JVM into which the agent is being installed (you can use the `sys:/path/to/plugin.jar` option for this purpose). It is not always possible to add this jar to the bootstrap classpath using the `boot:/plugin.jar` option because the module plugin manager needs to reference module system-specific code which will not itself be located in the bootstrap classpath. As a corollary this means it is not normally possible to inject into bootstrap classes when using a modules plugin.

Note that the `bminstall` script provides a command flag `-m` to automatically configure the JBoss Modules plugin when installing the Byteman agent dynamically. It expects to find the JBoss Modules plugin jar in directory `$BYTEMAN_HOME/contrib/jboss-modules-system`.

sys:jarfile where *jarfile* is a path to to a jar file to be added to the JVM *system* class path. This option makes classes contained in the jar file available for use when type checking, compiling and executing rule conditions and actions. It provides a useful way to ensure that Helper classes mentioned in rules are able to be resolved. If a rule's trigger class is loaded by some other class loader this loader will normally have the system loader as a parent so references to the Helper class should resolve correctly.

boot:jarfile where *jarfile* is a path to to a jar file to be added to the JVM *bootstrap* class path. This option provides a similar facility to the sys option but it ensures that the classes contained in the jar file are loaded by the bootstrap class loader. This is only significant when rules try to inject code into JVM classes which are loaded by the bootstrap class loader (which is a parent of the system loader).

policy:boolean where *boolean* is either `true` or `false`. When set to `true` this option causes the agent to install an access all areas security policy for the Byteman agent code. This may be necessary when the JVM is running an application employing a security manager which imposes access restrictions (this includes recent versions of JBoss Wildfly/EAP).

prop:name=value where *name* identifies a System property to be set to value or to the empty String if no value is provided. Note that property names must begin with the prefix "`org.jboss.byteman'.".

Note that when injecting into JVM classes it is necessary to install the byteman jar into the boot

classpath by passing option `boot:${BYTEMAN_HOME}/lib/byteman.jar`. Without it compilation of the transformed class will fail because it cannot locate classes `Rule`, `ExecuteException`, `ThrowException` and `ReturnException`. Script `bmjava` automatically takes care of this requirement.

Running Byteman Using Script bmjava

The installed `bin` directory contains a script `bmjava` which can be used to assemble the options passed to the byteman agent and combine them with other java options supplied on the java command line. The command line syntax for this script is

```
bmjava [-p port] [-h host] [ -l script|-b jar|-s jar|-nb|-nl|-nj ]* [--] javaargs
```

where terms between [ ] are optional, terms separated by | are alternatives, * means zero or more repetitions and

- `-l` *script* means load the rules in file *script* during program start up
- `-b` *jar* means add jar file *jar* to the bootstrap class path
- `-s` *jar* means add jar file *jar* to the system class path
- `-p` *port* means use *port* as the port when opening the listener socket (the default is 9091)
- `-h` *host* means use *host* as the host name when opening the listener socket (the default is `localhost`)
- `-nb` means don't add the byteman jar to the bootstrap class path (it is added by default)
- `-nl` means don't start the agent listener (it is started by default)
- `-nj` means don't inject into java.lang classes (this is allowed by default)

Submitting Rules Dynamically Using Script bmsubmit

The installed `bin` directory contains a script called `bmsubmit` which can be used to communicate with the agent listener started up when option `listener:true` is passed as an option to the Byteman agent (recall that this option is always enabled by the `bminstall` and `bmjava` scripts).

This script can be used to upload rules into the running program, to unload previously loaded rules and to report which rules have been injected and/or whether any errors were detected when attempting injection. `bmsubmit` can also be used to install jars containing helper classes into the bootstrap or system classpath.

The command line syntax for the `bmsubmit` script is:

```
submit [-p port] [-h host] [-l|-u] [script1 . . . scriptN]
submit [-p port] [-h host] [-b|-s] jarfile1 . . .
submit [-p port] [-h host] -c
submit [-p port] [-h host] -y [prop1[=[value1]]. . .]
submit [-p port] [-h host] -v
```

Flags `-p` and `-h` can be used to supply the port and host address used to connect to the Byteman agent listener. If not supplied they default to 9091 and localhost, respectively.

Flag `-l` selects the default execution mode for `bmsubmit`. If no other flag is supplied this mode will be used. When run with no arguments submit lists all currently applied transformations. This listing includes details of failed transformations, typechecks and compiles.

When run with a list of script files `bmsubmit` uploads the scripts to the agent:

1. If any of the uploaded rules have the same name as an existing rule then the old rule is replaced with the new rule and all classes transformed by the rule are automatically retransformed. This includes the case where a target class was transformed but a subsequent typecheck or compile of the rule failed. This will (eventually) cause all threads executing target methods to trigger the new rule instead of the old rule. Depending upon when the compiler installs the new method code and when executing threads switch to this new code there may be a small window following the upload where neither the old nor the new rule is triggered.
2. If any of the uploaded rules apply to already loaded classes which were not previously successfully transformed (because no rule applied to them or because attempts to transform them failed) then those classes will be retransformed. Once again, depending upon when the compiler installs the new method code and when executing threads switch to this new code there may be a small window following the upload where neither the old nor the new rule is triggered.
3. Any other rules will be stored in the agents rule set and applied to newly loaded classes which match the rule.

Flag `-u` selects the uninstall mode for `bmsubmit`. When run with no arguments `bmsubmit` uninstalls all currently loaded rules. When run with a list of script files `bmsubmit` uninstalls each installed rule whose name matches a rule definition contained in one of the supplied script files. `bmsubmit` does not check that rules in the supplied scripts are well formed, it merely looks for lines starting with the text `RULE`.

Flags `-b` and `-s` request installation of jar files into the bootstrap or system classpath, respectively. This is useful if rules which are to be submitted dynamically need to be provided with access to helper classes which were not in the original class path. There is no undo operation for this mode; jar files cannot be uninstalled once they have been installed. Also, it is not possible to use this option to install the byteman jar into the bootstrap classpath if it was omitted at agent startup. By the time the listener responds to this request the system class loader will already have loaded classes from the byteman jar so adding the jar to the bootstrap classpath will result in classloader conflicts.

Flag `-c` can be used to list all helper jars which have been installed by the agent into the bootstrap or system classpath.

Flag `-y` can be used to list or dynamically update the system properties which configure the operation of the Byteman agent. If no following arguments are supplied `bmsubmit` will print the value of all system properties in the agent JVM with prefix "`org.joss.byteman.`". When arguments are supplied `bmsubmit` will set or clear the relevant system properties and notify the agent to update its configuration. If a bare property name is supplied then `bmsubmit` will unset the property. If the property name is followed by = and a value then the system property will be set to this value (=

with no value sets it to an empty string). Note that for performance reasons the agent does not, by default, respond to configuration updates (this allows it to test configuration settings during operation without incurring any synchronization overhead). Dynamic configuration update must be enabled when the agent is started by setting system property `org.jboss.byteman.allow.config.updates` on the JVM command line (to any value).

Checking Rules Offline Using Script bmcheck

The installed `bin` directory contains a script called `bmcheck` which should be used to parse and typecheck your Byteman rules offline before attempting to inject them into your application. This script uses environment variable `BYTEMAN_HOME` to locate the byteman jars needed to type check the rules. However, it still needs to be supplied with a classpath locating any application classes mentioned in the rules.

The command line syntax for the `bmcheck` script is:

```
bmcheck [-cp classpath] [-p package]* script1 [. . . scriptN]
```

Flag `-cp` provides a single classpath element used to locate classes mentioned in the scripts. If your code is located in several jars then you must supply `-cp` multiple times.

Flag `-p` provides one or more package names which are used to resolve target classes which have been specified on the `CLASS` line without a qualifying package name. For example, if a rule specifies `CLASS myClass` and the class actually resides in package `org.myApp` then the rule will be applied when class `org.myApp.myClass` is loaded during application execution. However, the type checker cannot know where to find and load the relevant class without a hint. If `-p org.myApp` is provided on the command line then after failing to locate myClass in the empty package the type checker will try to lookup `myClass` in package `org.myApp`. Each package supplied on the command line is tried in order until the class name can be resolved.

Installing And Submitting from Java

The scripts `bminstall` and `bmsubmit` used, respectively, to install the agent into a running program and to upload and unload scripts are merely wrappers which invoke the behaviour of Java classes, `org.jboss.byteman.agent.install.Install` and `org.jboss.byteman.agent.submit.Submit`, located,respectively, in the `byteman-install` and `byteman-submit` jars in the installed `lib` directory. The behaviour provided by these classes may be invoked from any Java program in order to load the agent or rules into the current JVM or into a remote JVM. The contributed packages provided with the Byteman release provide interesting examples of how to use this powerful capability.

Package `BMUnit` extends the JUnit and TestNG test frameworks so that they automatically install an agent and loads and unloads rules into/from the JUnit or TestNg test JVM as successive unit tests are executed. This makes it trivially easy to inject side effects such as tracing, validation code and faults into your application before running a test and then remove these side effects ready for the next test. Details of how to use BMUnit are provided in the 2nd of the Byteman tutorials which can be accessed from the Byteman Documentation page of the Byteman project web site located at jboss.org.

Package DTest allows a standalone test client to load and unload rules into/from a server JVM which allow the client to inject faults into the server code then record and validate execution of the server. Consult the Javadoc of these two API classes and the contributed package README files and source code for full details.

Environment Settings

The agent is sensitive to various environment settings which configure its behaviour.

org.jboss.byteman.compileToBytecode
: When this system property is set (with any value), then the rule execution engine will compile rules to bytecode before executing them. If this property is unset it will execute rules by interpreting the rule parse tree.

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

org.jboss.byteman.dump.generated.classes
: When this system property is set, 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.

org.jboss.byteman.dump.generated.classes.directory
: When this system property 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.

org.jboss.byteman.dump.generated.classes.intermediate
: When this system property is set, the agent will dump successive versions of transformed bytecode as successive rules applicable to a given class are injected into the bytecode.

org.jboss.byteman.verbose, org.jboss.byteman.debug
: When either of these system properties is set, then the rule execution engine will display a variety of trace messages to the System.out as it parses, typechecks, compiles and executes rules.

  Note that the debug built-in is sensitive to this system property as well as to its own configuration switch

  If either of these properties is set then debug calls will print to System.out.

org.jboss.byteman.transform.all
: When this system property is set, then the agent will allow rules to be injected into methods of classes in the java.lang hierarchy. Note that this will require the Byteman jar to be installed in the bootstrap classpath using the boot: option to the -javaagent JVM command line argument.

org.jboss.byteman.skip.overriding.rules
: When this system property is set, then the agent will not perform injection into overriding

method implementations. If an overriding rule is installed the agent will print a warning to `System.err` and treat the rule as if it applies only to the class named in the `CLASS` clause. This setting is not actually provided to allow rules to be misused in this way. It is a performance tuning option. The agent has to check every class as it is loaded in order to see if there are rules which apply to it. It also has to check all loaded classes when rules are dynamically loaded via the agent listener. This requires traversing the superclass chain to locate overriding rules attached to superclasses. This increases the cost of running the agent (testing indicates that the cost goes from negligible (<< 1%) to, at worst, noticeable (~ 2%) but not to significant) So, if you do not intend to use overriding rules then setting this property helps to minimise the extent to which the agent perturbs the timing of application runs. This is particularly important when testing multi-threaded applications where timing is highly significant.

`org.jboss.byteman.allow.config.updates`

When this system property is set (with any value), then the Byteman agent will update its configuration when changes to the value of system properties are submitted using the `bmsubmit` client with option `-y`. Note that this configuration property cannot be reset dynamically using the submit client. It must be set when the agent is loaded either on the JVM command line or via the `bminstall` client.

`org.jboss.byteman.sysprops.strict`

When this system property is set to `false`, then the Byteman agent will allow the `bmsubmit` client to modify any system properties. If it is unset or set to `true` then only properties with prefix "`org.jboss.byteman.`" will be modified. Note that this configuration property cannot be reset dynamically using the `bmsubmit` client. It must be set when the agent is loaded either on the JVM command line or via the `bminstall` client.