Red Hat JBoss BRMS 6.0

Development Guide

For Red Hat JBoss Developers and Rules Authors

Last Updated: 2017-11-17

Red Hat JBoss BRMS 6.0 Development Guide

For Red Hat JBoss Developers and Rules Authors

Kanchan Desai
kadesai@redhat.com

Doug Hoffman

David Le Sage
Red Hat Engineering Content Services
dlesage@redhat.com

Red Hat Content Services

Legal Notice

Abstract

A guide to using business rules for developers

Table of Contents

CHAPTER 1. INTRODUCTION

1.1. ABOUT RED HAT JBOSS BRMS

Red Hat JBoss BRMS is an open source decision management platform that combines Business Rules Management and Complex Event Processing. It automates business decisions and makes that logic available to the entire business.

Red Hat JBoss BRMS uses a centralized repository where all resources are stored. This ensures consistency, transparency, and the ability to audit across the business. Business users can modify business logic without requiring assistance from IT personnel.

Business Resource Planner is included as a technical preview with this release.

Report a bug

1.2. USE CASE: BUSINESS DECISION MANAGEMENT IN THE INSURANCE INDUSTRY WITH JBOSS BRMS

BRMS comprises a high-performance rule engine from the Drools project, a rule repository and easy to use rule authoring tools from the Drools Guvnor project, and Complex Event Processing rule engine extensions from the Drools Fusion project. It also includes OptaPlanner, a solver for complex planning problems, as a technology preview.

The consumer insurance market is extremely competitive, and it is imperative that customers receive efficient, competitive, and comprehensive services when visiting an online insurance quotation solution. An insurance provider increased revenue from their online quotation solution by upselling to the visitors of the solution relevant, additional products during the quotation process.

JBoss BRMS was integrated with the insurance providers's infrastructure so that when a request for insurance was processed, BRMS was consulted and appropriate additional products were presented with the insurance quotation:

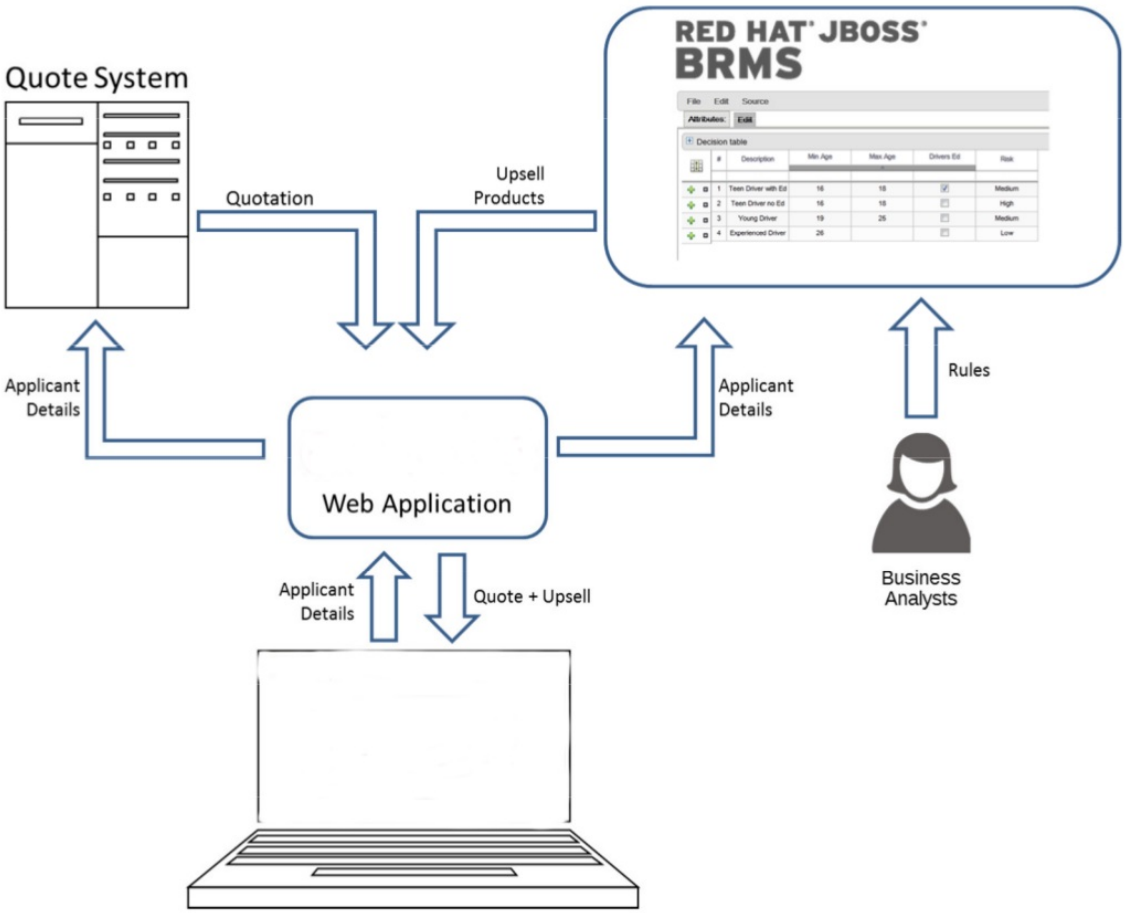


Figure 1.1. BRMS Use Case: Insurance Industry Decision Making

BRMS provided the decision management functionality, i.e. the automatic determination of the products to present to the applicant based on the rules defined by business analysts. The rules were implemented as decision tables, so they could be easily understood and modified without requiring additional support from IT.

Report a bug

CHAPTER 2. BUSINESS RULES ENGINE

2.1. THE BASICS

2.1.1. Business Rules Engine

Business Rules Engine is the rules engine provided as part of the Red Hat JBoss BRMS product. It is based on the community Drools Expert product.

Report a bug

2.1.2. Expert Systems

Expert systems are often used to refer to *production rules systems* or *Prolog-like systems*. Although acceptable, this comparison is technically incorrect as these are frameworks to build expert systems with, rather than expert systems themselves. An expert system develops once there is a model demonstrating the nature of the expert system itself; that is, a domain encompassing the aspects of an expert system which includes facilities for knowledge acquisition and explanation. *Mycin* is the most famous expert system.

Report a bug

2.1.3. Production Rules

A *production rule* is a two-part structure that uses first order logic to represent knowledge. It takes the following form:

```
when
 <conditions>
then
 <actions>
```

Report a bug

2.1.4. The Inference Engine

The *inference engine* is the part of the JBoss Rules engine which matches production facts and data to rules. It will then perform actions based on what it infers from the information. A production rules system's inference engine is *stateful* and is responsible for *truth maintenance*.

Report a bug

2.1.5. Production Memory

The **`production memory`** is where rules are stored.

Report a bug

2.1.6. Working Memory

The **`working memory`** is the part of the JBoss Rules engine where facts are asserted. From here, the facts can be modified or retracted.

Report a bug

2.1.7. Conflict Resolution Strategy

Conflict resolution is required when there are multiple rules on the agenda. As firing a rule may have side effects on the working memory, the rule engine needs to know in what order the rules should fire (for instance, firing ruleA may cause ruleB to be removed from the agenda).

Report a bug

2.1.8. Hybrid Rule Systems

A hybrid rule system pertains to using both forward-chaining and backward-chaining rule systems to process rules.

Report a bug

2.1.9. Reasoning Capabilities

JBoss Rules uses backward-chaining *reasoning capabilities* to help infer which rules to apply from the data.

Report a bug

CHAPTER 3. EXPERT SYSTEMS

3.1. PHREAK ALGORITHM

3.1.1. PHREAK Algorithm

The PHREAK algorithm used in JBoss Rules incorporates all of the existing code from Rete00. It is an enhancement of the Rete algorithm, and PHREAK incorporates the characteristics of a lazy, goal oriented algorithm where partial matching is aggressively delayed, and it also handles a large number of rules and facts. PHREAK is inspired by a number of algorithms including the following: LEAPS, RETE/UL and Collection-Oriented Match.

PHREAK has all the enhancements listed in the Rete00 algorithm. In addition, it also adds the following set of enhancements:

- Three layers of contextual memory: Node, Segment and Rule memories.
- Rule, segment, and node based linking.
- Lazy (delayed) rule evaluation.
- Stack based evaluations with pause and resume.
- Isolated rule evaluation.
- Set oriented propagations.

Report a bug

3.1.2. Three Layers of Contextual Memory

Rule evaluations in the PHREAK engine only occur while rules are linked. Before entering the beta network, the insert, update, and delete actions are queued for deployment. The rules are fired based on a simple heuristic; that is, based on the rule most likely to result in the firing of other rules, the heuristic selects the next rule for evaluation which delays the firing of other rules. Once all the inputs are populated, the rule becomes linked in. Next, a goal is created that represents the rule, and it is placed into a priority queue, which is ordered by salience. The queues themselves are associated with an AgendaGroup, and only the active AgendaGroup will inspect its queue by submitting the rule with the highest salience for evaluation. The actions of insert, update, delete phase and fireAllRules phase are achieved by this point. Accordingly, only the rule for which the goal was created is evaluated, and other potential rule evaluations are delayed. Node sharing is achieved through the process of segmentation while individual rules are evaluated.

PHREAK has 3 levels of memory. This allows for much more contextual understanding during evaluation of a Rule.

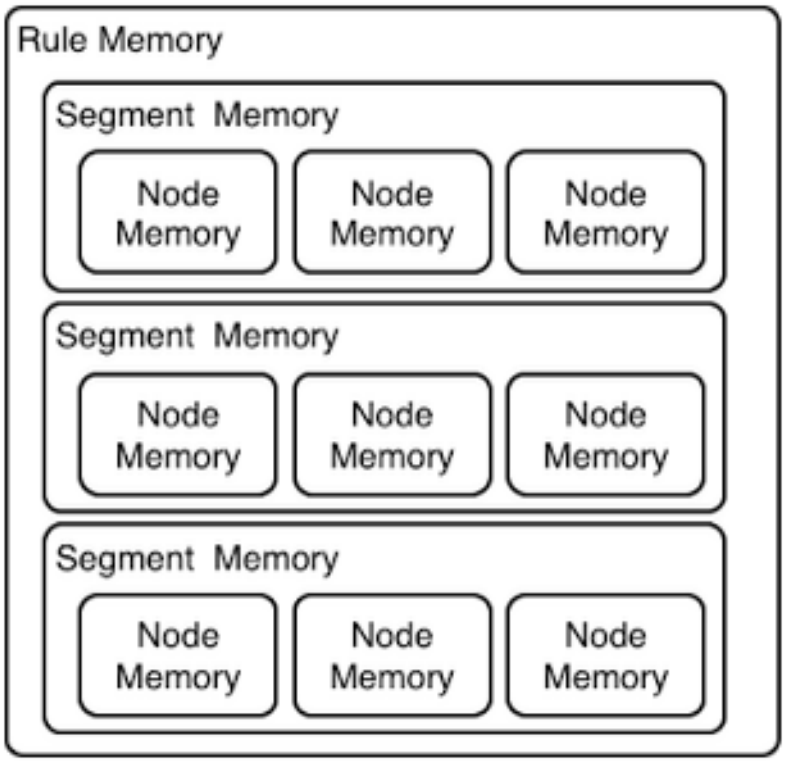


Figure 3.1. PHREAK 3 Layered memory system

Report a bug

3.1.3. Rule, Segment, and Node Based Linking

The Linking and Unlinking uses a layered bit mask system based on a network segmentation. When the rule network is built, segments are created for nodes that are shared by the same set of rules. A rule itself is made up from a path of segments; if there is no sharing, the rule will be a single segment. A bit-mask offset is assigned to each node in the segment. Another bit-mask (the layering) is assigned to each segment in the rule's path. When there is at least one input (data propagation), the node's bit is set to 'on'. When each node has its bit set to 'on,' the segment's bit is also set to 'on'. Conversely, if any node's bit is set to 'off', the segment is then also set to 'off'. If each segment in the rule's path is set to 'on', the rule is said to be linked in, and a goal is created to schedule the rule for evaluation.

The following examples illustrates the rule, segment, and node based linking in PHREAK algorithm.

Example 1: Single rule, no sharing
The example shows a single rule, with three patterns; A, B and C. It forms a single segment with bits 1, 2 and 4 for the nodes.

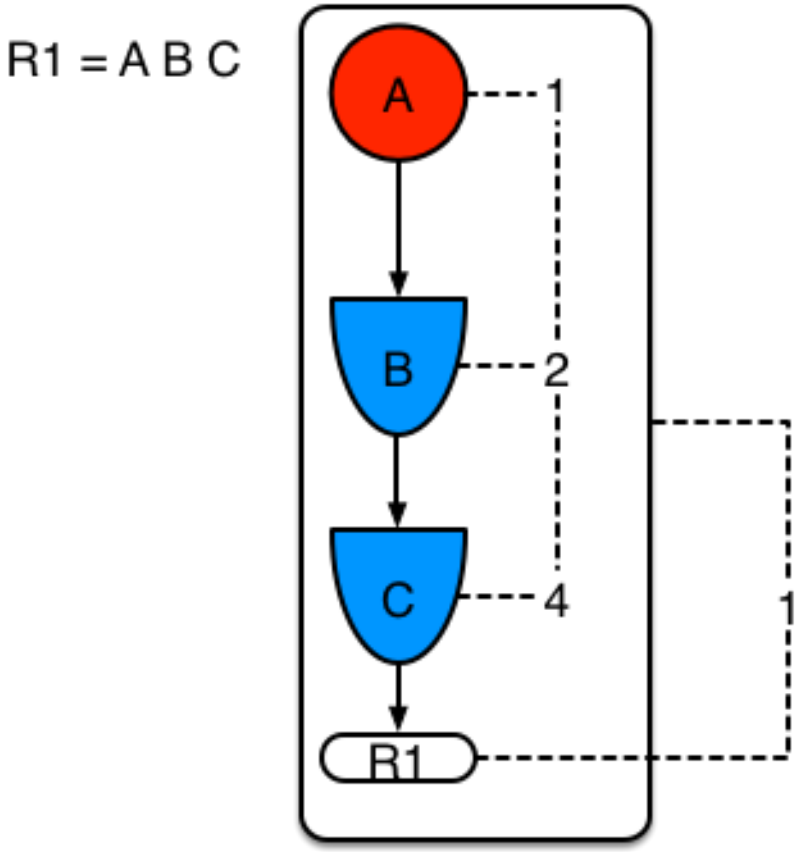


Figure 3.2. Example for a Single rule with no sharing

Example 2: Two rules with sharing

The following example demonstrates what happens when another rule is added that shares the pattern A. A is placed in its own segment, resulting in two segments per rule. These two segments form a path for their respective rules. The first segment is shared by both paths. When A is linked, the segment becomes linked; it then iterates each path the segment is shared by, setting the bit 1 to "on'. If B and C are later turned 'on', the second segment for path R1 is linked in; this causes bit 2 to be turned 'on' for R1. With bit 1 and bit 2 set to 'on' for R1, the rule is now linked and a goal is created to schedule the rule for later evaluation and firing.

When a rule is evaluated, its segments allow the results of matching to be shared. Each segment has a staging memory to queue all insert, update, and deletes for that segment. If R1 is evaluated, it will process A and result in a set of tuples. The algorithm detects that there is a segmentation split, and it will create peered tuples for each insert, update, and delete in the set and add them to R2's staging memory. These tuples will be merged with any existing staged tuples and wait for R2 to eventually be evaluated.

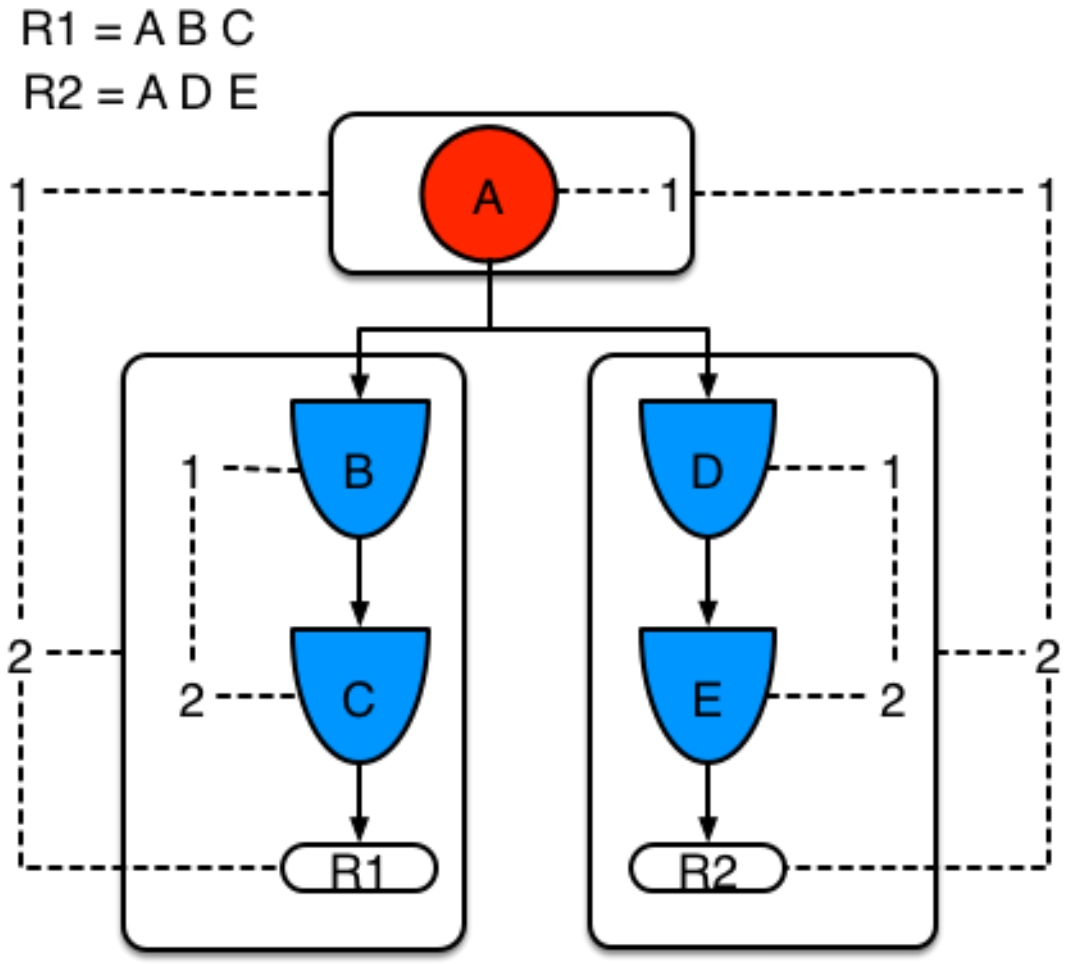


Figure 3.3. Example for two rules with sharing

Example 3: Three rules with sharing

The following example adds a third rule and demonstrates what happens when A and B are shared. Only the bits for the segments are shown this time. It demonstrating that R4 has 3 segments, R3 has 3 segments, and R1 has 2 segments. A and B are shared by R1, R3, and R4. Accordingly, D is shared by R3 and R4.

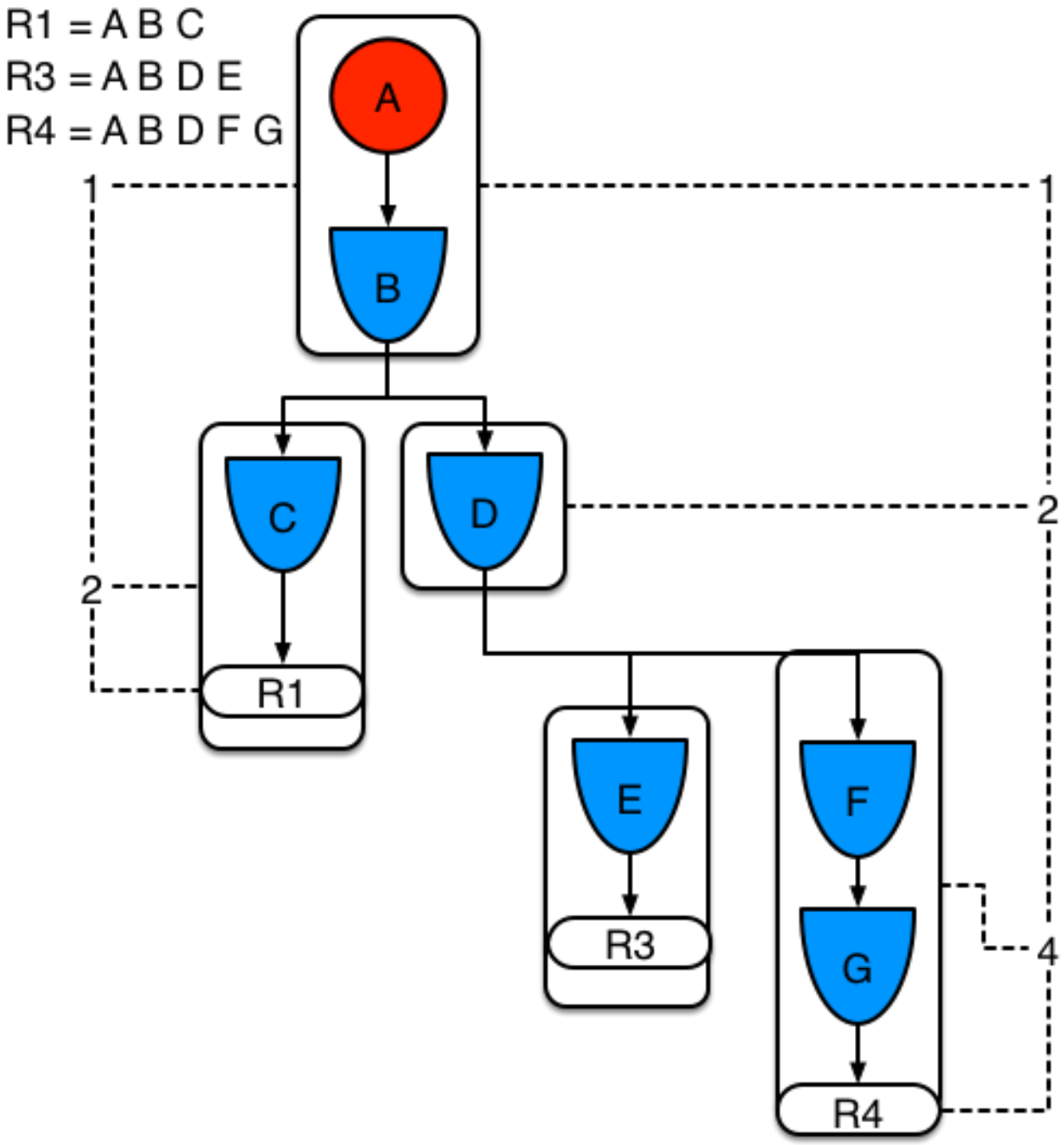


Figure 3.4. Example for Three rules with sharing

Example 4: Single rule, with sub-network and no sharing

Sub-networks are formed when a Not, Exists, or Accumulate node contain more than one element. In the following example, "B not(C)" forms the sub-network; note that "not(C)" is a single element and does not require a sub network, and it is merged inside of the Not node.

The sub-network gets its own segment. R1 still has a path of two segments. The sub-network forms another "inner" path. When the sub-network is linked in, it will link in the outer segment.

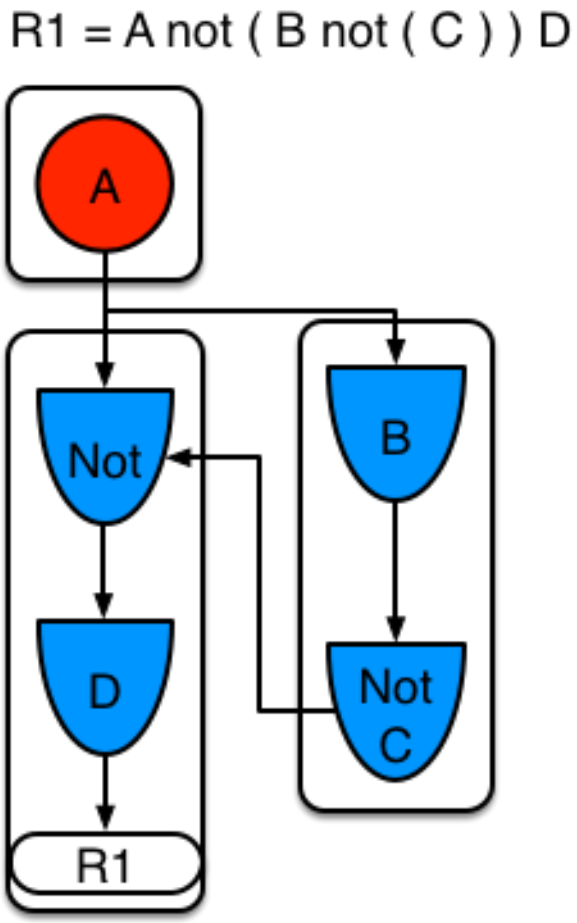


Figure 3.5. Example for a Single rule with sub-network and no sharing

Example 5: Two rules: one with a sub-network and sharing

The example shows that the sub-network nodes can be shared by a rule that does not have a sub-network. This results in the sub-network segment being split into two.

Note that nodes with constraints and accumulate nodes have special behaviour and can never unlink a segment; they are always considered to have their bits on.

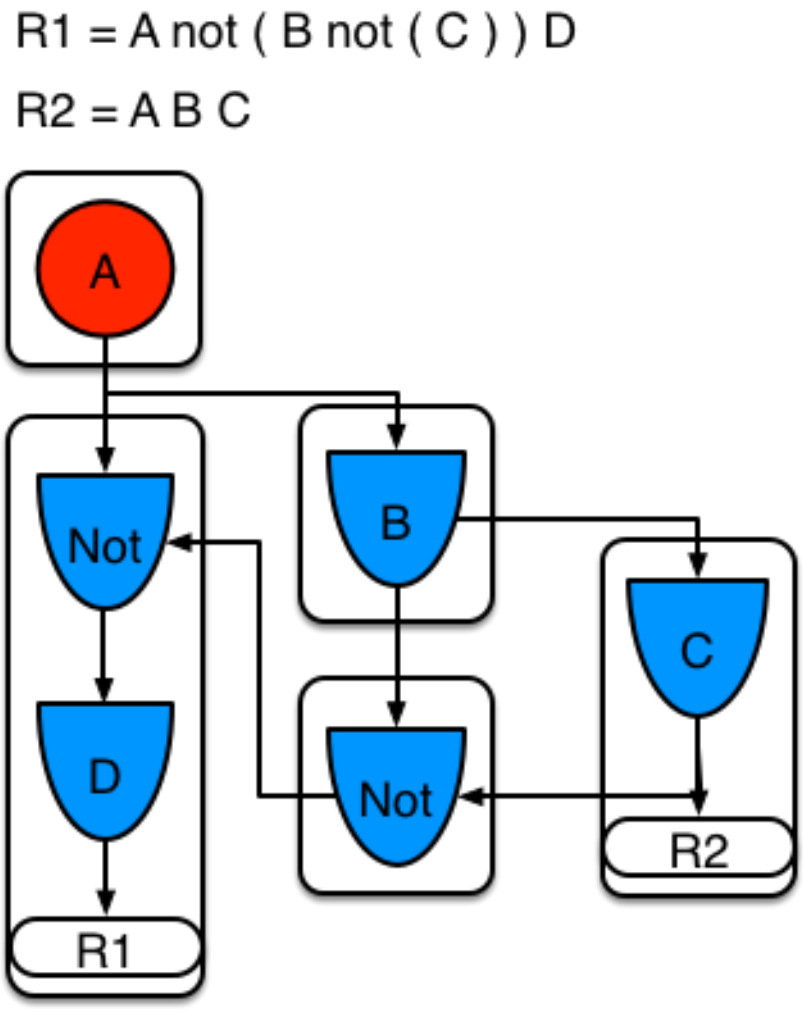


Figure 3.6. Example for Two rules, one with a sub-network and sharing

Report a bug

3.1.4. Delayed and Stack Based Evaluations

As discussed with the layering of segments in a rules path, a goal is created for rule evaluation based on the rule path segment's status. The same bit-mask technique is used to also track dirty nodes, segments, and rules. This process allows for a rule already linked to be scheduled for another evaluation if the previous evaluation labeled it dirty. This reevaluation ensures that no rule will ever evaluate partial matches. That is, it will not link rules with no data because they will not result in rule instances. Rete, however, will produce martial match attempt for all nodes, even empty ones.

While the incremental rule evaluation always starts from the root node, the dirty bit masks are used to allow nodes and segments that are not dirty to be skipped. This heuristic is fairy basic, and it uses at least one item of data per node. Future work attempts to delay linking even further, such as *arc consistency*, will determine if rule instance will fire regardless of matching.

During the phase of segmentation, other rules are delayed and evaluated. Note that all rule evaluations are incremental, and they will not waste work recomputing matches that have already been produced. The evaluations algorithm is stack based instead of method recursive. The evaluations can be paused and resumed at any time by using a StackEntry to represent the current node being evaluated. A StackEntry is created for the outer path segment and sub-network segment of a rule evaluation. When the evaluation reaches a sub-network, the sub-network segment is evaluated first. When the set reaches the end of the sub-networkd path, it is merged into a staging list for the outer node it reacts to.

The previous StackEntry is then resumed, it can then process the results of the sub-network. This process benefits from all the work being processed in batch before it is propagated to the child node, which is more efficient for accumulate nodes.

Report a bug

3.1.5. Propagations and Isolated Rules

PHREAK propagation is set oriented (or collection-oriented) instead of tuple oriented. For the evaluated rule, PHREAK will visit the first node and process all queued insert, update, and delete actions. The results are added to a set and the set is propagated to the child node. In the child node, all queued insert, update, and deletes are processed, which adds the results to the same set. Once the processing has finished, the set is propagated to the next child node, and this process repeats until the terminal node is reached. These actions create a single pass, pipeline type effect, that is isolated to the current rule being evaluated. Accordingly, this leads to the creation of a batch process effect which provides performance advantages for certain rule constructs such as sub-networks with accumulates.

As mentioned prior, the process of StackEntry adds the benefit of efficient batch-based work before propagating to the child node. This same stack system is efficient for backward chaining. When a rule evaluation reaches a query node, it pauses the current evaluation by placing it on the stack. The query is then evaluated and produces a result set, which is saved in a memory location for the resumed StackEntry to pick up and propagate to the child node. If the query itself called other queries, the process would repeat; accordingly, the current query would be paused, and a new evaluation would be created for the current query node.

Report a bug

3.1.6. RETE to PHREAK

In general, a single rule will not evaluate any faster with PHREAK than it does with RETE. Both a given rule and the same data set, which uses a root context object to enable and disable matching, attempt the same amount of matches, produce the same number of rule instances, and take roughly the same time. However, variations occur for the use case with subnetworks and accumulates.

PHREAK is considered more forgiving than RETE for poorly written rule bases; it also displays a more graceful degradation of performance as the number of rules and complexity increases.

RETE produces partial matches for rules that do not have data in all the joints; PHREAK avoids any partial matching. Accordingly, PHREAK's performance will not slow down as your system grows.

AgendaGroups did not help RETE performance, that is, all rules were evaluated at all times, regardless of the group. This also occurred with salience, which relied on context objects to limit matching attempts. PHREAK only evaluates rules for the active AgendaGroup. Within that active group, PHREAK will avoid evaluation of rules (via salience) that do not result in rule instance firings. With PHREAK, AgendaGroups and salience now become useful performance tools.

NOTE

With PHREAK, root context objects are no longer needed as they may be counter productive to performance. That is, they may force the flushing and recreation of matches for rules.

Report a bug

3.1.7. Switching Between PHREAK and ReteOO

Switching Using System Properties

For users to switch between the PHREAK algorithm and the ReteOO algorithm, the **`drools.ruleEngine`** system properties need to be edited with the following values:

```
drools.ruleEngine=phreak
```

or

```
drools.ruleEngine=reteoo
```

The previous value of "phreak" is the default value for 6.0.

The Maven GAV (Group, Artifact, Version) value for ReteOO is depicted below:

```
<dependency>
  <groupId>org.drools</groupId>
  <artifactId>drools-reteoo</artifactId>
  <version>${drools.version}</version>
</dependency>
```

Switching in KieBaseConfiguration

When creating a particular KieBase, it is possible to specify the rule engine algorithm in the KieBaseConfiguration:

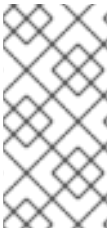
```
import org.kie.api.KieBase;
import org.kie.api.KieBaseConfiguration;
import org.kie.api.KieServices;
import org.kie.api.runtime.KieContainer;
...
```

```
KieServices kservices = KieServices.Factory.get();
KieBaseConfiguration kconfig =
kieServices.Factory.get().newKieBaseConfiguration();

// you can either specify phreak (default)
kconfig.setOption(RuleEngineOption.PHREAK);

// or legacy ReteOO
kconfig.setOption(RuleEngineOption.RETEOO);

// and then create a KieBase for the selected algorithm
(getKieClasspathContainer() is just an example)
KieContainer container = kservices.getKieClasspathContainer();
KieBase kbase = container.newKieBase(kieBaseName, kconfig);
```



NOTE

Take note that switching to ReteOO requires `drools-reteoo-(version).jar` to exist on the classpath. If not, the JBoss Rules Engine reverts back to PHREAK and issues a warning. This applies for switching with KieBaseConfiguration and System Properties.

Report a bug

3.2. RETE ALGORITHM

3.2.1. ReteOO

The Rete implementation used in JBoss Rules is called *ReteOO*. It is an enhanced and optimized implementation of the Rete algorithm specifically for object-oriented systems. The Rete Algorithm has now been deprecated, and PHREAK is an enhancement of Rete. However, Rete can still be used by developers. This section describes how the Rete Algorithm functions.

Report a bug

3.2.2. The Rete Root Node

When using ReteOO, the root node is where all objects enter the network. From there, it immediately goes to the ObjectTypeNode.

Report a bug

3.2.3. The ObjectTypeNode

The *ObjectTypeNode* helps to reduce the workload of the rules engine. If there are several objects and the rules engine tried to evaluate every single node against every object, it would waste a lot of cycles. To make things efficient, the ObjectTypeNode is used so the engine only passes objects to the nodes that match the object's type. This way, if an application asserts a new Account, it won't propagate to the nodes for the Order object.

In JBoss Rules, an object which has been asserted will retrieve a list of valid ObjectTypesNodes via a lookup in a HashMap from the object's Class. If this list doesn't exist it scans all the ObjectTypeNodes finding valid matches which it caches in the list. This enables JBoss Rules to match against any Class type that matches with an `instanceof` check.

Report a bug

3.2.4. AlphaNodes

AlphaNodes are used to evaluate literal conditions. When a rule has multiple literal conditions for a single object type, they are linked together. This means that if an application asserts an Account object, it must first satisfy the first literal condition before it can proceed to the next AlphaNode.

AlphaNodes are propagated using ObjectTypeNodes.

Report a bug

3.2.5. Hashing

JBoss Rules uses *hashing* to extend Rete by optimizing the propagation from ObjectTypeNode to AlphaNode. Each time an AlphaNode is added to an ObjectTypeNode it adds the literal value as a key to the HashMap with the AlphaNode as the value. When a new instance enters the ObjectType node, rather than propagating to each AlphaNode, it can instead retrieve the correct AlphaNode from the HashMap, thereby avoiding unnecessary literal checks.

Report a bug

3.2.6. BetaNodes

BetaNodes are used to compare two objects and their fields. The objects may be the same or different types.

Report a bug

3.2.7. Alpha Memory

Alpha memory refers to the left input on a BetaNode. In JBoss Rules, this input remembers all incoming objects.

Report a bug

3.2.8. Beta Memory

Beta memory is the term used to refer to the right input of a BetaNode. It remembers all incoming tuples.

Report a bug

3.2.9. Lookups with BetaNodes

When facts enter from one side, you can do a hash lookup returning potentially valid candidates (referred to as indexing). At any point a valid join is found, the Tuple will join with the Object (referred to as a partial match) and then propagate to the next node.

Report a bug

3.2.10. LeftInputNodeAdapters

A *LeftInputNodeAdapter* takes an Object as an input and propagates a single Object Tuple.

Report a bug

3.2.11. Terminal Nodes

Terminal nodes are used to indicate when a single rule has matched all its conditions (that is, the rule has a full match). A rule with an 'or' conditional disjunctive connective will result in a sub-rule generation for each possible logically branch. Because of this, one rule can have multiple terminal nodes.

Report a bug

3.2.12. Node Sharing

Node sharing is used to prevent unnecessary redundancy. Because many rules repeat the same patterns, node sharing allows users to collapse those patterns so they do not have to be reevaluated for every single instance.

The following two rules share the first pattern but not the last:

```
rule
when
    vehicle( $sedan : name == "sedan" )
    $driver: Driver( typeCar == $sedan )
then
    System.out.println( $driver.getName() + " drives sedan" );
end
```

```
rule
when
    Vehicle( $sedan : name == "sedan" )
    $driver : Driver( typeCar != $sedan )
then
    System.out.println( $driver.getName() + " does not drive sedan" );
end
```

Report a bug

3.2.13. Join Attempts

Each successful join attempt in RETE produces a tuple (or token, or partial match) that will be propagated to the child nodes. For this reason, it is characterized as a tuple oriented algorithm. For each child node that it reaches, it will attempt to join with the other side of the node; moreover, each successful join attempt will be propagated straight away. This creates a descent recursion effect that targets from the point of entry in the beta network and spreads to all the reachable leaf nodes.

Report a bug

3.3. STRONG AND LOOSE COUPLING

3.3.1. Loose Coupling

Loose coupling involves "loosely" linking rules so that the execution of one rule will not lead to the execution of another.

Generally, a design exhibiting loose coupling is preferable because it allows for more flexibility. If the rules are all strongly coupled, they are likely to be inflexible. More significantly, it indicates that deploying a rule engine is overkill for the situation.

Report a bug

3.3.2. Strong Coupling

Strong coupling is a way of linking rules. If rules are strongly-coupled, it means executing one rule will directly result in the execution of another. In other words, there is a clear chain of logic. (A clear chain can be hard-coded, or implemented using a decision tree.)

Report a bug

3.4. ADVANTAGES OF A RULE ENGINE

3.4.1. Declarative Programming

Declarative programming refers to the way the rule engine allows users to declare "what to do" as opposed to "how to do it". The key advantage of this point is that using rules can make it easy to express solutions to difficult problems and consequently have those solutions verified. Rules are much easier to read than code.

Report a bug

3.4.2. Logic and Data Separation

*Logic and Data separation*refers to the process of de-coupling logic and data components. Using this method, the logic can be spread across many domain objects or controllers and it can all be organized in one or more discrete rules files.

Report a bug

3.4.3. Knowledge Base

A knowledge base is a collection of rules which have been compiled by the **`KnowledgeBuilder`**. It is a repository of all the application's knowledge definitions. It may contain rules, processes, functions, and type models. The Knowledge Base itself does not contain instance data (known as facts). Instead, sessions are created from the Knowledge Base into which data can be inserted and where process instances may be started. It is recommended that Knowledge Bases be cached where possible to allow for repeated session creation.

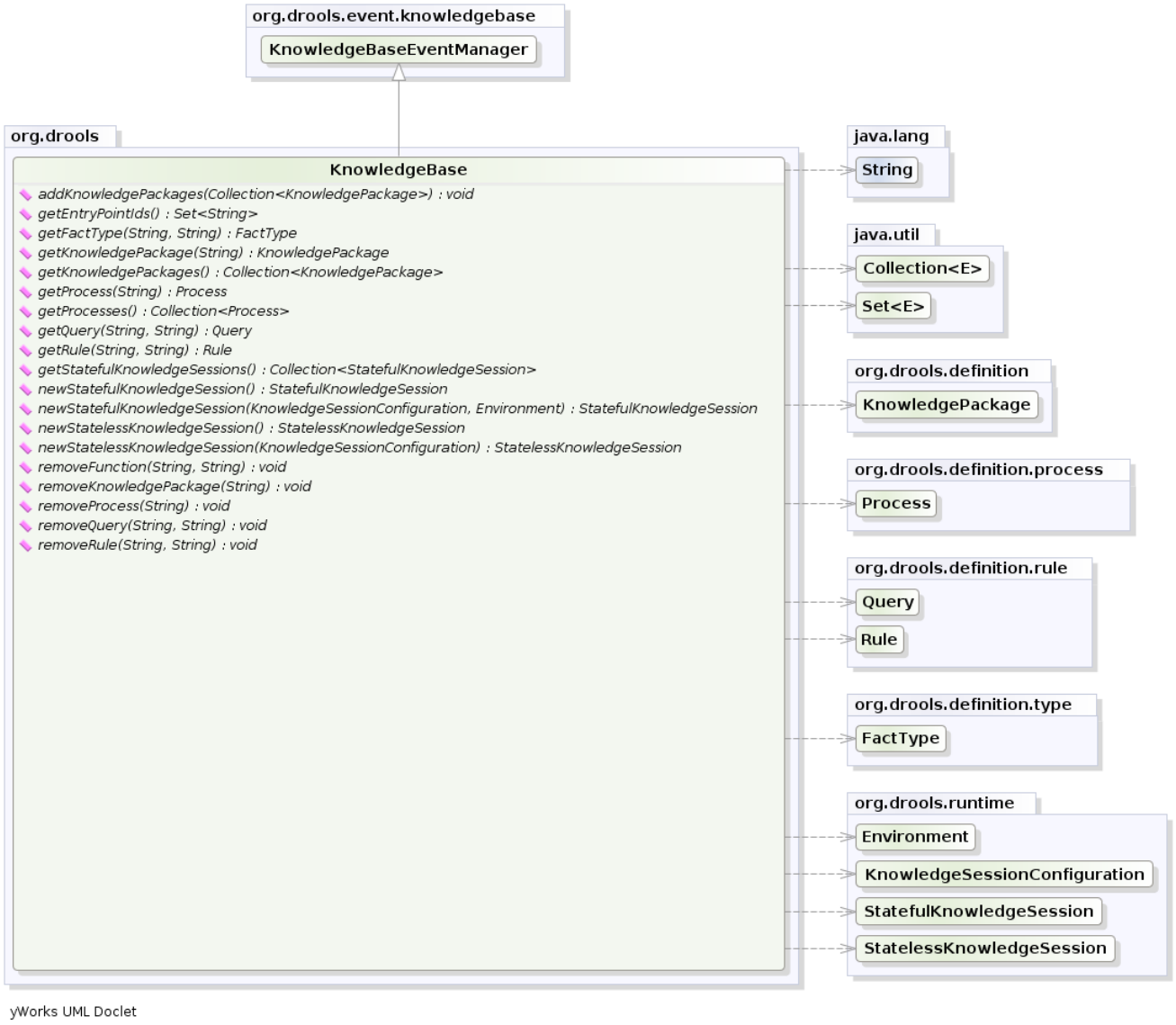


Figure 3.7. KnowledgeBase

Report a bug

CHAPTER 4. MAVEN

4.1. LEARN ABOUT MAVEN

4.1.1. About Maven

Apache Maven is a distributed build automation tool used in Java application development to build and manage software projects. Maven uses configuration XML files called POM (Project Object Model) to define project properties and manage the build process. POM files describe the project's module and component dependencies, build order, and targets for the resulting project packaging and output. This ensures that projects are built in a correct and uniform manner.

Maven uses repositories to store Java libraries, plug-ins, and other build artifacts. Repositories can be either local or remote. A local repository is a download of artifacts from a remote repository cached on a local machine. A remote repository is any other repository accessed using common protocols, such as `http://` when located on an HTTP server, or `file://` when located on a file server. The default repository is the public remote Maven 2 Central Repository .

Configuration of Maven is performed by modifying the `settings.xml` file. You can either configure global Maven settings in the ***`M2_HOME`***`/conf/settings.xml` file, or user-level settings in the ***`USER_HOME`***`/.m2/settings.xml` file.

For more information about Maven, see Welcome to Apache Maven .

For more information about Maven repositories, see Apache Maven Project - Introduction to Repositories.

For more information about Maven POM files, see the Apache Maven Project POM Reference .

Report a bug

4.1.2. About the Maven POM File

The Project Object Model, or POM, file is a configuration file used by Maven to build projects. It is an XML file that contains information about the project and how to build it, including the location of the source, test, and target directories, the project dependencies, plug-in repositories, and goals it can execute. It can also include additional details about the project including the version, description, developers, mailing list, license, and more. A `pom.xml` file requires some configuration options and will default all others. See Section 4.1.3, “Minimum Requirements of a Maven POM File” for details.

The schema for the `pom.xml` file can be found at http://maven.apache.org/maven-v4_0_0.xsd.

For more information about POM files, see the Apache Maven Project POM Reference .

Report a bug

4.1.3. Minimum Requirements of a Maven POM File

Minimum requirements

The minimum requirements of a `pom.xml` file are as follows:

- project root
- modelVersion

- groupId - the id of the project's group
- artifactId - the id of the artifact (project)
- version - the version of the artifact under the specified group

Sample pom.xml file

A basic **pom.xml** file might look like this:

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.jboss.app</groupId>
  <artifactId>my-app</artifactId>
  <version>1</version>
</project>
```

Report a bug

4.1.4. About the Maven Settings File

The Maven **settings.xml** file contains user-specific configuration information for Maven. It contains information that should not be distributed with the **pom.xml** file, such as developer identity, proxy information, local repository location, and other settings specific to a user.

There are two locations where the **settings.xml** can be found.

In the Maven install

The settings file can be found in the ***M2_HOME*/conf/** directory. These settings are referred to as **global** settings. The default Maven settings file is a template that can be copied and used as a starting point for the user settings file.

In the user's install

The settings file can be found in the ***USER_HOME*/.m2/** directory. If both the Maven and user **settings.xml** files exist, the contents are merged. Where there are overlaps, the user's **settings.xml** file takes precedence.

The following is an example of a Maven **settings.xml** file:

```
<settings>
    <profiles>
        <profile>
            <id>my-profile</id>
            <activation>
                <activeByDefault>true</activeByDefault>
            </activation>
            <repositories>
                <repository>
                    <id>fusesource</id>

<url>http://repo.fusesource.com/nexus/content/groups/public/</url>
                    <snapshots>
                        <enabled>false</enabled>
```

```
                    </snapshots>
                    <releases>
                        <enabled>true</enabled>
                    </releases>
                </repository>
                ...
            </repositories>
        </profile>
    </profiles>
    ...
</settings>
```

The schema for the **settings.xml** file can be found at http://maven.apache.org/xsd/settings-1.0.0.xsd.

Report a bug

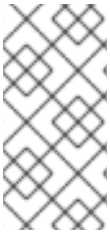
4.1.5. KIE Plugin

The KIE plugin for Maven ensures that artifact resources are validated and pre-compiled, it is recommended that this is used at all times. To use the plugin simply add it to the build section of the Maven pom.xml

Example 4.1. Adding the KIE plugin to a Maven pom.xml

```
<build>
  <plugins>
    <plugin>
      <groupId>org.kie</groupId>
      <artifactId>kie-maven-plugin</artifactId>
      <version>${project.version}</version>
      <extensions>true</extensions>
    </plugin>
  </plugins>
</build>
```

Building a KIE module without the Maven plugin will copy all the resources, as is, into the resulting JAR. When that JAR is loaded by the runtime, it will attempt to build all the resources then. If there are compilation issues it will return a null KieContainer. It also pushes the compilation overhead to the runtime. In general this is not recommended, and the Maven plugin should always be used.



NOTE

For compiling decision tables and processes, appropriate dependencies must be added either to project dependencies (as compile scope) or as plugin dependencies. For decision tables the dependency is **org.drools:drools-decisiontables** and for processes **org.jbpm:jbpm-bpmn2**.

Report a bug

4.1.6. Maven Versions and Dependencies

Maven supports a number of mechanisms to manage versioning and dependencies within applications. Modules can be published with specific version numbers, or they can use the SNAPSHOT suffix. Dependencies can specify version ranges to consume, or take avantage of SNAPSHOT mechanism.

If you always want to use the newest version, Maven has two keywords you can use as an alternative to version ranges. You should use these options with care as you are no longer in control of the plugins/dependencies you are using.

When you depend on a plugin or a dependency, you can use the a version value of LATEST or RELEASE. LATEST refers to the latest released or snapshot version of a particular artifact, the most recently deployed artifact in a particular repository. RELEASE refers to the last non-snapshot release in the repository. In general, it is not a best practice to design software which depends on a non-specific version of an artifact. If you are developing software, you might want to use RELEASE or LATEST as a convenience so that you don't have to update version numbers when a new release of a third-party library is released. When you release software, you should always make sure that your project depends on specific versions to reduce the chances of your build or your project being affected by a software release not under your control. Use LATEST and RELEASE with caution, if at all.

Here's an example illustrating the various options. In the Maven repository, com.foo:my-foo has the following metadata:

```
<metadata>
  <groupId>com.foo</groupId>
  <artifactId>my-foo</artifactId>
  <version>2.0.0</version>
  <versioning>
    <release>1.1.1</release>
    <versions>
      <version>1.0</version>
      <version>1.0.1</version>
      <version>1.1</version>
      <version>1.1.1</version>
      <version>2.0.0</version>
    </versions>
    <lastUpdated>20090722140000</lastUpdated>
  </versioning>
</metadata>
```

If a dependency on that artifact is required, you have the following options (other version ranges can be specified of course, just showing the relevant ones here): Declare an exact version (will always resolve to 1.0.1):

```
<version>[1.0.1]</version>
```

Declare an explicit version (will always resolve to 1.0.1 unless a collision occurs, when Maven will select a matching version):

```
<version>1.0.1</version>
```

Declare a version range for all 1.x (will currently resolve to 1.1.1):

```
<version>[1.0.0,2.0.0)</version>
```

Declare an open-ended version range (will resolve to 2.0.0):

```
<version>[1.0.0,)</version>
```

Declare the version as LATEST (will resolve to 2.0.0):

```
<version>LATEST</version>
```

Declare the version as RELEASE (will resolve to 1.1.1):

```
<version>RELEASE</version>
```

Note that by default your own deployments will update the "latest" entry in the Maven metadata, but to update the "release" entry, you need to activate the "release-profile" from the Maven super POM. You can do this with either "-Prelease-profile" or "-DperformRelease=true"

Report a bug

4.1.7. Remote Repository Setup

The maven settings.xml is used to configure Maven execution.

The settings.xml file can be located in 3 locations, the actual settings used is a merge of those 3 locations.

- The Maven install: $M2_HOME/conf/settings.xml
- A user's install: ${user.home}/.m2/settings.xml
- Folder location specified by the system propert kie.maven.settings.custom

The settings.xml is used to specify the location of remote repositories. It is important that you activate the profile that specifies the remote repository, typically this can be done using "activeByDefault":

```
<profiles>
  <profile>
    <id>profile-1</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    ...
  </profile>
</profiles>
```

Report a bug

CHAPTER 5. KIE API

5.1. KIE FRAMEWORK

5.1.1. KIE Systems

The various aspects, or life cycles, of KIE systems in the JBoss Rules environment can typically be broken down into the following labels:

- **Author**
  - Knowledge author using UI metaphors such as DRL, BPMN2, decision tables, and class models.
- **Build**
  - Builds the authored knowledge into deployable units.
  - For KIE this unit is a JAR.
- **Test**
  - Test KIE knowledge before it is deployed to the application.
- **Deploy**
  - Deploys the unit to a location where applications may use them.
  - KIE uses Maven style repository.
- **Utilize**
  - The loading of a JAR to provide a KIE session (KieSession), for which the application can interact with.
  - KIE exposes the JAR at runtime via a KIE container (KieContainer).
  - KieSessions, for the runtimes to interact with, are created from the KieContainer.
- **Run**
  - System interaction with the KieSession, via API.
- **Work**
  - User interaction with the KieSession, via command line or UI.
- **Manage**
  - Manage any KieSession or KieContainer.

Report a bug

5.1.2. KieBase

A **`KieBase`** is a repository of all the application's knowledge definitions. It contains rules, processes, functions, and type models. The **`KieBase`** itself does not contain data; instead, sessions are created

from the **`KieBase`** into which data can be inserted, and, ultimately, process instances may be started. Creating the **`KieBase`** can be quite heavy, whereas session creation is very light; therefore, it is recommended that **`KieBase`** be cached where possible to allow for repeated session creation. Accordingly, the caching mechanism is automatically provided by the **`KieContainer`**.

Table 5.1. kbase Attributes

Attribute name	Default value	Admitted values	Meaning
name	none	any	The name which retrieves the KieBase from the KieContainer. This is the only mandatory attribute.
includes	none	any comma separated list	A comma separated list of other KieBases contained in this kmodule. The artifacts of all these KieBases will also be included in this one.
packages	all	any comma separated list	By default all the JBoss Rules artifacts under the resources folder, at any level, are included into the KieBase. This attribute allows to limit the artifacts that will be compiled in this KieBase to only the ones belonging to the list of packages.
default	false	true, false	Defines if this KieBase is the default one for this module, so it can be created from the KieContainer without passing any name to it. There can be at most one default KieBase in each module.

Attribute name	Default value	Admitted values	Meaning
equalsBehavior	identity	identity, equality	Defines the behavior of JBoss Rules when a new fact is inserted into the Working Memory. With identity it always create a new FactHandle unless the same object isn't already present in the Working Memory, while with equality only if the newly inserted object is not equal (according to its equal method) to an already existing fact.
eventProcessingMode	cloud	cloud, stream	When compiled in cloud mode the KieBase treats events as normal facts, while in stream mode allow temporal reasoning on them.
declarativeAgenda	disabled	disabled, enabled	Defines if the Declarative Agenda is enabled or not.

Report a bug

5.1.3. KieSession

The **`KieSession`** stores and executes on runtime data. It is created from the **`KieBase`**, or, more easily, created directly from the **`KieContainer`** if it has been defined in the kmodule.xml file

Table 5.2. ksession Attributes

Attribute name	Default value	Admitted values	Meaning
name	none	any	Unique name of this KieSession. Used to fetch the KieSession from the KieContainer. This is the only mandatory attribute.

Attribute name	Default value	Admitted values	Meaning
type	stateful	stateful, stateless	A stateful session allows to iteratively work with the Working Memory, while a stateless one is a one-off execution of a Working Memory with a provided data set.
default	false	true, false	Defines if this KieSession is the default one for this module, so it can be created from the KieContainer without passing any name to it. In each module there can be at most one default KieSession for each type.
clockType	realtime	realtime, pseudo	Defines if events timestamps are determined by the system clock or by a psuedo clock controlled by the application. This clock is specially useful for unit testing temporal rules.
beliefSystem	simple	simple, jtms, defeasible	Defines the type of belief system used by the KieSession.

Report a bug

5.1.4. KieFileSystem

It is also possible to define the **KieBases** and **KieSessions** belonging to a KieModule programmatically instead of the declarative definition in the kmodule.xml file. The same programmatic API also allows in explicitly adding the file containing the Kie artifacts instead of automatically read them from the resources folder of your project. To do that it is necessary to create a **KieFileSystem**, a sort of virtual file system, and add all the resources contained in your project to it.

Like all other Kie core components you can obtain an instance of the **KieFileSystem** from the **KieServices**. The kmodule.xml configuration file must be added to the filesystem. This is a mandatory step. Kie also provides a convenient fluent API, implemented by the **KieModuleModel**, to programmatically create this file.

To do this in practice it is necessary to create a **KieModuleModel** from the **KieServices**, configure

it with the desired **KieBases** and **KieSessions**, convert it in XML and add the XML to the **KieFileSystem**. This process is shown by the following example:

Example 5.1. Creating a kmodule.xml programmatically and adding it to a KieFileSystem

```
KieServices kieServices = KieServices.Factory.get();
KieModuleModel kieModuleModel = kieServices.newKieModuleModel();

KieBaseModel kieBaseModel1 = kieModuleModel.newKieBaseModel( "KBase1 ")
        .setDefault( true )
        .setEqualsBehavior( EqualityBehaviorOption.EQUALITY )
        .setEventProcessingMode( EventProcessingOption.STREAM );

KieSessionModel ksessionModel1 = kieBaseModel1.newKieSessionModel(
"KSession1" )
        .setDefault( true )
        .setType( KieSessionModel.KieSessionType.STATEFUL )
        .setClockType( ClockTypeOption.get("realtime") );

KieFileSystem kfs = kieServices.newKieFileSystem();
```

At this point it is also necessary to add to the **KieFileSystem**, through its fluent API, all others Kie artifacts composing your project. These artifacts have to be added in the same position of a corresponding usual Maven project.

Report a bug

5.1.5. KieResources

Example 5.2. Adding Kie artifacts to a KieFileSystem

```
KieFileSystem kfs = ...
kfs.write( "src/main/resources/KBase1/ruleSet1.drl",
stringContainingAValidDRL )
        .write( "src/main/resources/dtable.xls",
                kieServices.getResources().newInputStreamResource(
dtableFileStream ) );
```

This example shows that it is possible to add the Kie artifacts both as plain Strings and as **Resources**. In the latter case the **Resources** can be created by the **KieResources** factory, also provided by the **KieServices**. The **KieResources** provides many convenient factory methods to convert an **InputStream**, a **URL**, a **File**, or a **String** representing a path of your file system to a **Resource** that can be managed by the **KieFileSystem**.

Normally the type of a **Resource** can be inferred from the extension of the name used to add it to the **KieFileSystem**. However it also possible to not follow the Kie conventions about file extensions and explicitly assign a specific **ResourceType** to a **Resource** as shown below:

Example 5.3. Creating and adding a Resource with an explicit type

```
KieFileSystem kfs = ...
```

```
kfs.write( "src/main/resources/myDrl.txt",
           kieServices.getResources().newInputStreamResource( drlStream
)
                      .setResourceType(ResourceType.DRL) );
```

Add all the resources to the **KieFileSystem** and build it by passing the **KieFileSystem** to a **KieBuilder**

When the contents of a **KieFileSystem** are successfully built, the resulting **KieModule** is automatically added to the **KieRepository**. The **KieRepository** is a singleton acting as a repository for all the available **KieModule**s.

Report a bug

5.2. BUILDING WITH MAVEN

5.2.1. The kmodule

BRMS 6.0 introduces a new configuration and convention approach to building knowledge bases instead of using the programmatic builder approach in 5.x. The builder is still available to fall back on, as it's used for the tooling integration.

Building now uses Maven, and aligns with Maven practices. A KIE project or module is simply a Maven Java project or module; with an additional metadata file META-INF/kmodule.xml. The kmodule.xml file is the descriptor that selects resources to knowledge bases and configures those knowledge bases and sessions. There is also alternative XML support via Spring and OSGi BluePrints.

While standard Maven can build and package KIE resources, it will not provide validation at build time. There is a Maven plugin which is recommended to use to get build time validation. The plugin also generates many classes, making the runtime loading faster too.

KIE uses defaults to minimise the amount of configuration. With an empty kmodule.xml being the simplest configuration. There must always be a kmodule.xml file, even if empty, as it's used for discovery of the JAR and its contents.

Maven can either 'mvn install' to deploy a KieModule to the local machine, where all other applications on the local machine use it. Or it can 'mvn deploy' to push the KieModule to a remote Maven repository. Building the Application will pull in the KieModule and populate the local Maven repository in the process.

JARs can be deployed in one of two ways. Either added to the classpath, like any other JAR in a Maven dependency listing, or they can be dynamically loaded at runtime. KIE will scan the classpath to find all the JARs with a kmodule.xml in it. Each found JAR is represented by the KieModule interface. The terms classpath KieModule and dynamic KieModule are used to refer to the two loading approaches. While dynamic modules supports side by side versioning, classpath modules do not. Further once a module is on the classpath, no other version may be loaded dynamically.

The kmodule.xml allows to define and configure one or more **KieBase**s and for each **KieBase** all the different **KieSession**s that can be created from it, as shown by the following example:

Example 5.4. A sample kmodule.xml file

```
<kmodule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
        xmlns="http://jboss.org/kie/6.0.0/kmodule">
  <kbase name="KBase1" default="true" eventProcessingMode="cloud"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg1">
    <ksession name="KSession2_1" type="stateful" default="true/">
    <ksession name="KSession2_1" type="stateless" default="false/"
beliefSystem="jtms">
  </kbase>
  <kbase name="KBase2" default="false" eventProcessingMode="stream"
equalsBehavior="equality" declarativeAgenda="enabled"
packages="org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    <ksession name="KSession2_1" type="stateful" default="false"
clockType="realtime">
      <fileLogger file="drools.log" threaded="true" interval="10"/>
      <workItemHandlers>
        <workItemHandler name="name"
type="org.domain.WorkItemHandler"/>
      </workItemHandlers>
      <listeners>
        <ruleRuntimeEventListener
type="org.domain.RuleRuntimeListener"/>
        <agendaEventListener type="org.domain.FirstAgendaListener"/>
        <agendaEventListener type="org.domain.SecondAgendaListener"/>
        <processEventListener type="org.domain.ProcessListener"/>
      </listeners>
    </ksession>
  </kbase>
</kmodule>
```

Here 2 **KieBase**s have been defined and it is possible to instantiate 2 different types of **KieSession**s from the first one, while only one from the second.

Report a bug

5.2.2. Creating a KIE Project

A Kie Project has the structure of a normal Maven project with the only peculiarity of including a kmodule.xml file defining in a declaratively way the **KieBase**s and **KieSession**s that can be created from it. This file has to be placed in the resources/META-INF folder of the Maven project while all the other Kie artifacts, such as DRL or a Excel files, must be stored in the resources folder or in any other subfolder under it.

Since meaningful defaults have been provided for all configuration aspects, the simplest kmodule.xml file can contain just an empty kmodule tag like the following:

Example 5.5. An empty kmodule.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<kmodule xmlns="http://jboss.org/kie/6.0.0/kmodule"/>
```

In this way the kmodule will contain one single default **KieBase**. All Kie assets stored under the resources folder, or any of its subfolders, will be compiled and added to it. To trigger the building of these artifacts it is enough to create a **KieContainer** for them.

Report a bug

5.2.3. Creating a KIE Container

Illustrated below is a simple case example on how to create a **KieContainer** that reads files built from the classpath:

Example 5.6. Creating a KieContainer from the classpath

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kContainer = kieServices.getKieClasspathContainer();
```

After defining a kmodule.xml, it is possible to simply retrieve the KieBases and KieSessions from the KieContainer using their names.

Example 5.7. Retriving KieBases and KieSessions from the KieContainer

```
KieServices kieServices = KieServices.Factory.get();
KieContainer kContainer = kieServices.getKieClasspathContainer();

KieBase kBase1 = kContainer.getKieBase("KBase1");
KieSession kieSession1 = kContainer.newKieSession("KSession2_1");
StatelessKieSession kieSession2 =
kContainer.newStatelessKieSession("KSession2_2");
```

It has to be noted that since KSession2_1 and KSession2_2 are of 2 different types (the first is stateful, while the second is stateless) it is necessary to invoke 2 different methods on the **KieContainer** according to their declared type. If the type of the **KieSession** requested to the **KieContainer** doesn't correspond with the one declared in the kmodule.xml file the **KieContainer** will throw a **RuntimeException**. Also since a **KieBase** and a **KieSession** have been flagged as default is it possible to get them from the **KieContainer** without passing any name.

Example 5.8. Retriving default KieBases and KieSessions from the KieContainer

```
KieContainer kContainer = ...

KieBase kBase1 = kContainer.getKieBase(); // returns KBase1
KieSession kieSession1 = kContainer.newKieSession(); // returns
KSession2_1
```

Since a Kie project is also a Maven project the groupId, artifactId and version declared in the pom.xml file are used to generate a **ReleaseId** that uniquely identifies this project inside your application. This allows creation of a new KieContainer from the project by simply passing its **ReleaseId** to the **KieServices**.

Example 5.9. Creating a KieContainer of an existing project by ReleaseId

```
KieServices kieServices = KieServices.Factory.get();
ReleaseId releaseId = kieServices.newReleaseId( "org.acme",
"myartifact", "1.0" );
KieContainer kieContainer = kieServices.newKieContainer( releaseId );
```

Report a bug

5.2.4. KieServices

KieServices is the interface from where it possible to access all the Kie building and runtime facilities:

In this way all the Java sources and the Kie resources are compiled and deployed into the KieContainer which makes its contents available for use at runtime.

Report a bug

5.3. KIE DEPLOYMENT

5.3.1. KieRepository

When the contents of a **KieFileSystem** are successfully built, the resulting **KieModule** is automatically added to the **KieRepository**. The **KieRepository** is a singleton acting as a repository for all the available **KieModules**.

After this it is possible to create through the **KieServices** a new **KieContainer** for that **KieModule** using its **ReleaseId**. However, since in this case the **KieFileSystem** don't contain any pom.xml file (it is possible to add one using the **KieFileSystem.writePomXML** method), Kie cannot determine the **ReleaseId** of the **KieModule** and assign to it a default one. This default **ReleaseId** can be obtained from the **KieRepository** and used to identify the **KieModule** inside the **KieRepository** itself. The following example shows this whole process.

Example 5.10. Building the contents of a KieFileSystem and creating a KieContainer

```
KieServices kieServices = KieServices.Factory.get();
KieFileSystem kfs = ...
kieServices.newKieBuilder( kfs ).buildAll();
KieContainer kieContainer =
kieServices.newKieContainer(kieServices.getRepository().getDefaultReleas
eId());
```

At this point it is possible to get **KieBases** and create new **KieSessions** from this **KieContainer** exactly in the same way as in the case of a **KieContainer** created directly from the classpath.

It is a best practice to check the compilation results. The **KieBuilder** reports compilation results of 3 different severities: ERROR, WARNING and INFO. An ERROR indicates that the compilation of the project failed and in the case no **KieModule** is produced and nothing is added to the **KieRepository**. WARNING and INFO results can be ignored, but are available for inspection.

Example 5.11. Checking that a compilation didn't produce any error

```
KieBuilder kieBuilder = kieServices.newKieBuilder( kfs ).buildAll();
assertEquals( 0, kieBuilder.getResults().getMessages(
Message.Level.ERROR ).size() );
```

Report a bug

5.3.2. Session Modification

The **KieBase** is a repository of all the application's knowledge definitions. It will contain rules, processes, functions, and type models. The **KieBase** itself does not contain data; instead, sessions are created from the **KieBase** into which data can be inserted and from which process instances may be started. The **KieBase** can be obtained from the **KieContainer** containing the **KieModule** where the **KieBase** has been defined.

Sometimes, for instance in a OSGi environment, the **KieBase** needs to resolve types that are not in the default class loader. In this case it will be necessary to create a **KieBaseConfiguration** with an additional class loader and pass it to **KieContainer** when creating a new **KieBase** from it.

Example 5.12. Creating a new KieBase with a custom ClassLoader

```
KieServices kieServices = KieServices.Factory.get();
KieBaseConfiguration kbaseConf = kieServices.newKieBaseConfiguration(
null, MyType.class.getClassLoader() );
KieBase kbase = kieContainer.newKieBase( kbaseConf );
```

The **KieBase** creates and returns **KieSession** objects, and it may optionally keep references to those. When **KieBase** modifications occur those modifications are applied against the data in the sessions. This reference is a weak reference and it is also optional, which is controlled by a boolean flag.

Report a bug

5.3.3. KieScanner

The **KieScanner** allows continuous monitoring of your Maven repository to check whether a new release of a Kie project has been installed. A new release is deployed in the **KieContainer** wrapping that project. The use of the **KieScanner** requires kie-ci.jar to be on the classpath.

A **KieScanner** can be registered on a **KieContainer** as in the following example.

Example 5.13. Registering and starting a KieScanner on a KieContainer

```
KieServices kieServices = KieServices.Factory.get();
ReleaseId releaseId = kieServices.newReleaseId( "org.acme",
"myartifact", "1.0-SNAPSHOT" );
KieContainer kContainer = kieServices.newKieContainer( releaseId );
KieScanner kScanner = kieServices.newKieScanner( kContainer );
```

```
// Start the KieScanner polling the Maven repository every 10 seconds
kScanner.start( 10000L );
```

In this example the **KieScanner** is configured to run with a fixed time interval, but it is also possible to run it on demand by invoking the **scanNow()** method on it. If the **KieScanner** finds in the Maven repository an updated version of the Kie project used by that **KieContainer** it automatically downloads the new version and triggers an incremental build of the new project. From this moment all the new **KieBase**s and **KieSession**s created from that **KieContainer** will use the new project version.

Report a bug

5.4. RUNNING IN KIE

5.4.1. KieRuntime

The **KieRuntime** provides methods that are applicable to both rules and processes, such as setting globals and registering channels. ("Exit point" is an obsolete synonym for "channel".)

Report a bug

5.4.2. Globals in KIE

Globals are named objects that are made visible to the rule engine, but in a way that is fundamentally different from the one for facts: changes in the object backing a global do not trigger reevaluation of rules. Still, globals are useful for providing static information, as an object offering services that are used in the RHS of a rule, or as a means to return objects from the rule engine. When you use a global on the LHS of a rule, make sure it is immutable, or, at least, don't expect changes to have any effect on the behavior of your rules.

A global must be declared in a rules file, and then it needs to be backed up with a Java object.

```
global java.util.List list
```

With the Knowledge Base now aware of the global identifier and its type, it is now possible to call **ksession.setGlobal()** with the global's name and an object, for any session, to associate the object with the global. Failure to declare the global type and identifier in DRL code will result in an exception being thrown from this call.

```
List list = new ArrayList();
ksession.setGlobal("list", list);
```

Make sure to set any global before it is used in the evaluation of a rule. Failure to do so results in a **NullPointerException**.

Report a bug

5.4.3. Event Packages

The event package provides means to be notified of rule engine events, including rules firing, objects being asserted, etc. This allows separation of logging and auditing activities from the main part of your application (and the rules).

The **`KieRuntimeEventManager`** interface is implemented by the **`KieRuntime`** which provides two interfaces, **`RuleRuntimeEventManager`** and **`ProcessEventManager`**. We will only cover the **`RuleRuntimeEventManager`** here.

The **`RuleRuntimeEventManager`** allows for listeners to be added and removed, so that events for the working memory and the agenda can be listened to.

The following code snippet shows how a simple agenda listener is declared and attached to a session. It will print matches after they have fired.

Example 5.14. Adding an AgendaEventListener

```
ksession.addEventListener( new DefaultAgendaEventListener() {
    public void afterMatchFired(AfterMatchFiredEvent event) {
        super.afterMatchFired( event );
        System.out.println( event );
    }
});
```

JBoss Rules also provides **`DebugRuleRuntimeEventListener`** and **`DebugAgendaEventListener`** which implement each method with a debug print statement. To print all Working Memory events, you add a listener like this:

Example 5.15. Adding a DebugRuleRuntimeEventListener

```
ksession.addEventListener( new DebugRuleRuntimeEventListener() );
```

All emitted events implement the **`KieRuntimeEvent`** interface which can be used to retrieve the actual **`KnowlegeRuntime`** the event originated from.

The events currently supported are:

- MatchCreatedEvent
- MatchCancelledEvent
- BeforeMatchFiredEvent
- AfterMatchFiredEvent
- AgendaGroupPushedEvent
- AgendaGroupPoppedEvent
- ObjectInsertEvent
- ObjectDeletedEvent
- ObjectUpdatedEvent

- ProcessCompletedEvent
- ProcessNodeLeftEvent
- ProcessNodeTriggeredEvent
- ProcessStartEvent

Report a bug

5.4.4. KieRuntimeLogger

The KieRuntimeLogger uses the comprehensive event system in JBoss Rules to create an audit log that can be used to log the execution of an application for later inspection, using tools such as the Eclipse audit viewer.

Example 5.16. FileLogger

```
KieRuntimeLogger logger =
KieServices.Factory.get().getLoggers().newFileLogger( session, "audit"
);
...
// Be sure to close the logger otherwise it will not write.
logger.close();
```

Report a bug

5.4.5. CommandExecutor Interface

KIE has the concept of stateful or stateless sessions. Stateful sessions have already been covered, which use the standard KieRuntime, and can be worked with iteratively over time. Stateless is a one-off execution of a KieRuntime with a provided data set. It may return some results, with the session being disposed at the end, prohibiting further iterative interactions. You can think of stateless as treating an engine like a function call with optional return results.

The foundation for this is the **`CommandExecutor`** interface, which both the stateful and stateless interfaces extend. This returns an **`ExecutionResults`**:

The **`CommandExecutor`** allows for commands to be executed on those sessions, the only difference being that the StatelessKieSession executes **`fireAllRules()`** at the end before disposing the session. The commands can be created using the **`CommandExecutor`** .The Javadocs provide the full list of the allowed comands using the **`CommandExecutor`**.

setGlobal and getGlobal are two commands relevant to JBoss Rules.

Set Global calls setGlobal underneath. The optional boolean indicates whether the command should return the global's value as part of the **`ExecutionResults`**. If true it uses the same name as the global name. A String can be used instead of the boolean, if an alternative name is desired.

Example 5.17. Set Global Command

```
StatelessKieSession ksession = kbase.newStatelessKieSession();
ExecutionResults bresults =
```

```
    ksession.execute( CommandFactory.newSetGlobal( "stilton", new
Cheese( "stilton" ), true);
Cheese stilton = bresults.getValue( "stilton" );
```

Allows an existing global to be returned. The second optional String argument allows for an alternative return name.

Example 5.18. Get Global Command

```
StatelessKieSession ksession = kbase.newStatelessKieSession();
ExecutionResults bresults =
    ksession.execute( CommandFactory.getGlobal( "stilton" );
Cheese stilton = bresults.getValue( "stilton" );
```

All the above examples execute single commands. The **`BatchExecution`** represents a composite command, created from a list of commands. It will iterate over the list and execute each command in turn. This means you can insert some objects, start a process, call fireAllRules and execute a query, all in a single **`execute(...)`** call, which is quite powerful.

The StatelessKieSession will execute **`fireAllRules()`** automatically at the end. However the keen-eyed reader probably has already noticed the **`FireAllRules`** command and wondered how that works with a StatelessKieSession. The **`FireAllRules`** command is allowed, and using it will disable the automatic execution at the end; think of using it as a sort of manual override function.

Any command, in the batch, that has an out identifier set will add its results to the returned **`ExecutionResults`** instance.

Example 5.19. BatchExecution Command

```
StatelessKieSession ksession = kbase.newStatelessKieSession();

List cmds = new ArrayList();
cmds.add( CommandFactory.newInsertObject( new Cheese( "stilton", 1),
"stilton") );
cmds.add( CommandFactory.newStartProcess( "process cheeses" ) );
cmds.add( CommandFactory.newQuery( "cheeses" ) );
ExecutionResults bresults = ksession.execute(
CommandFactory.newBatchExecution( cmds ) );
Cheese stilton = ( Cheese ) bresults.getValue( "stilton" );
QueryResults qresults = ( QueryResults ) bresults.getValue( "cheeses" );
```

In the above example multiple commands are executed, two of which populate the **`ExecutionResults`**. The query command defaults to use the same identifier as the query name, but it can also be mapped to a different identifier.

All commands support XML and jSON marshalling using XStream, as well as JAXB marshalling. This is covered in the *Rule Commands* section: Section 8.1, “Available API”.

Report a bug

5.5. KIE CONFIGURATION

5.5.1. Build Result Severity

In some cases, it is possible to change the default severity of a type of build result. For instance, when a new rule with the same name of an existing rule is added to a package, the default behavior is to replace the old rule by the new rule and report it as an INFO. This is probably ideal for most use cases, but in some deployments the user might want to prevent the rule update and report it as an error.

Changing the default severity for a result type, configured like any other option in JBoss Rules, can be done by API calls, system properties or configuration files. As of this version, JBoss Rules supports configurable result severity for rule updates and function updates. To configure it using system properties or configuration files, the user has to use the following properties:

Example 5.20. Setting the severity using properties

```
// sets the severity of rule updates
drools.kbuilder.severity.duplicateRule = <INFO|WARNING|ERROR>
// sets the severity of function updates
drools.kbuilder.severity.duplicateFunction = <INFO|WARNING|ERROR>
```

Report a bug

5.5.2. StatelessKieSession

The **StatelessKieSession** wraps the **KieSession**, instead of extending it. Its main focus is on the decision service type scenarios. It avoids the need to call **dispose()**. Stateless sessions do not support iterative insertions and the method call **fireAllRules()** from Java code; the act of calling **execute()** is a single-shot method that will internally instantiate a **KieSession**, add all the user data and execute user commands, call **fireAllRules()**, and then call **dispose()**. While the main way to work with this class is via the **BatchExecution** (a subinterface of **Command**) as supported by the **CommandExecutor** interface, two convenience methods are provided for when simple object insertion is all that's required. The **CommandExecutor** and **BatchExecution** are talked about in detail in their own section.

Our simple example shows a stateless session executing a given collection of Java objects using the convenience API. It will iterate the collection, inserting each element in turn.

Example 5.21. Simple StatelessKieSession execution with a Collection

```
StatelessKieSession ksession = kbase.newStatelessKieSession();
ksession.execute( collection );
```

If this was done as a single Command it would be as follows:

Example 5.22. Simple StatelessKieSession execution with InsertElements Command

```
ksession.execute( CommandFactory.newInsertElements( collection ) );
```

If you wanted to insert the collection itself, and the collection's individual elements, then **`CommandFactory.newInsert(collection)`** would do the job.

Methods of the **`CommandFactory`** create the supported commands, all of which can be marshalled using XStream and the **`BatchExecutionHelper`**. **`BatchExecutionHelper`** provides details on the XML format as well as how to use Drools Pipeline to automate the marshalling of **`BatchExecution`** and **`ExecutionResults`**.

`StatelessKieSession` supports globals, scoped in a number of ways. We cover the non-command way first, as commands are scoped to a specific execution call. Globals can be resolved in three ways.

- The StatelessKieSession method **`getGlobals()`** returns a Globals instance which provides access to the session's globals. These are shared for *all* execution calls. Exercise caution regarding mutable globals because execution calls can be executing simultaneously in different threads.

  Example 5.23. Session scoped global

  ```
  StatelessKieSession ksession = kbase.newStatelessKieSession();
  // Set a global hbnSession, that can be used for DB interactions
  in the rules.
  ksession.setGlobal( "hbnSession", hibernateSession );
  // Execute while being able to resolve the "hbnSession"
  identifier.
  ksession.execute( collection );
  ```

- Using a delegate is another way of global resolution. Assigning a value to a global (with **`setGlobal(String, Object)`**) results in the value being stored in an internal collection mapping identifiers to values. Identifiers in this internal collection will have priority over any supplied delegate. Only if an identifier cannot be found in this internal collection, the delegate global (if any) will be used.

- The third way of resolving globals is to have execution scoped globals. Here, a **`Command`** to set a global is passed to the **`CommandExecutor`**.

The **`CommandExecutor`** interface also offers the ability to export data via "out" parameters. Inserted facts, globals and query results can all be returned.

Example 5.24. Out identifiers

```
// Set up a list of commands
List cmds = new ArrayList();
cmds.add( CommandFactory.newSetGlobal( "list1", new ArrayList(), true )
);
cmds.add( CommandFactory.newInsert( new Person( "jon", 102 ), "person" )
);
cmds.add( CommandFactory.newQuery( "Get People" "getPeople" );

// Execute the list
ExecutionResults results =
  ksession.execute( CommandFactory.newBatchExecution( cmds ) );

// Retrieve the ArrayList
```

```
results.getValue( "list1" );
// Retrieve the inserted Person fact
results.getValue( "person" );
// Retrieve the query as a QueryResults instance.
results.getValue( "Get People" );
```

Report a bug

5.5.3. Marshalling

The **KieMarshallers** are used to marshal and unmarshal KieSessions.

An instance of the **KieMarshallers** can be retrieved from the **KieServices**. A simple example is shown below:

Example 5.25. Simple Marshaller Example

```
// ksession is the KieSession
// kbase is the KieBase
ByteArrayOutputStream baos = new ByteArrayOutputStream();
Marshaller marshaller =
KieServices.Factory.get().getMarshallers().newMarshaller( kbase );
marshaller.marshall( baos, ksession );
baos.close();
```

However, with marshalling, you will need more flexibility when dealing with referenced user data. To achieve this use the **ObjectMarshallingStrategy** interface. Two implementations are provided, but users can implement their own. The two supplied strategies are **IdentityMarshallingStrategy** and **SerializeMarshallingStrategy**. **SerializeMarshallingStrategy** is the default, as shown in the example above, and it just calls the **Serializable** or **Externalizable** methods on a user instance. **IdentityMarshallingStrategy** creates an integer id for each user object and stores them in a Map, while the id is written to the stream. When unmarshalling it accesses the **IdentityMarshallingStrategy** map to retrieve the instance. This means that if you use the **IdentityMarshallingStrategy**, it is stateful for the life of the Marshaller instance and will create ids and keep references to all objects that it attempts to marshal. Below is the code to use an Identity Marshalling Strategy.

Example 5.26. IdentityMarshallingStrategy

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
KieMarshallers kMarshallers = KieServices.Factory.get().getMarshallers()
ObjectMarshallingStrategy oms =
kMarshallers.newIdentityMarshallingStrategy()
Marshaller marshaller =
        kMarshallers.newMarshaller( kbase, new
ObjectMarshallingStrategy[]{ oms } );
marshaller.marshall( baos, ksession );
baos.close();
```

Im most cases, a single strategy is insufficient. For added flexibility, the

`ObjectMarshallingStrategyAcceptor` interface can be used. This Marshaller has a chain of strategies, and while reading or writing a user object it iterates the strategies asking if they accept responsibility for marshalling the user object. One of the provided implementations is **`ClassFilterAcceptor`**. This allows strings and wild cards to be used to match class names. The default is "*.*", so in the above example the Identity Marshalling Strategy is used which has a default "*.*" acceptor.

Assuming that we want to serialize all classes except for one given package, where we will use identity lookup, we could do the following:

Example 5.27. IdentityMarshallingStrategy with Acceptor

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
KieMarshallers kMarshallers = KieServices.Factory.get().getMarshallers()
ObjectMarshallingStrategyAcceptor identityAcceptor =
        kMarshallers.newClassFilterAcceptor( new String[] {
"org.domain.pkg1.*" } );
ObjectMarshallingStrategy identityStrategy =
        kMarshallers.newIdentityMarshallingStrategy( identityAcceptor
);
ObjectMarshallingStrategy sms =
kMarshallers.newSerializeMarshallingStrategy();
Marshaller marshaller =
        kMarshallers.newMarshaller( kbase,
                                    new ObjectMarshallingStrategy[]{
identityStrategy, sms } );
marshaller.marshall( baos, ksession );
baos.close();
```

Note that the acceptance checking order is in the natural order of the supplied elements.

Also note that if you are using scheduled matches (i.e. some of your rules use timers or calendars) they are marshallable only if, before you use it, you configure your KieSession to use a trackable timer job factory manager as follows:

Example 5.28. Configuring a trackable timer job factory manager

```
KieSessionConfiguration ksconf =
KieServices.Factory.get().newKieSessionConfiguration();
ksconf.setOption(TimerJobFactoryOption.get("trackable"));
KSession ksession = kbase.newKieSession(ksconf, null);
```

Report a bug

5.5.4. KIE Persistence

Longterm out of the box persistence with Java Persistence API (JPA) is possible with JBoss Rules. It is necessary to have some implementation of the Java Transaction API (JTA) installed. For development purposes the Bitronix Transaction Manager is suggested, as it's simple to set up and works embedded, but for production use JBoss Transactions is recommended.

Example 5.29. Simple example using transactions

```
KieServices kieServices = KieServices.Factory.get();
Environment env = kieServices.newEnvironment();
env.set( EnvironmentName.ENTITY_MANAGER_FACTORY,
         Persistence.createEntityManagerFactory( "emf-name" ) );
env.set( EnvironmentName.TRANSACTION_MANAGER,
         TransactionManagerServices.getTransactionManager() );

// KieSessionConfiguration may be null, and a default will be used
KieSession ksession =
        kieServices.getStoreServices().newKieSession( kbase, null, env
);
int sessionId = ksession.getId();

UserTransaction ut =
  (UserTransaction) new InitialContext().lookup(
"java:comp/UserTransaction" );
ut.begin();
ksession.insert( data1 );
ksession.insert( data2 );
ksession.startProcess( "process1" );
ut.commit();
```

To use a JPA, the Environment must be set with both the **EntityManagerFactory** and the **TransactionManager**. If rollback occurs the ksession state is also rolled back, hence it is possible to continue to use it after a rollback. To load a previously persisted KieSession you'll need the id, as shown below:

Example 5.30. Loading a KieSession

```
KieSession ksession =
        kieServices.getStoreServices().loadKieSession( sessionId,
kbase, null, env );
```

To enable persistence several classes must be added to your persistence.xml, as in the example below:

Example 5.31. Configuring JPA

```
<persistence-unit name="org.drools.persistence.jpa" transaction-
type="JTA">
   <provider>org.hibernate.ejb.HibernatePersistence</provider>
   <jta-data-source>jdbc/BitronixJTADataSource</jta-data-source>
   <class>org.drools.persistence.info.SessionInfo</class>
   <class>org.drools.persistence.info.WorkItemInfo</class>
   <properties>
         <property name="hibernate.dialect"
value="org.hibernate.dialect.H2Dialect"/>
         <property name="hibernate.max_fetch_depth" value="3"/>
         <property name="hibernate.hbm2ddl.auto" value="update" />
         <property name="hibernate.show_sql" value="true" />
         <property name="hibernate.transaction.manager_lookup_class"
```

```
value="org.hibernate.transaction.BTMTransactionManagerLookup" />
   </properties>
</persistence-unit>
```

The jdbc JTA data source would have to be configured first. Bitronix provides a number of ways of doing this, and its documentation should be consulted for details. For a quick start, here is the programmatic approach:

Example 5.32. Configuring JTA DataSource

```
PoolingDataSource ds = new PoolingDataSource();
ds.setUniqueName( "jdbc/BitronixJTADataSource" );
ds.setClassName( "org.h2.jdbcx.JdbcDataSource" );
ds.setMaxPoolSize( 3 );
ds.setAllowLocalTransactions( true );
ds.getDriverProperties().put( "user", "sa" );
ds.getDriverProperties().put( "password", "sasa" );
ds.getDriverProperties().put( "URL", "jdbc:h2:mem:mydb" );
ds.init();
```

Bitronix also provides a simple embedded JNDI service, ideal for testing. To use it, add a jndi.properties file to your META-INF folder and add the following line to it:

Example 5.33. JNDI properties

```
java.naming.factory.initial=bitronix.tm.jndi.BitronixInitialContextFacto
ry
```

Report a bug

CHAPTER 6. RULE SYSTEMS

6.1. PATTERNS

A pattern is an important *conditional element*. It can potentially match on each fact that is inserted in the working memory.

A pattern contains zero or more constraints, and it has an optional pattern binding. The diagram below shows the syntax for this.

Figure 6.1. Pattern

In its simplest form, with no constraints, a pattern matches against a fact of the given type. In the following case the type is **Cheese**, which means that the pattern will match against all **Person** objects in the Working Memory:

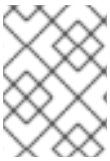
```
Person()
```

The type need not be the actual class of some fact object. Patterns may refer to superclasses or even interfaces, thereby potentially matching facts from many different classes.

```
Object() // matches all objects in the working memory
```

Inside of the pattern parenthesis is where all the action happens: it defines the constraints for that pattern. For example, with a age related constraint:

```
Person( age == 100 )
```



NOTE

For backwards compatibility reasons it's allowed to suffix patterns with the `;` character. But it is not recommended to do that.

Report a bug

6.2. POSITIONAL ARGUMENTS

Patterns support positional arguments on type declarations.

Positional arguments are ones where you don't need to specify the field name, as the position maps to a known named field. i.e. Person(name == "mark") can be rewritten as Person("mark";). The semicolon ';' is important so that the engine knows that everything before it is a positional argument. Otherwise we might assume it was a boolean expression, which is how it could be interpreted after the semicolon. You can mix positional and named arguments on a pattern by using the semicolon ';' to separate them. Any variables used in a positional that have not yet been bound will be bound to the field that maps to that position.

```
declare Cheese
```

```
    name : String
    shop : String
    price : int
end
```

Example patterns, with two constraints and a binding. Remember semicolon ';' is used to differentiate the positional section from the named argument section. Variables and literals and expressions using just literals are supported in positional arguments, but not variables. Positional arguments are always resolved using unification.

```
Cheese( "stilton", "Cheese Shop", p; )
Cheese( "stilton", "Cheese Shop"; p : price )
Cheese( "stilton"; shop == "Cheese Shop", p : price )
Cheese( name == "stilton"; shop == "Cheese Shop", p : price )
```

Positional arguments that are given a previously declared binding will constrain against that using unification; these are referred to as input arguments. If the binding does not yet exist, it will create the declaration binding it to the field represented by the position argument; these are referred to as output arguments.

Report a bug

6.3. QUERIES

Queries are used to search the working memory for facts that match a basic condition or pattern. It contains only the structure of the rule, so that you specify neither "when" nor "then". A query has an optional set of parameters, each of which can be optionally typed. If the type is not given, the type Object is assumed. The engine will attempt to coerce the values as needed. Query names are global to the KieBase; so do not add queries of the same name to different packages for the same RuleBase.

The semicolon ';' used in queries is important so that the engine knows that everything before it is a positional argument.

To return the results use **`ksession.getQueryResults("name")`**, where "name" is the query's name. This returns a list of query results, which allow you to retrieve the objects that matched the query.

The first example presents a simple query for all the people over the age of 30. The second one, using parameters, combines the age limit with a location.

Example 6.1. Query People over the age of 30

```
query "people over the age of 30"
    person : Person( age > 30 )
end
```

Example 6.2. Query People over the age of x, and who live in y

```
query "people over the age of x" (int x, String y)
    person : Person( age > x, location == y )
end
```

We iterate over the returned QueryResults using a standard "for" loop. Each element is a QueryResultsRow which we can use to access each of the columns in the tuple. These columns can be accessed by bound declaration name or index position.

Example 6.3. Query People over the age of 30

```
QueryResults results = ksession.getQueryResults( "people over the age of
30" );
System.out.println( "we have " + results.size() + " people over the age
of 30" );

System.out.println( "These people are are over 30:" );

for ( QueryResultsRow row : results ) {
    Person person = ( Person ) row.get( "person" );
    System.out.println( person.getName() + "\n" );
}
```

Support for positional syntax has been added for more compact code. By default the declared type order in the type declaration matches the argument position. But it possible to override these using the @position annotation. This allows patterns to be used with positional arguments, instead of the more verbose named arguments.

```
declare Cheese
    name : String @position(1)
    shop : String @position(2)
    price : int @position(0)
end
```

The @Position annotation, in the org.drools.definition.type package, can be used to annotate original pojos on the classpath. Currently only fields on classes can be annotated. Inheritance of classes is supported, but not interfaces or methods. The isContainedIn query below demonstrates the use of positional arguments in a pattern; **`Location(x, y;)`** instead of **`Location( thing == x, location == y)`**.

Queries can now call other queries, this combined with optional query arguments provides derivation query style backward chaining. Positional and named syntax is supported for arguments. It is also possible to mix both positional and named, but positional must come first, separated by a semi colon. Literal expressions can be passed as query arguments, but at this stage you cannot mix expressions with variables. Here is an example of a query that calls another query. Note that 'z' here will always be an 'out' variable. The '?' symbol means the query is pull only, once the results are returned you will not receive further results as the underlying data changes.

```
declare Location
    thing : String
    location : String
end

query isContainedIn( String x, String y )
    Location(x, y;)
    or
    ( Location(z, y;) and ?isContainedIn(x, z;) )
end
```

As previously mentioned you can use live "open" queries to reactively receive changes over time from the query results, as the underlying data it queries against changes. Notice the "look" rule calls the query without using '?'.

```
query isContainedIn( String x, String y )
    Location(x, y;)
    or
    ( Location(z, y;) and isContainedIn(x, z;) )
end

rule look when
    Person( $l : likes )
    isContainedIn( $l, 'office'; )
then
   insertLogical( $l 'is in the office' );
end
```

JBoss Rules supports unification for derivation queries, in short this means that arguments are optional. It is possible to call queries from java leaving arguments unspecified using the static field org.drools.core.runtime.rule.Variable.v - note you must use 'v' and not an alternative instance of Variable. These are referred to as 'out' arguments. Note that the query itself does not declare at compile time whether an argument is in or an out, this can be defined purely at runtime on each use. The following example will return all objects contained in the office.

```
results = ksession.getQueryResults( "isContainedIn", new Object[] {
Variable.v, "office" } );
l = new ArrayList<List<String>>();
for ( QueryResultsRow r : results ) {
    l.add( Arrays.asList( new String[] { (String) r.get( "x" ), (String)
r.get( "y" ) } ) );
}
```

The algorithm uses stacks to handle recursion, so the method stack will not blow up.

The following is not yet supported:

- List and Map unification
- Variables for the fields of facts
- Expression unification - pred(X, X + 1, X * Y / 7)

Report a bug

6.4. FORWARD-CHAINING

Forward-chaining is a production rule system. It is data-driven which means it reacts to the data it is presented. Facts are inserted into the working memory which results in one or more rules being true. They are then placed on the schedule to be executed by the agenda.

JBoss Rules is a forward-chaining engine.

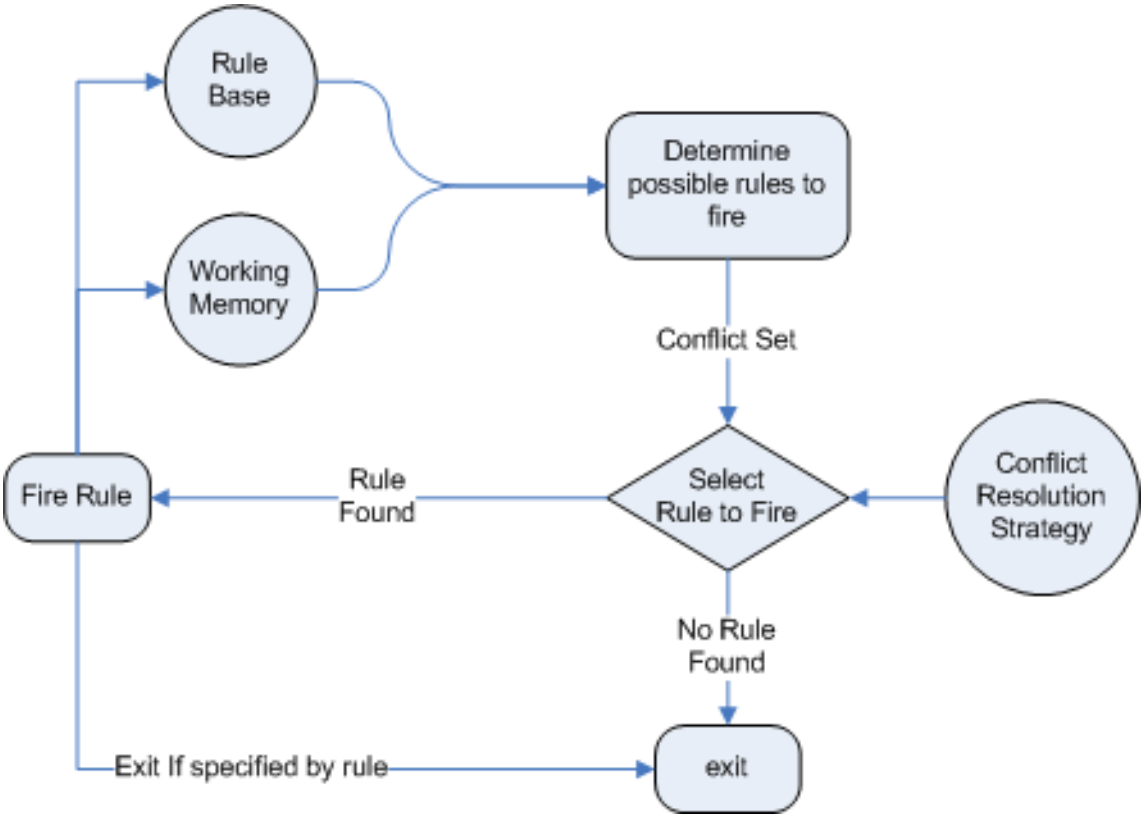


Figure 6.2. Forward Chaining Chart

Report a bug

6.5. BACKWARD-CHAINING

6.5.1. Backward-Chaining

A backward-chaining rule system is goal-driven. This means the system starts with a *conclusion* which the engine tries to satisfy. If it cannot do so it searches for sub-goals, that is, conclusions that will complete part of the current goal. It continues this process until either the initial conclusion is satisfied or there are no more unsatisfied sub-goals. **`Prolog`** is an example of a backward-chaining engine.

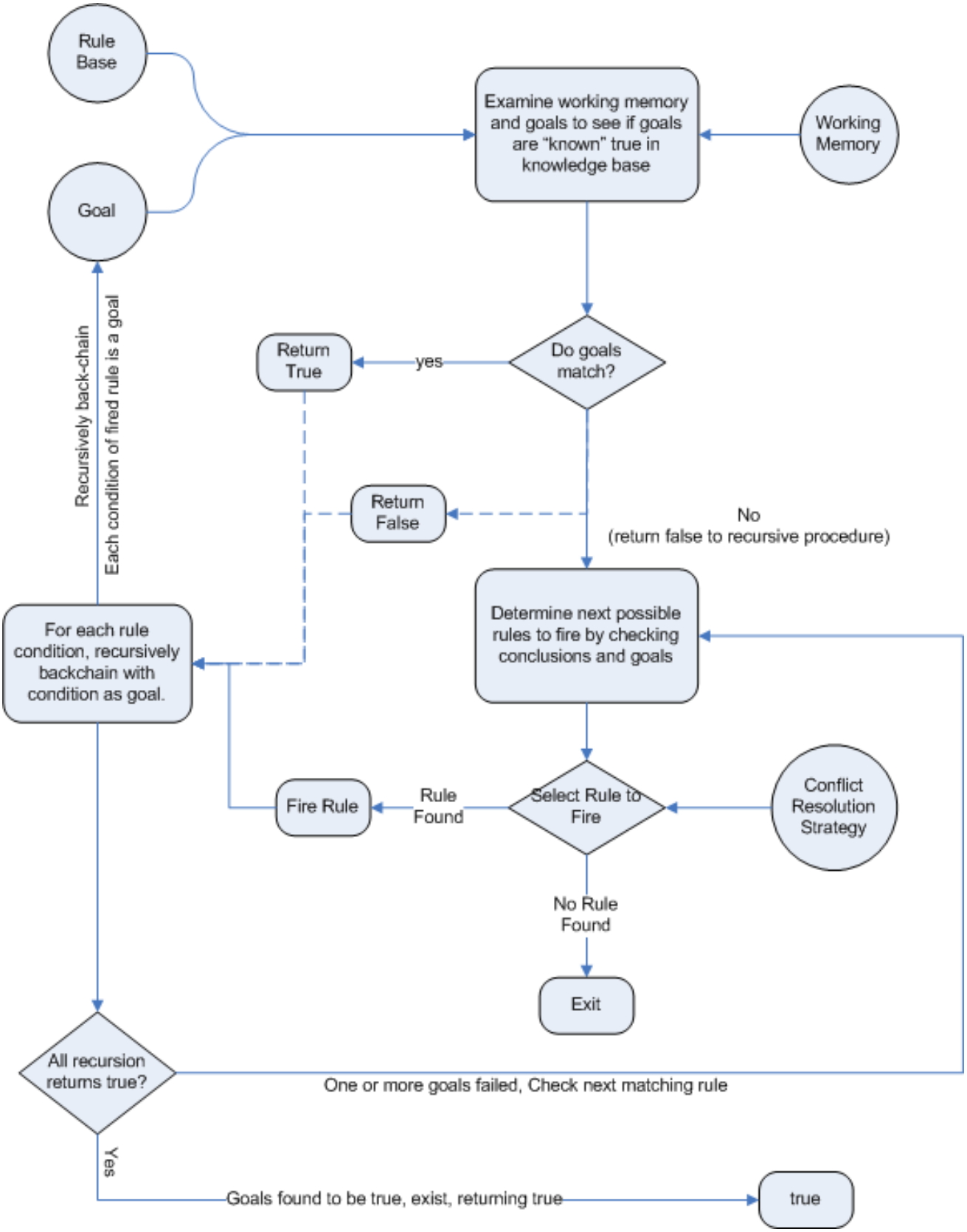


Figure 6.3. Backward Chaining Chart

IMPORTANT

Backward-chaining was implemented in JBoss BRMS 5.2.

Report a bug

6.5.2. Backward-Chaining Systems

Backward-Chaining is a feature recently added to the JBoss Rules Engine. This process is often referred to as derivation queries, and it is not as common compared to reactive systems since JBoss Rules is primarily reactive forward chaining. That is, it responds to changes in your data. The backward-chaining added to the engine is for product-like derivations.

Report a bug

6.5.3. Cloning Transitive Closures

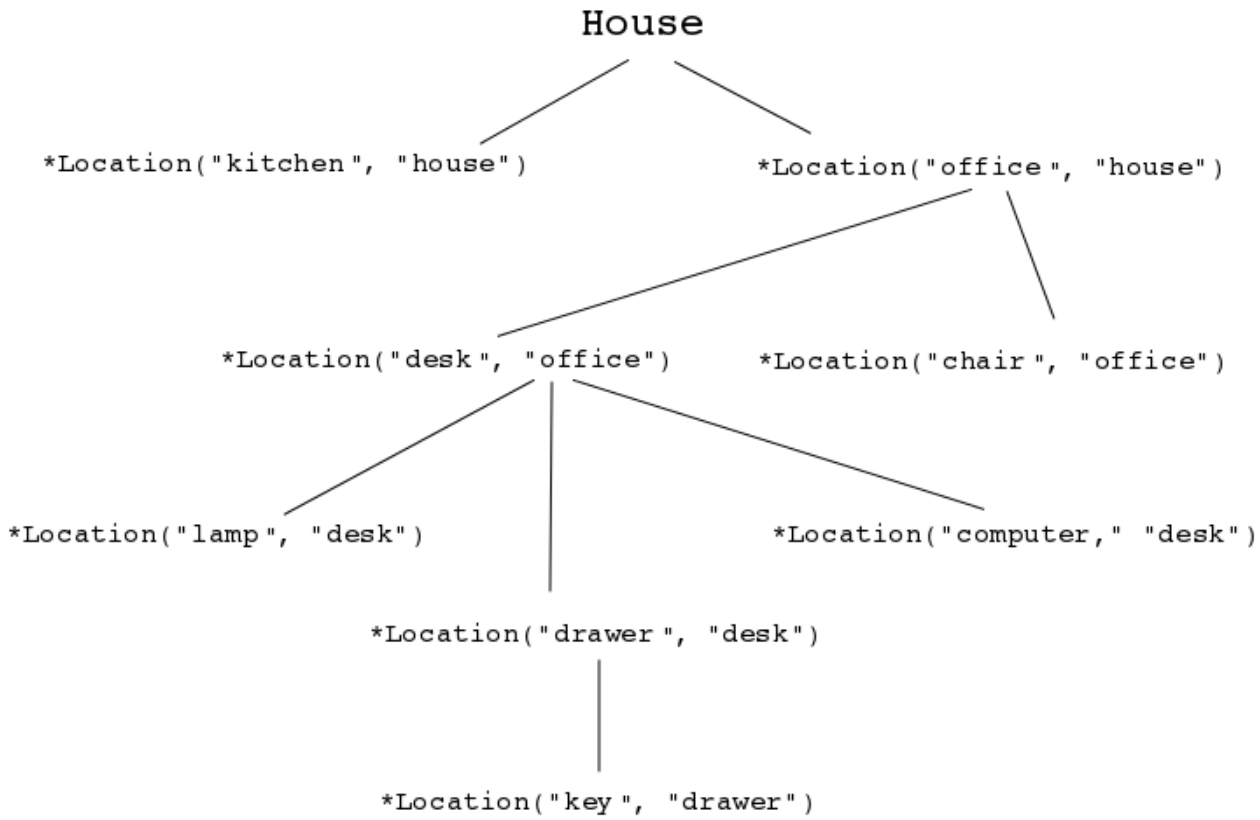


Figure 6.4. Reasoning Graph

The previous chart demonstrates a House example of transitive items. A similar reasoning chart can be created by implementing the following rules:

Procedure 6.1. Configure Transitive Closures

1. First, create some java rules to develop reasoning for transitive items. It inserts each of the locations.
2. Next, create the **`Location`** class; it has the item and where it is located.
3. Type the rules for the House example as depicted below:

```
ksession.insert( new Location("office", "house") );
ksession.insert( new Location("kitchen", "house") );
ksession.insert( new Location("knife", "kitchen") );
```

```
ksession.insert( new Location("cheese", "kitchen") );
ksession.insert( new Location("desk", "office") );
ksession.insert( new Location("chair", "office") );
ksession.insert( new Location("computer", "desk") );
ksession.insert( new Location("drawer", "desk") );
```

4. A transitive design is created in which the item is in its designated location such as a "desk" located in an "office."

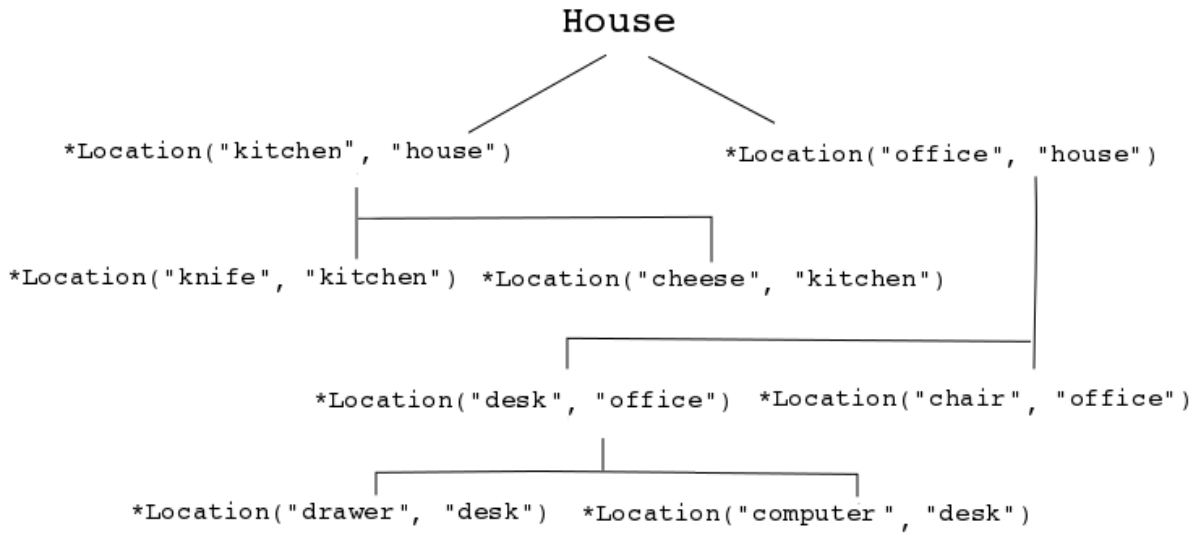


Figure 6.5. Transitive Reasoning Graph of a House.

NOTE

Notice compared to the previous graph, there is no **"key"** item in a **"drawer"** location. This will become evident in a later topic.

Report a bug

6.5.4. Defining a Query

Procedure 6.2. Define a Query

1. Create a query to look at the data inserted into the rules engine:

```
query isContainedIn( String x, String y )
  Location( x, y; )
  or
  ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

Notice how the query is recursive and is calling "isContainedIn."

2. Create a rule to print out every string inserted into the system to see how things are implemented. The rule should resemble the following format:

```
rule "go" salience 10
when
    $s : String( )
then
    System.out.println( $s );
end
```

3. Using Step 2 as a model, create a rule that calls upon the Step 1 query "isContainedIn."

```
rule "go1"
when
    String( this == "go1" )
    isContainedIn("office", "house"; )
then
    System.out.println( "office is in the house" );
end
```

The "go1" rule will fire when the first string is inserted into the engine. That is, it asks if the item "office" is in the location "house." Therefore, the Step 1 query is evoked by the previous rule when the "go1" String is inserted.

4. Create the "go1," insert it into the engine, and call the fireAllRules.

```
ksession.insert( "go1" );
ksession.fireAllRules();
---
go1
office is in the house
```

The --- line indicates the separation of the output of the engine from the firing of the "go" rule and the "go1" rule.

- "go1" is inserted
- Salience ensures it goes first
- The rule matches the query

Report a bug

6.5.5. Transitive Closure Example

Procedure 6.3. Create a Transitive Closure

1. Create a Transitive Closure by implementing the following rule:

```
rule "go2"
when
 String( this == "go2" )
 isContainedIn("drawer", "house"; )
then
 System.out.println( "Drawer in the House" );
end
```

2. Recall from the Cloning Transitive Closure's topic, there was no instance of "drawer" in "house." "drawer" was located in "desk."

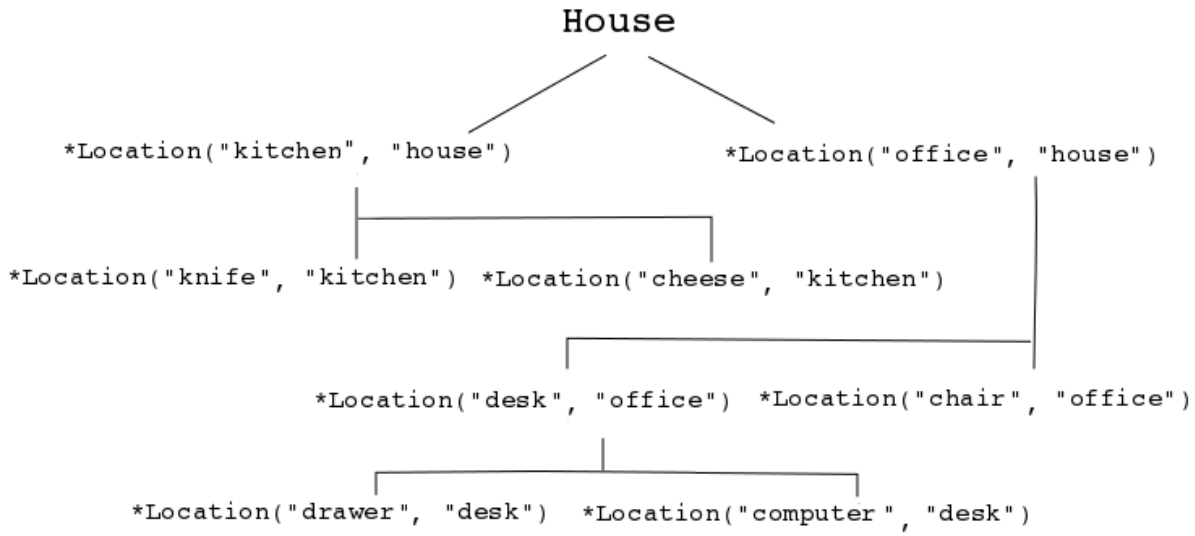


Figure 6.6. Transitive Reasoning Graph of a Drawer.

3. Use the previous query for this recursive information.

```
query isContainedIn( String x, String y )
 Location( x, y; )
 or
 ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

4. Create the "go2," insert it into the engine, and call the fireAllRules.

```
ksession.insert( "go2" );
ksession.fireAllRules();
---
go2
Drawer in the House
```

When the rule is fired, it correctly tells you "go2" has been inserted and that the "drawer" is in the "house."

5. Check how the engine determined this outcome.
   - The query has to recurse down several levels to determine this.
   - Instead of using **`Location( x, y; )`**, The query uses the value of **`(z, y; )`** since "drawer" is not in "house."
   - The **z** is currently unbound which means it has no value and will return everything that is in the argument.
   - **y** is currently bound to "house," so **z** will return "office" and "kitchen."
   - Information is gathered from "office" and checks recursively if the "drawer" is in the

"office." The following query line is being called for these parameters: **`isContainedIn (x ,z; )`**

There is no instance of "drawer" in "office;" therefore, it does not match. With **`z`** being unbound, it will return data that is within the "office," and it will gather that **`z == desk`**.

```
isContainedIn(x==drawer, z==desk)
```

`isContainedIn` recurses three times. On the final recurse, an instance triggers of "drawer" in the "desk."

```
Location(x==drawer, y==desk)
```

This matches on the first location and recurses back up, so we know that "drawer" is in the "desk," the "desk" is in the "office," and the "office" is in the "house;" therefore, the "drawer" is in the "house" and returns **`true`**.

Report a bug

6.5.6. Reactive Transitive Queries

Procedure 6.4. Create a Reactive Transitive Query

1. Create a Reactive Transitive Query by implementing the following rule:

```
rule "go3"
when
  String( this == "go3" )
  isContainedIn("key", "office"; )
then
  System.out.println( "Key in the Office" );
end
```

Reactive Transitive Queries can ask a question even if the answer can not be satisfied. Later, if it is satisfied, it will return an answer.

NOTE

Recall from the Cloning Transitive Closures example that there was no "key" item in the system.

2. Use the same query for this reactive information.

```
query isContainedIn( String x, String y )
  Location( x, y; )
  or
  ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

3. Create the "go3," insert it into the engine, and call the fireAllRules.

```
ksession.insert( "go3" );
```

```
ksession.fireAllRules();
---
go3
```

- "go3" is inserted
- `fireAllRules();` is called

The first rule that matches any String returns "go3" but nothing else is returned because there is no answer; however, while "go3" is inserted in the system, it will continuously wait until it is satisfied.

4. Insert a new location of "key" in the "drawer":

```
ksession.insert( new Location("key", "drawer") );
ksession.fireAllRules();
---
Key in the Office
```

This new location satisfies the transitive closure because it is monitoring the entire graph. In addition, this process now has four recursive levels in which it goes through to match and fire the rule.

Report a bug

6.5.7. Queries with Unbound Arguments

Procedure 6.5. Create an Unbound Argument's Query

1. Create a Query with Unbound Arguments by implementing the following rule:

```
rule "go4"
when
  String( this == "go4" )
  isContainedIn(thing, "office"; )
then
  System.out.println( "thing" + thing + "is in the Office"
);
end
```

This rule is asking for everything in the "office," and it will tell everything in all the rows below. The unbound argument (out variable **`thing`**) in this example will return every possible value; accordingly, it is very similar to the **z** value used in the Reactive Transitive Query example.

2. Use the query for the unbound arguments.

```
query isContainedIn( String x, String y )
  Location( x, y; )
  or
  ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

3. Create the "go4," insert it into the engine, and call the fireAllRules.

```
ksession.insert( "go4" );
ksession.fireAllRules();
---
go4
thing Key is in the Office
thing Computer is in the Office
thing Drawer is in the Office
thing Desk is in the Office
thing Chair is in the Office
```

When "go4" is inserted, it returns all the previous information that is transitively below "Office."

Report a bug

6.5.8. Multiple Unbound Arguments

Procedure 6.6. Creating Multiple Unbound Arguments

1. Create a query with Mulitple Unbound Arguments by implementing the following rule:

```
rule "go5"
when
  String( this == "go5" )
  isContainedIn(thing, location; )
then
  System.out.println( "thing" + thing + "is in" + location
);
end
```

Both **`thing`** and **`location`** are unbound out variables, and without bound arguments, everything is called upon.

2. Use the query for multiple unbound arguments.

```
query isContainedIn( String x, String y )
  Location( x, y; )
  or
  ( Location( z, y; ) and isContainedIn( x, z; ) )
end
```

3. Create the "go5," insert it into the engine, and call the fireAllRules.

```
ksession.insert( "go5" );
ksession.fireAllRules();
---
go5
thing Knife is in House
thing Cheese is in House
thing Key is in House
thing Computer is in House
thing Drawer is in House
thing Desk is in House
thing Chair is in House
```

```
thing Key is in Office
thing Computer is in Office
thing Drawer is in Office
thing Key is in Desk
thing Office is in House
thing Computer is in Desk
thing Knife is in Kitchen
thing Cheese is in Kitchen
thing Kitchen is in House
thing Key is in Drawer
thing Drawer is in Desk
thing Desk is in Office
thing Chair is in Office
```

When "go5" is called, it returns everything within everything.

Report a bug

6.6. DECISION TABLES

6.6.1. Decision Tables

Decision tables are a "precise yet compact" (ref. Wikipedia) way of representing conditional logic, and they are well suited to *business* level rules.

Red Hat JBOSS BRMS supports managing rules in a spreadsheet format. Supported formats are Excel (XLS) and CSV; accordingly, a variety of spreadsheet programs (such as Microsoft Excel) can be utilized.

In BRMS, decision tables are a way to generate rules driven from the data entered into a spreadsheet. All the usual features of a spreadsheet for data capture and manipulation can be taken advantage of.

When to Use Decision Tables
Consider decision tables as a course of action if rules exist that can be expressed as rule templates and data: each row of a decision table provides data that is combined with a template to generate a rule.

Many businesses already use spreadsheets for managing data, calculations, etc. With BRMS, you can also manage your business rules with spreadsheets. This assumes you are happy to manage packages of rules in `.xls` or `.csv` files. Decision tables are not recommended for rules that do not follow a set of templates or where there are a small number of rules. They are ideal in the sense that they can control what *parameters* of rules can be edited without exposing the rules directly.

Decision tables also provide a degree of insulation from the underlying object model.

Report a bug

6.6.2. Working with Decision Tables

It is important to remember that within a decision table, each row is a rule and each column in that row is either a condition or action for that rule. Ideally, rules are authored without regard for the order of rows; this makes maintenance easier as rows will not need to be shifted around all the time. As the rule engine processes the facts, any rules that match may fire.

Note that you can have multiple tables on one spreadsheet. Accordingly, rules can be grouped where they share common templates, yet they are all combined into one rule package. Decision tables are essentially a tool to generate DRL rules automatically.

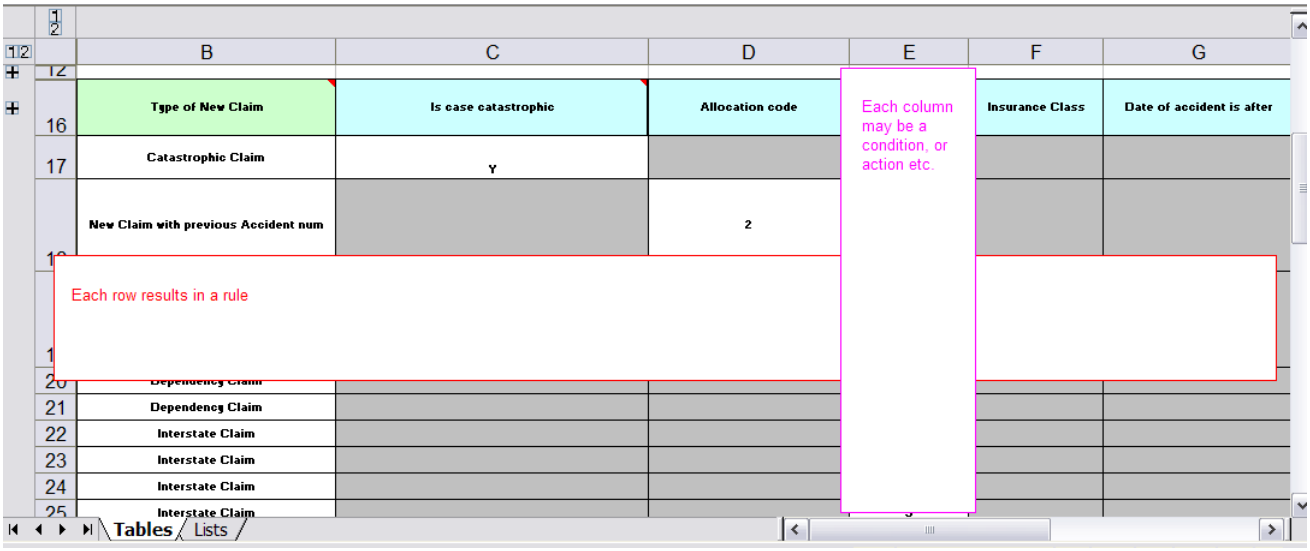


Figure 6.7. Decision Table Rows and Columns

The spreadsheet looks for the *RuleTable* keyword to indicate the start of a rule table (both the starting row and column). Other keywords are used to define other package level attributes (covered later). It is important to keep the keywords in one column. By convention the second column ("B") is used for this, but it can be any column (convention is to leave a margin on the left for notes). In the following diagram, C is actually the column where it starts by illustrating the conditions. Everything to the left of this is ignored. Note the keywords in column C.

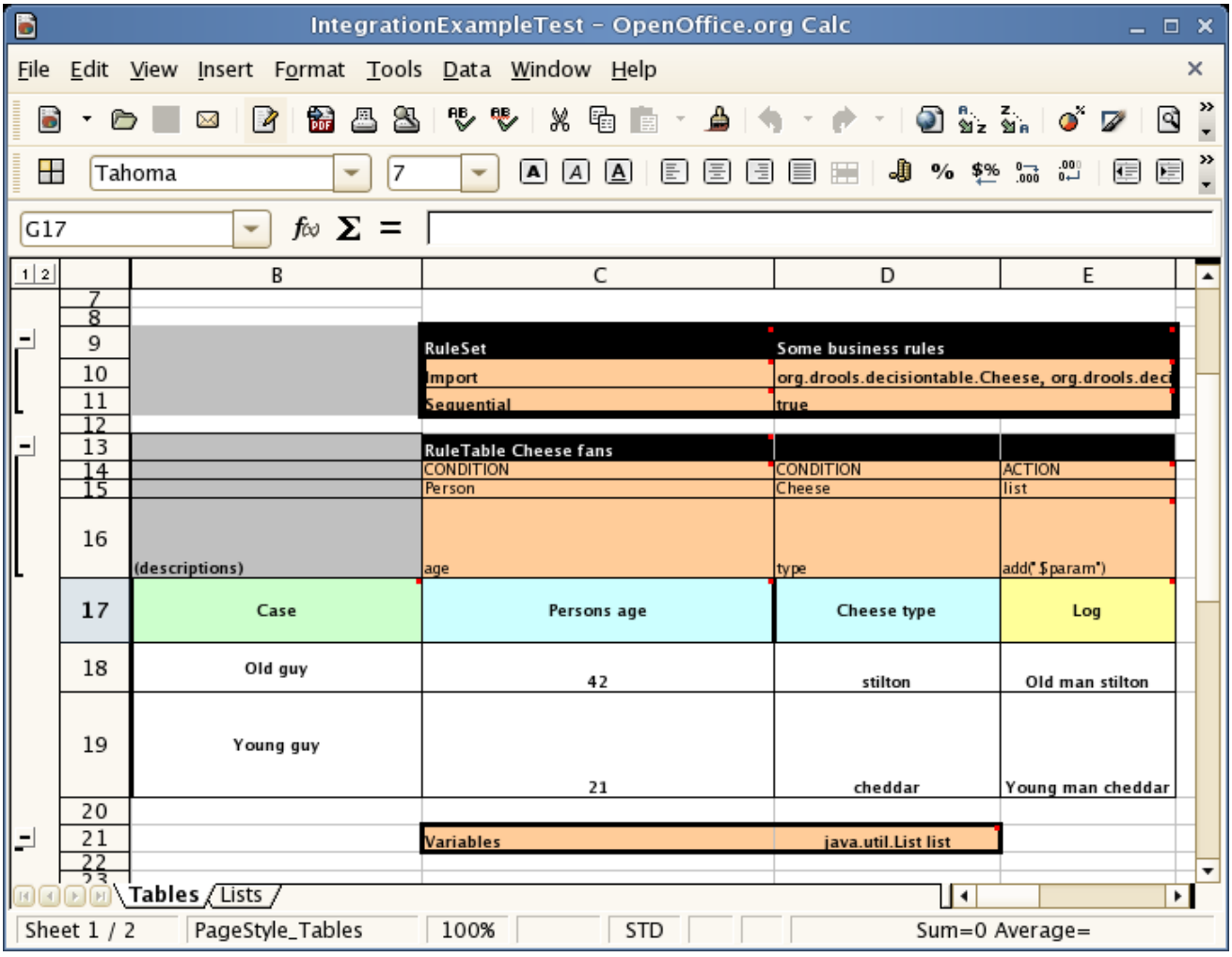


Figure 6.8. Expanded Rule Templates

In the previous example, the RuleSet keyword indicates the name to be used in the *rule package* that will encompass all the rules. This name is optional, using a default, but it *must* have the *RuleSet* keyword in the cell immediately to the right.

The other keywords visible in Column C are Import and Sequential, and they will be covered later. The RuleTable keyword indicates that a chunk of rules will follow, based on some rule templates. After the RuleTable keyword, there is a name used to prefix the names of the generated rules. The sheet name and row numbers are appended to guarantee unique rule names.

WARNING

The RuleTable name combined with the sheet name must be unique across all spreadsheet files in the same KieBase. If not, some rules might have the same name and only 1 of them will be applied. To show such ignored rules, raise the severity of such rule name conflicts Section 5.5.1, “Build Result Severity”.

The column of RuleTable indicates the column in which the rules start; columns to the left are ignored. In general, the keywords make up name-value pairs.

Referring to row 14 (the row immediately after RuleTable), the keywords CONDITION and ACTION indicate that the data in the columns below are for either the LHS or the RHS parts of a rule. There are other attributes on the rule which can also be optionally set this way.

Row 15 contains declarations of *ObjectTypes*. The content in this row is optional, but if this option is not in use, the row must be left blank. This option is usually found to be quite useful. When using this row, the values in the cells below (row 16) become constraints on that object type. In the above case, it generates **`Person(age=="42")`** and **`Cheese(type=="stilton")`**, where 42 and "stilton" come from row 18. In the above example, the "==" is implicit; if just a field name is given the translator assumes that it is to generate an exact match.

NOTE

An ObjectType declaration can span columns (via merged cells), meaning that all columns below the merged range are to be combined into one set of constraints within a single pattern matching a single fact at a time, as opposed to non-merged cells containing the same ObjectType, but resulting in different patterns, potentially matching different or identical facts.

Row 16 contains the rule templates themselves. They can use the "$param" placeholder to indicate where data from the cells below should be interpolated. (For multiple insertions, use "$1", "$2", etc., indicating parameters from a comma-separated list in a cell below.) Row 17 is ignored; it may contain textual descriptions of the column's purpose.

Rows 18 and 19 show data, which will be combined (interpolated) with the templates in row 15, to generate rules. If a cell contains no data, then its template is ignored. (This would mean that some condition or action does not apply for that rule row.) Rule rows are read until there is a blank row. Multiple RuleTables can exist in a sheet. Row 20 contains another keyword, and a value. The row positions of keywords like this do not matter (most people put them at the top) but their column should be the same one where the RuleTable or RuleSet keywords should appear. In our case column C has been chosen to be significant, but any other column could be used instead.

In the above example, rules would be rendered like the following (as it uses the "ObjectType" row):

```
//row 18
rule "Cheese_fans_18"
when
    Person(age=="42")
    Cheese(type=="stilton")
then
    list.add("Old man stilton");
end
```

NOTE

The constraints **`age=="42"`** and **`type=="stilton"`** are interpreted as single constraints, to be added to the respective ObjectType in the cell above. If the cells above were spanned, then there could be multiple constraints on one "column".

WARNING

Very large decision tables may have very large memory requirements.

Report a bug

6.6.3. Spreadsheet Structure

There are two types of rectangular areas defining data that is used for generating a DRL file. One, marked by a cell labelled **`RuleSet`**, defines all DRL items except rules. The other one may occur repeatedly and is to the right and below a cell whose contents begin with **`RuleTable`**. These areas represent the actual decision tables, each area resulting in a set of rules of similar structure.

A Rule Set area may contain cell pairs, one below the **`RuleSet`** cell and containing a keyword designating the kind of value contained in the other one that follows in the same row.

The columns of a Rule Table area define patterns and constraints for the left hand sides of the rules derived from it, actions for the consequences of the rules, and the values of individual rule attributes. Thus, a Rule Table area should contain one or more columns, both for conditions and actions, and an arbitrary selection of columns for rule attributes, at most one column for each of these. The first four rows following the row with the cell marked with **`RuleTable`** are earmarked as header area, mostly used for the definition of code to construct the rules. It is any additional row below these four header rows that spawns another rule, with its data providing for variations in the code defined in the Rule Table header.

All keywords are case insensitive.

Only the first worksheet is examined for decision tables.

Report a bug

6.6.4. Rule Set Entries

Entries in a Rule Set area may define DRL constructs (except rules) and specify rule attributes. While entries for constructs may be used repeatedly, each rule attribute may be given at most once. This applies to all rules unless it is overruled by the same attribute being defined within the Rule Table area.

Entries must be given in a vertically stacked sequence of cell pairs. The first one contains a keyword and the one to its right contains the value as shown in the table below. This sequence of cell pairs may be interrupted by blank rows or even a Rule Table as long as the column marked by **`RuleSet`** is upheld as the one containing the keyword.

Table 6.1. Entries in the Rule Set area

Keyword	Value	Usage
RuleSet	The package name for the generated DRL file. Optional, the default is **`rule_table`**.	Must be First entry.

Keyword	Value	Usage
Sequential	"true" or "false". If "true", then salience is used to ensure that rules fire from the top down.	Optional, at most once. If omitted, no firing order is imposed.
EscapeQuotes	"true" or "false". If "true", then quotation marks are escaped so that they appear literally in the DRL.	Optional, at most once. If omitted, quotation marks are escaped.
Import	A comma-separated list of Java classes to import.	Optional, may be used repeatedly.
Variables	Declarations of DRL globals, i.e., a type followed by a variable name. Multiple global definitions must be separated with a comma.	Optional, may be used repeatedly.
Functions	One or more function definitions, according to DRL syntax.	Optional, may be used repeatedly.
Queries	One or more query definitions, according to DRL syntax.	Optional, may be used repeatedly.
Declare	One or more declarative types, according to DRL syntax.	Optional, may be used repeatedly.

WARNING

In some locales, MS Office, LibreOffice and OpenOffice will encode a double quoth " differently, which will cause a compilation error. The difference is often hard to see. For example: **“A”** will fail, but **"A"** will work.

For defining rule attributes that apply to all rules in the generated DRL file, you can use any of the entries in the following table. Notice that the proper keyword must be used. Also, each of these attributes may be used only once.

Table 6.2. Rule attribute entries in the Rule Set area

Keyword	Initial	Value
PRIORITY	P	An integer defining the "salience" value for the rule. Overridden by the "Sequential" flag.

Keyword	Initial	Value
DURATION	D	A long integer value defining the "duration" value for the rule.
TIMER	T	A timer definition. See "Timers and Calendars".
ENABLED	B	A Boolean value. "true" enables the rule; "false" disables the rule.
CALENDARS	E	A calendars definition. See "Timers and Calendars".
NO-LOOP	U	A Boolean value. "true" inhibits looping of rules due to changes made by its consequence.
LOCK-ON-ACTIVE	L	A Boolean value. "true" inhibits additional activations of all rules with this flag set within the same ruleflow or agenda group.
AUTO-FOCUS	F	A Boolean value. "true" for a rule within an agenda group causes activations of the rule to automatically give the focus to the group.
ACTIVATION-GROUP	X	A string identifying an activation (or XOR) group. Only one rule within an activation group will fire, i.e., the first one to fire cancels any existing activations of other rules within the same group.
AGENDA-GROUP	G	A string identifying an agenda group, which has to be activated by giving it the "focus", which is one way of controlling the flow between groups of rules.
RULEFLOW-GROUP	R	A string identifying a rule-flow group.

Report a bug

6.6.5. Rule Tables

All Rule Tables begin with a cell containing "RuleTable". Optionally, they are followed by a string within the same cell. The string is used as the initial part of the name for all rules derived from this Rule Table

with the row number appended for distinction. (This automatic naming can be overridden by using a NAME column.) All other cells defining rules of this Rule Table are below and to the right of this cell.

The next row defines the column type, with each column resulting in a part of the condition or the consequence. In addition, they may provide some rule attribute, the rule name, or a comment. The table below shows which column headers are available; additional columns may be used according to the table showing rule attribute entries given in the preceding section. Note that each attribute column may be used at most once. For a column header, either use the keyword or any other word beginning with the letter given in the "Initial" column of these tables.

Table 6.3. Column Headers in the Rule Table

Keyword	Initial	Value	Usage
NAME	N	Provides the name for the rule generated from that row. The default is constructed from the text following the RuleTable tag and the row number.	At most one column
DESCRIPTION	I	A text, resulting in a comment within the generated rule.	At most one column
CONDITION	C	Code snippet and interpolated values for constructing a constraint within a pattern in a condition.	At least one per rule table
ACTION	A	Code snippet and interpolated values for constructing an action for the consequence of the rule.	At least one per rule table
METADATA	@	Code snippet and interpolated values for constructing a metadata entry for the rule.	Optional, any number of columns

Given a column headed CONDITION, the cells in successive lines result in a conditional element.

- Text in the first cell below CONDITION develops into a pattern for the rule condition, with the snippet in the next line becoming a constraint. If the cell is merged with one or more neighbours, a single pattern with multiple constraints is formed: all constraints are combined into a parenthesized list and appended to the text in this cell. The cell may be left blank, which means that the code snippet in the next row must result in a valid conditional element on its own.

  To include a pattern without constraints, you can write the pattern in front of the text for another pattern.

The pattern may be written with or without an empty pair of parentheses. A "from" clause may be appended to the pattern.

If the pattern ends with "eval", code snippets are supposed to produce boolean expressions for inclusion into a pair of parentheses after "eval".

- Text in the second cell below CONDITION is processed in two steps.

  1. The code snippet in this cell is modified by interpolating values from cells farther down in the column. If you want to create a constraint consisting of a comparison using "==" with the value from the cells below, the field selector alone is sufficient. Any other comparison operator must be specified as the last item within the snippet, and the value from the cells below is appended. For all other constraint forms, you must mark the position for including the contents of a cell with the symbol **`$param`**. Multiple insertions are possible by using the symbols **`$1`**, **`$2`**, etc., and a comma-separated list of values in the cells below.

     A text according to the pattern **`forall(`***delimiter***`){`***snippet***`}`** is expanded by repeating the *snippet* once for each of the values of the comma-separated list of values in each of the cells below, inserting the value in place of the symbol **`$`** and by joining these expansions by the given *delimiter*. Note that the forall construct may be surrounded by other text.

  2. If the cell in the preceding row is not empty, the completed code snippet is added to the conditional element from that cell. A pair of parentheses is provided automatically, as well as a separating comma if multiple constraints are added to a pattern in a merged cell.

     If the cell above is empty, the interpolated result is used as is.

- Text in the third cell below CONDITION is for documentation only. It should be used to indicate the column's purpose to a human reader.

- From the fourth row on, non-blank entries provide data for interpolation as described above. A blank cell results in the omission of the conditional element or constraint for this rule.

Given a column headed ACTION, the cells in successive lines result in an action statement.

- Text in the first cell below ACTION is optional. If present, it is interpreted as an object reference.

- Text in the second cell below ACTION is processed in two steps.

  1. The code snippet in this cell is modified by interpolating values from cells farther down in the column. For a singular insertion, mark the position for including the contents of a cell with the symbol **`$param`**. Multiple insertions are possible by using the symbols **`$1`**, **`$2`**, etc., and a comma-separated list of values in the cells below.

     A method call without interpolation can be achieved by a text without any marker symbols. In this case, use any non-blank entry in a row below to include the statement.

     The forall construct is available here, too.

  2. If the first cell is not empty, its text, followed by a period, the text in the second cell and a terminating semicolon are stringed together, resulting in a method call which is added as an action statement for the consequence.

     If the cell above is empty, the interpolated result is used as is.

- Text in the third cell below ACTION is for documentation only. It should be used to indicate the column's purpose to a human reader.
- From the fourth row on, non-blank entries provide data for interpolation as described above. A blank cell results in the omission of the action statement for this rule.

NOTE

Using `$1` instead of `$param` works in most cases, but it will fail if the replacement text contains a comma: then, only the part preceding the first comma is inserted. Use this "abbreviation" judiciously.

Given a column headed METADATA, the cells in successive lines result in a metadata annotation for the generated rules.

- Text in the first cell below METADATA is ignored.
- Text in the second cell below METADATA is subject to interpolation, as described above, using values from the cells in the rule rows. The metadata marker character @ is prefixed automatically, and thus it should not be included in the text for this cell.
- Text in the third cell below METADATA is for documentation only. It should be used to indicate the column's purpose to a human reader.
- From the fourth row on, non-blank entries provide data for interpolation as described above. A blank cell results in the omission of the metadata annotation for this rule.

Report a bug

6.6.6. Decision Table Examples

The various interpolations are illustrated in the following example.

> **Example 6.4. Interpolating cell data**
>
> If the template is `Foo(bar == $param)` and the cell is `42`, then the result is `Foo(bar == 42)`.
>
> If the template is `Foo(bar < $1, baz == $2)` and the cell contains `42,43`, the result will be `Foo(bar < 42, baz ==43)`.
>
> The template `forall(&&){bar != $}` with a cell containing `42,43` results in `bar != 42 && bar != 43`.

The next example demonstrates the joint effect of a cell defining the pattern type and the code snippet below it.

13	RuleTable Cheese fans	
14	CONDITION	CONDITION
15	Person	
16	age	type
17	Persons age	Cheese type
18	42	stilton
19	21	cheddar

Figure 6.9. Spanned Column

This spreadsheet section shows how the `Person` type declaration spans 2 columns, and thus both constraints will appear as `Person(age == ..., type == ...)`. Since only the field names are present in the snippet, they imply an equality test.

In the following example the marker symbol `$param` is used:

CONDITION
Person
age==“$param”
Persons age
42

Figure 6.10. Condition Parameter

The result of this column is the pattern `Person(age == "42"))`. You may have noticed that the marker and the operator "==" are redundant.

The next example illustrates that a trailing insertion marker can be omitted:

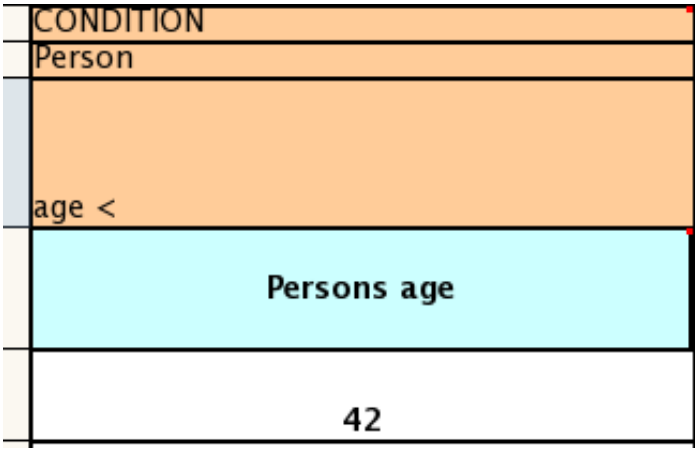


Figure 6.11. Operator Paremeter

Here, appending the value from the cell is implied, resulting in `Person(age < "42"))`.

You can provide the definition of a binding variable, as in the example below:

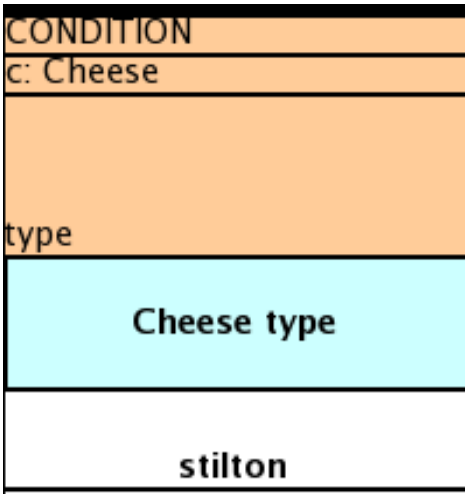


Figure 6.12. Bindnig Variables

Here, the result is `c: Cheese(type == "stilton")`. Note that the quotes are provided automatically. Actually, anything can be placed in the object type row. Apart from the definition of a binding variable, it could also be an additional pattern that is to be inserted literally.

A simple construction of an action statement with the insertion of a single value is shown below:

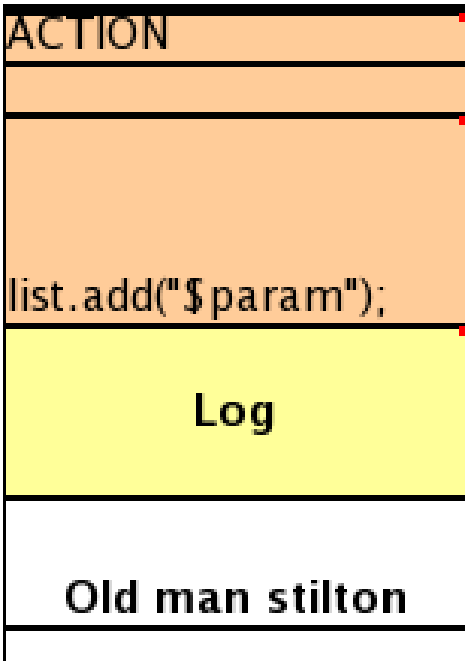
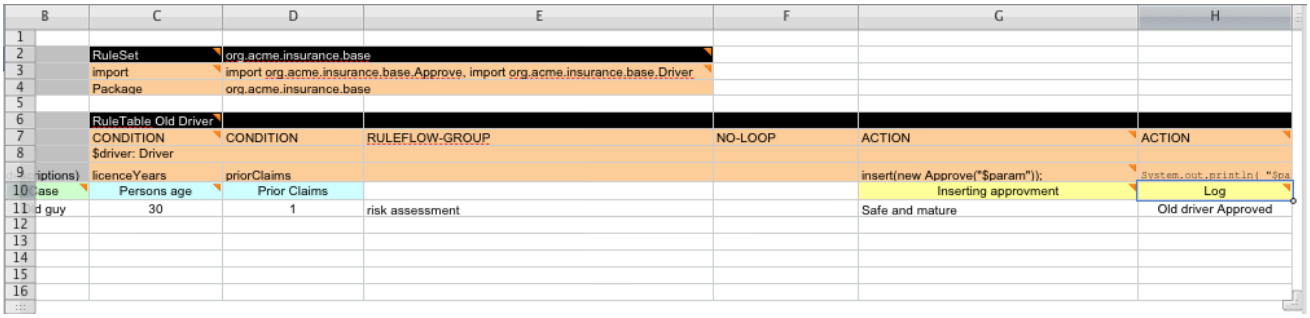


Figure 6.13. Action Consequence

The cell below the ACTION header is left blank. Using this style, anything can be placed in the consequence, not just a single method call. (The same technique is applicable within a CONDITION column as well.)

Below is a comprehensive example, showing the use of various column headers. It is not an error to have no value below a column header (as in the NO-LOOP column): here, the attribute will not be applied in any of the rules.



	B	C	D	E	F	G	H
1							
2		RuleSet	org.acme.insurance.base				
3		import	import org.acme.insurance.base.Approve, import org.acme.insurance.base.Driver				
4		Package	org.acme.insurance.base				
5							
6		RuleTable Old Driver					
7		CONDITION	CONDITION	RULEFLOW-GROUP	NO-LOOP	ACTION	ACTION
8		$driver: Driver					
9	iptions)	licenceYears	priorClaims			insert(new Approve("$param"));	System.out.println("$pa
10	ase	Persons age	Prior Claims			Inserting approvment	Log
11	d guy	30	1	risk assessment		Safe and mature	Old driver Approved
12							
13							
14							
15							
16							

Figure 6.14. Keyword's Example

And, finally, here is an example of Import, Variables and Functions.

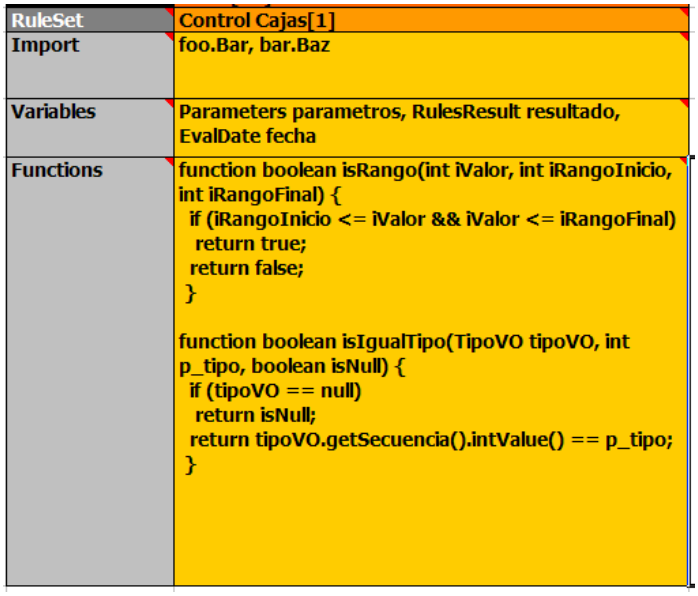


Figure 6.15. RuleSet Variable Keywords

Multiple package names within the same cell must be separated by a comma. Also, the pairs of type and variable names must be comma-separated. Functions, however, must be written as they appear in a DRL file. This should appear in the same column as the "RuleSet" keyword; it could be above, between or below all the rule rows.

NOTE

It may be more convenient to use Import, Variables, Functions and Queries repeatedly rather than packing several definitions into a single cell.

Report a bug

6.6.7. Spreadsheet based Decision Tables

The API to use spreadsheet based decision tables is in the drools-decisiontables module. There is really only one class to look at: `SpreadsheetCompiler`. This class will take spreadsheets in various formats and generate rules in DRL (which you can then use in the normal way). The `SpreadsheetCompiler` can be used to generate partial rule files and assemble them into a complete rule package after the fact (this allows the separation of technical and non-technical aspects of the rules if needed).

To get started, a sample spreadsheet can be used as a base. Alternatively, if the plug-in is being used (Rule Workbench IDE), the wizard can generate a spreadsheet from a template (to edit it an xls compatible spreadsheet editor will need to be used).

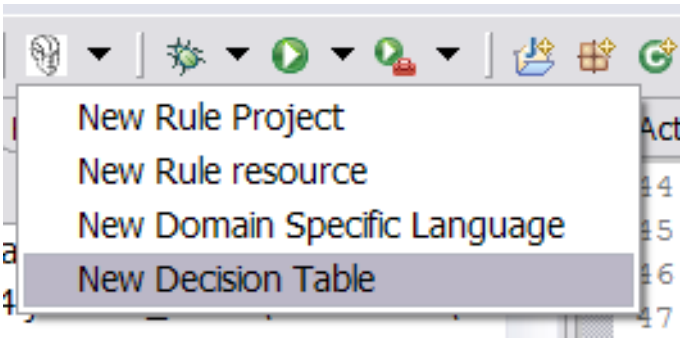


Figure 6.16. Spreadsheet Editor

Report a bug

6.6.8. Business Rules in Decision Tables

Decision tables lend themselves to close collaboration between IT and domain experts. While making the business rules clear to business analysts, decision tables provide an ideal separation of concerns.

Typically, the whole process of authoring rules (coming up with a new decision table) would be something like:

1. Business analyst takes a template decision table (from a repository, or from IT)
2. Decision table business language descriptions are entered in the table(s)
3. Decision table rules (rows) are entered (roughly)
4. Decision table is handed to a technical resource, who maps the business language (descriptions) to scripts (this may involve software development of course, if it is a new application or data model)
5. Technical person hands back and reviews the modifications with the business analyst.
6. The business analyst can continue editing the rule rows as needed (moving columns around is also fine etc).
7. In parallel, the technical person can develop test cases for the rules (liaising with business analysts) as these test cases can be used to verify rules and rule changes once the system is running.

Using spreadsheet features

Features of applications like Excel can be used to provide assistance in entering data into spreadsheets, such as validating fields. Lists that are stored in other worksheets can be used to provide valid lists of values for cells, like in the following diagram.

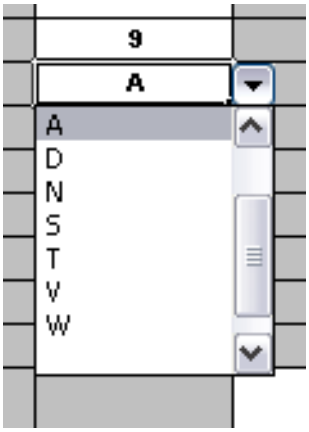


Figure 6.17. Spreadsheet List

Some applications provide a limited ability to keep a history of changes, but it is recommended to use an alternative means of revision control. When changes are being made to rules over time, older versions are archived (many open source solutions exist for this, such as Subversion or Git).

Report a bug

6.6.9. Rule Templates

Rule Templates are similar to decision tables, but they do not necessarily require a spreadsheet. They use any tabular data source as a source of rule data by populating a template to generate many rules. This can allow for more flexible spreadsheets and more flexible rules in existing databases.

With Rule Templates, the data is separated from the rule, and there are no restrictions on which part of the rule is data-driven. So whilst you can do everything you could do in decision tables, you can also do the following:

- store your data in a database (or any other format)
- conditionally generate rules based on the values in the data
- use data for any part of your rules (e.g. condition operator, class name, property name)
- run different templates over the same data

As an example, a more classic decision table is shown, but without any hidden rows for the rule meta data (so the spreadsheet only contains the raw data to generate the rules).

Case	Persons age	Cheese type	Log
Old guy	42	stilton	Old man stilton
Young guy	21	cheddar	Young man cheddar

Figure 6.18. Template Table

If this were a regular decision table, there would be hidden rows before row 1 and between rows 1 and 2 containing rule metadata. With rule templates, the data is completely separate from the rules. This has two handy consequences - you can apply multiple rule templates to the same data and your data is not

tied to your rules at all. So what does the template look like?

```
1 template header
2 age
3 type
4 log
5
6 package org.drools.examples.templates;
7
8 global java.util.List list;
9
10 template "cheesefans"
11
12 rule "Cheese fans_@{row.rowNumber}"
13 when
14 Person(age == @{age})
15 Cheese(type == "@{type}")
16 then
17 list.add("@{log}");
18 end
19
20 end template
```

Below are annotations to the preceding program listing:

- Line 1: All rule templates start with **`template header`**.
- Lines 2-4: Following the header is the list of columns in the order they appear in the data. In this case we are calling the first column **age**, the second **`type`** and the third **`log`**.
- Line 5: An empty line signifies the end of the column definitions.
- Lines 6-9: Standard rule header text. This is standard rule DRL and will appear at the top of the generated DRL. Put the package statement and any imports and global and function definitions into this section.
- Line 10: The keyword **`template`** signals the start of a rule template. There can be more than one template in a template file, but each template should have a unique name.
- Lines 11-18: The rule template - see below for details.
- Line 20: The keywords **`end template`** signify the end of the template.

The rule templates rely on MVEL to do substitution using the syntax @{token_name}. There is currently one built-in expression, @{row.rowNumber} which gives a unique number for each row of data and enables you to generate unique rule names. For each row of data, a rule will be generated with the values in the data substituted for the tokens in the template. With the example data above, the following rule file would be generated:

```
package org.drools.examples.templates;

global java.util.List list;

rule "Cheese fans_1"
when
  Person(age == 42)
```

```
  Cheese(type == "stilton")
then
  list.add("Old man stilton");
end

rule "Cheese fans_2"
when
  Person(age == 21)
  Cheese(type == "cheddar")
then
  list.add("Young man cheddar");
end
```

The code to run this is depicted below:

```
DecisionTableConfiguration dtableconfiguration =
org.kie.internal.builder.KnowledgeBuilderFactory.newDecisionTableConfigura
tion();

dtableconfiguration.setInputType( DecisionTableInputType.XLS );

Resource dt = ResourceFactory.newClassPathResource( getSpreadsheetName(),
getClass() )
 .setConfiguration( dtableconfiguration );
```

Report a bug

CHAPTER 7. RULE LANGUAGES

7.1. RULE OVERVIEW

7.1.1. Overview

JBoss Rules has a native rule language. This format is very light in terms of punctuation, and supports natural and domain specific languages via "expanders" that allow the language to morph to your problem domain.

Report a bug

7.1.2. A rule file

A rule file is typically a file with a .drl extension. In a DRL file you can have multiple rules, queries and functions, as well as some resource declarations like imports, globals, and attributes that are assigned and used by your rules and queries. However, you are also able to spread your rules across multiple rule files (in that case, the extension .rule is suggested, but not required) - spreading rules across files can help with managing large numbers of rules. A DRL file is simply a text file.

Report a bug

7.1.3. The structure of a rule file

The overall structure of a rule file is the following:

Example 7.1. Rules file

```
package package-name

imports

globals

functions

queries

rules
```

The order in which the elements are declared is not important, except for the package name that, if declared, must be the first element in the rules file. All elements are optional, so you will use only those you need.

Report a bug

7.1.4. What is a rule

For the inpatients, just as an early view, a rule has the following rough structure:

```
rule "name"
    attributes
```

```
    when
        LHS
    then
        RHS
end
```

Mostly punctuation is not needed, even the double quotes for "name" are optional, as are newlines. Attributes are simple (always optional) hints to how the rule should behave. LHS is the conditional parts of the rule, which follows a certain syntax which is covered below. RHS is basically a block that allows dialect specific semantic code to be executed.

It is important to note that white space is not important, *except* in the case of domain specific languages, where lines are processed one by one and spaces may be significant to the domain language.

Report a bug

7.2. RULE LANGUAGE KEYWORDS

7.2.1. Hard Keywords

Hard keywords are words which you cannot use when naming your domain objects, properties, methods, functions and other elements that are used in the rule text. The hard keywords are `true`, `false`, and `null`.

Report a bug

7.2.2. Soft Keywords

Soft keywords can be used for naming domain objects, properties, methods, functions and other elements. The rules engine recognizes their context and processes them accordingly.

Report a bug

7.2.3. List of Soft Keywords

- `lock-on-active`
- `date-effective`
- `date-expires`
- `no-loop`
- `auto-focus`
- `activation-group`
- `agenda-group`
- `ruleflow-group`
- `entry-point`

- **duration**
- **package**
- **import**
- **dialect**
- **salience**
- **enabled**
- **attributes**
- **rule**
- **extend**
- when
- then
- **template**
- **query**
- **declare**
- **function**
- **global**
- **eval**
- **not**
- **in**
- **or**
- **and**
- **exists**
- **forall**
- accumulate
- collect
- from
- **action**
- **reverse**
- **result**

- **end**
- over
- **init**

Report a bug

7.3. RULE LANGUAGE COMMENTS

7.3.1. Comments

Comments are sections of text that are ignored by the rule engine. They are stripped out when they are encountered, except inside semantic code blocks (like a rule's RHS).

Report a bug

7.3.2. Single Line Comment Example

This is what a single line comment looks like. To create single line comments, you can use '//'. The parser will ignore anything in the line after the comment symbol:

```
rule "Testing Comments"
when
    // this is a single line comment
    eval( true ) // this is a comment in the same line of a pattern
then
    // this is a comment inside a semantic code block
end
```

Report a bug

7.3.3. Multi-Line Comment Example

This is what a multi-line comment looks like. This configuration comments out blocks of text, both in and outside semantic code blocks:

```
rule "Test Multi-line Comments"
when
    /* this is a multi-line comment
       in the left hand side of a rule */
    eval( true )
then
    /* and this is a multi-line comment
       in the right hand side of a rule */
end
```

Report a bug

7.4. RULE LANGUAGE MESSAGES

7.4.1. Error Messages

JBoss Rules introduces standardized error messages. This standardization aims to help users to find and resolve problems in a easier and faster way.

Report a bug

7.4.2. Error Message Format

This is the standard error message format.

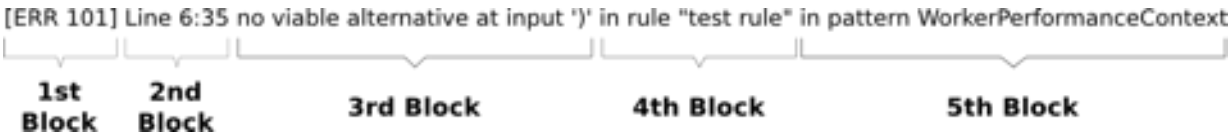


Figure 7.1. Error Message Format Example

1st Block: This area identifies the error code.

2nd Block: Line and column information.

3rd Block: Some text describing the problem.

4th Block: This is the first context. Usually indicates the rule, function, template or query where the error occurred. This block is not mandatory.

5th Block: Identifies the pattern where the error occurred. This block is not mandatory.

Report a bug

7.4.3. Error Messages Description

Table 7.1. Error Messages

Error Message	Description	Example	

[ERR 101] Line 4:4 no viable alternative at input 'exits' in rule one	Indicates when the parser came to a decision point but couldn't identify an alternative.	`1: rule one`
		`2:  when`
		`3:  exists Foo()`
		`4:  exits Bar()`
		`5:  then`
		`6: end`	

[ERR 101] Line 3:2 no viable alternative at input 'WHEN'	This message means the parser has encountered the token **WHEN** (a hard keyword) which is in the wrong place, since the rule name is missing.	`1: package org.drools;`
		`2: rule`
		`3:  when`
		`4:  Object()`
		`5:  then`
		`6:  System.out.println("A RHS");`
		`7: end`	

Error Message	Description	Example	

[ERR 101] Line 0:-1 no viable alternative at input '<eof>' in rule simple_rule in pattern [name]	Indicates an open quote, apostrophe or parentheses.	`1: rule simple_rule`
		`2:   when`
		`3:     Student( name == "Andy )`
		`4:   then`
		`5: end`	

[ERR 102] Line 0:-1 mismatched input '<eof>' expecting ')' in rule simple_rule in pattern Bar	Indicates that the parser was looking for a particular symbol that it didn't end at the current input position.	`1: rule simple_rule`
		`2:   when`
		`3:     foo3 : Bar(`	

[ERR 102] Line 0:-1 mismatched input '<eof>' expecting ')' in rule simple_rule in pattern [name]	This error is the result of an incomplete rule statement. Usually when you get a 0:-1 position, it means that parser reached the end of source. To fix this problem, it is necessary to complete the rule statement.	`1: package org.drools;`
		`2:`
		`3: rule "Avoid NPE on wrong syntax"`
		`4:   when`
		`5:     not( Cheese( ( type == "stilton", price == 10 ) || ( type == "brie", price == 15 ) ) from $cheeseList )`
		`6:   then`
		`7:     System.out.println("OK");`
		`8: end`	

[ERR 103] Line 7:0 rule 'rule_key' failed predicate: {(validateIdentifierKey(DroolsSoftKeywords.RULE))}? in rule	A validating semantic predicate evaluated to false. Usually these semantic predicates are used to identify soft keywords.	`1: package nesting;`
		` 2: dialect "mvel"`
		` 3:`
		` 4: import org.drools.Person`
		` 5: import org.drools.Address`
		` 6:`
		` 7: fdsfdsfds`
		` 8:`
		` 9: rule "test something"`
		`10:   when`
		`11:     p: Person( name=="Michael" )`
		`12:   then`
		`13:     p.name = "other";`
		`14:`
		`System.out.println(p.name);`
		`15: end`	

Error Message	Description	Example	

[ERR 104] Line 3:4 trailing semi-colon not allowed in rule simple_rule	This error is associated with the **eval** clause, where its expression may not be terminated with a semicolon. This problem is simple to fix: just remove the semi-colon.	`1: rule simple_rule`
		`2:  when`
		`3:  eval(abc();)`
		`4:  then`
		`5: end`	

[ERR 105] Line 2:2 required (...)+ loop did not match anything at input 'aa' in template test_error	The recognizer came to a subrule in the grammar that must match an alternative at least once, but the subrule did not match anything. To fix this problem it is necessary to remove the numeric value as it is neither a valid data type which might begin a new template slot nor a possible start for any other rule file construct.	`1: template test_error`
		`2:  aa s  11;`
		`3: end`	

Report a bug

7.4.4. Package

A *package* is a collection of rules and other related constructs, such as imports and globals. The package members are typically related to each other, such as HR rules. A package represents a namespace, which ideally is kept unique for a given grouping of rules. The package name itself is the namespace, and is not related to files or folders in any way.

It is possible to assemble rules from multiple rule sources, and have one top level package configuration that all the rules are kept under (when the rules are assembled). It is not possible to merge into the same package resources declared under different names. A single Rulebase may, however, contain multiple packages built on it. A common structure is to have all the rules for a package in the same file as the package declaration (so that is it entirely self-contained).

Report a bug

7.4.5. Import Statements

Import statements work like import statements in Java. You need to specify the fully qualified paths and type names for any objects you want to use in the rules. JBoss Rules automatically imports classes from the Java package of the same name, and also from the package `java.lang`.

Report a bug

7.4.6. Using Globals

In order to use globals you must:

1. Declare the global variable in the rules file and use it in rules. Example:

```
global java.util.List myGlobalList;

rule "Using a global"
when
    eval( true )
then
    myGlobalList.add( "Hello World" );
end
```

2. Set the global value on the working memory. It is best practice to set all global values before asserting any fact to the working memory. Example:

```
List list = new ArrayList();
WorkingMemory wm = rulebase.newStatefulSession();
wm.setGlobal( "myGlobalList", list );
```

Report a bug

7.4.7. The From Element

The *from* element allows you to pass a Hibernate session as a global. It also lets you pull data from a named Hibernate query.

Report a bug

7.4.8. Using Globals with an e-Mail Service

Procedure 7.1. Task

1. Open the integration code that is calling the rule engine.
2. Obtain your emailService object and then set it in the working memory.
3. In the DRL, declare that you have a global of type emailService and give it the name "email".
4. In your rule consequences, you can use things like email.sendSMS(number, message).

WARNING

Globals are not designed to share data between rules and they should never be used for that purpose. Rules always reason and react to the working memory state, so if you want to pass data from rule to rule, assert the data as facts into the working memory.

IMPORTANT

Do not set or change a global value from inside the rules. We recommend to you always set the value from your application using the working memory interface.

Report a bug

7.5. DOMAIN SPECIFIC LANGUAGES (DSLS)

7.5.1. Domain Specific Languages

Domain Specific Languages (or DSLs) are a way of creating a rule language that is dedicated to your problem domain. A set of DSL definitions consists of transformations from DSL "sentences" to DRL constructs, which lets you use of all the underlying rule language and engine features. You can write rules in DSL rule (or DSLR) files, which will be translated into DRL files.

DSL and DSLR files are plain text files and you can use any text editor to create and modify them. There are also DSL and DSLR editors you can use, both in the IDE as well as in the web based BRMS, although they may not provide you with the full DSL functionality.

Report a bug

7.5.2. Using DSLs

DSLs can serve as a layer of separation between rule authoring (and rule authors) and the technical intricacies resulting from the modeling of domain object and the rule engine's native language and methods. A DSL hides implementation details and focuses on the rule logic proper. DSL sentences can also act as "templates" for conditional elements and consequence actions that are used repeatedly in your rules, possibly with minor variations. You may define DSL sentences as being mapped to these repeated phrases, with parameters providing a means for accommodating those variations.

Report a bug

7.5.3. DSL Example

Table 7.2. DSL Example

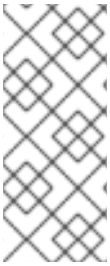
Example	Description
`[when]Something is {colour}=Something(colour=="{colour}")`	**[when]** indicates the scope of the expression (that is, whether it is valid for the LHS or the RHS of a rule). The part after the bracketed keyword is the expression that you use in the rule. The part to the right of the equal sign ("=") is the mapping of the expression into the rule language. The form of this string depends on its destination, RHS or LHS. If it is for the LHS, then it ought to be a term according to the regular LHS syntax; if it is for the RHS then it might be a Java statement.

Report a bug

7.5.4. How the DSL Parser Works

Whenever the DSL parser matches a line from the rule file written in the DSL with an expression in the DSL definition, it performs three steps of string manipulation:

- The DSL extracts the string values appearing where the expression contains variable names in brackets.
- The values obtained from these captures are interpolated wherever that name occurs on the right hand side of the mapping.
- The interpolated string replaces whatever was matched by the entire expression in the line of the DSL rule file.



NOTE

You can use (for instance) a '?' to indicate that the preceding character is optional. One good reason to use this is to overcome variations in natural language phrases of your DSL. But, given that these expressions are regular expression patterns, this means that all wildcard characters in Java's pattern syntax have to be escaped with a preceding backslash ('\').

Report a bug

7.5.5. The DSL Compiler

The DSL compiler transforms DSL rule files line by line. If you do not wish for this to occur, ensure that the captures are surrounded by characteristic text (words or single characters). As a result, the matching operation done by the parser plucks out a substring from somewhere within the line. In the example below, quotes are used as distinctive characters. (The characters that surround the capture are not included during interpolation, just the contents between them.)

Report a bug

7.5.6. DSL Syntax Examples

Table 7.3. DSL Syntax Examples

Name	Description	Example
Quotes	Use quotes for textual data that a rule editor may want to enter. You can also enclose the capture with words to ensure that the text is correctly matched.	`[when]something is "{color}"=Something(color=="{color}")` `[when]another {state} thing=OtherThing(state=="{state}"`
Braces	In a DSL mapping, the braces "{" and "}" should only be used to enclose a variable definition or reference, resulting in a capture. If they should occur literally, either in the expression or within the replacement text on the right hand side, they must be escaped with a preceding backslash ("\").	`[then]do something= if (foo) \{ doSomething(); \}`

Mapping with correct syntax example	n/a	`# This is a comment to be ignored.`
		`[when]There is a person with name of "{name}"=Person(name=="{name}")`
		`[when]Person is at least {age} years old and lives in "{location}"= Person(age >= {age}, location=="{location}")`
		`[then]Log "{message}"=System.out.println("{message}");`
		`[when]And = and`

Name	Description	Example
Expanded DSL example	n/a	`There is a person with name of "Kitty" ==> Person(name="Kitty") Person is at least 42 years old and lives in "Atlanta" ==> Person(age >= 42, location="Atlanta") Log "boo" ==> System.out.println("boo"); There is a person with name of "Bob" and Person is at least 30 years old and lives in "Utah" ==> Person(name="Bob") and Person(age >= 30, location="Utah")`

NOTE

If you are capturing plain text from a DSL rule line and want to use it as a string literal in the expansion, you must provide the quotes on the right hand side of the mapping.

Report a bug

7.5.7. Chaining DSL Expressions

DSL expressions can be chained together one one line to be used at once. It must be clear where one ends and the next one begins and where the text representing a parameter ends. (Otherwise you risk getting all the text until the end of the line as a parameter value.) The DSL expressions are tried, one after the other, according to their order in the DSL definition file. After any match, all remaining DSL expressions are investigated, too.

Report a bug

7.5.8. Adding Constraints to Facts

Table 7.4. Adding Constraints to Facts

Name	Description	Example
Expressing LHS conditions	The DSL facility allows you to add constraints to a pattern by a simple convention: if your DSL expression starts with a hyphen (minus character, "-") it is assumed to be a field constraint and, consequently, is is added to the last pattern line preceding it. In the example, the class **Cheese**, has these fields: type, price, age and country. You can express some LHS condition in normal DRL.	`Cheese(age < 5, price == 20, type=="stilton", country=="ch")`

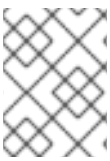
DSL definitions	The DSL definitions given in this example result in three DSL phrases which may be used to create any combination of constraint involving these fields.	`[when]There is a Cheese with=Cheese()`
		`[when]- age is less than {age}=age<{age}`
		`[when]- type is '{type}'=type=='{type}'`
		`[when]- country equal to '{country}'=country=='{country}'`

"-"	The parser will pick up a line beginning with "-" and add it as a constraint to the preceding pattern, inserting a comma when it is required.	`There is a Cheese with`
		`- age is less than 42`
		`- type is 'stilton'`
		
		`Cheese(age<42, type=='stilton')`

Name	Description	Example

Defining DSL phrases	Defining DSL phrases for various operators and even a generic expression that handles any field constraint reduces the amount of DSL entries.	`[when][]is less than or equal to=<=`
		`[when][]is less than=<`
		`[when][]is greater than or equal to=>=`
		`[when][]is greater than=>`
		`[when][]is equal to===`
		`[when][]equals===`
		`[when][]There is a Cheese with=Cheese()`
		[when][]- {field:\w*} {operator} {value:\d*}={field} {operator} {value}

DSL definition rule	n/a	`There is a Cheese with`
		`  - age is less than 42`
		`  - rating is greater than 50`
		`  - type equals 'stilton'`
		In this specific case, a phrase such as "is less than" is replaced by <, and then the line matches the last DSL entry. This removes the hyphen, but the final result is still added as a constraint to the preceding pattern. After processing all of the lines, the resulting DRL text is:
		`Cheese(age<42, rating > 50, type=='stilton')`



NOTE

The order of the entries in the DSL is important if separate DSL expressions are intended to match the same line, one after the other.

Report a bug

7.5.9. Tips for Developing DSLs

- Write representative samples of the rules your application requires and test them as you develop.
- Rules, both in DRL and in DSLR, refer to entities according to the data model representing the application data that should be subject to the reasoning process defined in rules.
- Writing rules is easier if most of the data model's types are facts.
- Mark variable parts as parameters. This provides reliable leads for useful DSL entries.
- You may postpone implementation decisions concerning conditions and actions during this first design phase by leaving certain conditional elements and actions in their DRL form by prefixing a line with a greater sign (">"). (This is also handy for inserting debugging statements.)
- New rules can be written by reusing the existing DSL definitions, or by adding a parameter to an existing condition or consequence entry.
- Keep the number of DSL entries small. Using parameters lets you apply the same DSL sentence for similar rule patterns or constraints.

Report a bug

7.5.10. DSL and DSLR Reference

A DSL file is a text file in a line-oriented format. Its entries are used for transforming a DSLR file into a file according to DRL syntax:

- A line starting with "#" or "//" (with or without preceding white space) is treated as a comment. A comment line starting with "#/" is scanned for words requesting a debug option, see below.
- Any line starting with an opening bracket ("[") is assumed to be the first line of a DSL entry definition.
- Any other line is appended to the preceding DSL entry definition, with the line end replaced by a space.

Report a bug

7.5.11. The Make Up of a DSL Entry

A DSL entry consists of the following four parts:

- A scope definition, written as one of the keywords "when" or "condition", "then" or "consequence", "*" and "keyword", enclosed in brackets ("[" and "]"). This indicates whether the DSL entry is valid for the condition or the consequence of a rule, or both. A scope indication of "keyword" means that the entry has global significance, that is, it is recognized anywhere in a DSLR file.
- A type definition, written as a Java class name, enclosed in brackets. This part is optional unless the next part begins with an opening bracket. An empty pair of brackets is valid, too.
- A DSL expression consists of a (Java) regular expression, with any number of embedded *variable definitions,* terminated by an equal sign ("="). A variable definition is enclosed in braces ("{" and "}"). It consists of a variable name and two optional attachments, separated by

colons (":"). If there is one attachment, it is a regular expression for matching text that is to be assigned to the variable. If there are two attachments, the first one is a hint for the GUI editor and the second one the regular expression.

Note that all characters that are "magic" in regular expressions must be escaped with a preceding backslash ("\") if they should occur literally within the expression.

- The remaining part of the line after the delimiting equal sign is the replacement text for any DSLR text matching the regular expression. It may contain variable references, i.e., a variable name enclosed in braces. Optionally, the variable name may be followed by an exclamation mark ("!") and a transformation function, see below.

  Note that braces ("{" and "}") must be escaped with a preceding backslash ("\") if they should occur literally within the replacement string.

Report a bug

7.5.12. Debug Options for DSL Expansion

Table 7.5. Debug Options for DSL Expansion

Word	Description
result	Prints the resulting DRL text, with line numbers.
steps	Prints each expansion step of condition and consequence lines.
keyword	Dumps the internal representation of all DSL entries with scope "keyword".
when	Dumps the internal representation of all DSL entries with scope "when" or "*".
then	Dumps the internal representation of all DSL entries with scope "then" or "*".
usage	Displays a usage statistic of all DSL entries.

Report a bug

7.5.13. DSL Definition Example

This is what a DSL definition looks like:

```
# Comment: DSL examples

#/ debug: display result and usage

# keyword definition: replaces "regula" by "rule"
[keyword][]regula=rule
```

```
# conditional element: "T" or "t", "a" or "an", convert matched word
[when][][Tt]here is an? {entity:\w+}=
        ${entity!lc}: {entity!ucfirst} ()

# consequence statement: convert matched word, literal braces
[then][]update {entity:\w+}=modify( ${entity!lc} )\{ \}
```

Report a bug

7.5.14. Transformation of a DSLR File

The transformation of a DSLR file proceeds as follows:

1. The text is read into memory.

2. Each of the "keyword" entries is applied to the entire text. The regular expression from the keyword definition is modified by replacing white space sequences with a pattern matching any number of white space characters, and by replacing variable definitions with a capture made from the regular expression provided with the definition, or with the default (".*?"). Then, the DSLR text is searched exhaustively for occurrences of strings matching the modified regular expression. Substrings of a matching string corresponding to variable captures are extracted and replace variable references in the corresponding replacement text, and this text replaces the matching string in the DSLR text.

3. Sections of the DSLR text between "when" and "then", and "then" and "end", respectively, are located and processed in a uniform manner, line by line, as described below.

   For a line, each DSL entry pertaining to the line's section is taken in turn, in the order it appears in the DSL file. Its regular expression part is modified: white space is replaced by a pattern matching any number of white space characters; variable definitions with a regular expression are replaced by a capture with this regular expression, its default being ".*?". If the resulting regular expression matches all or part of the line, the matched part is replaced by the suitably modified replacement text.

   Modification of the replacement text is done by replacing variable references with the text corresponding to the regular expression capture. This text may be modified according to the string transformation function given in the variable reference; see below for details.

   If there is a variable reference naming a variable that is not defined in the same entry, the expander substitutes a value bound to a variable of that name, provided it was defined in one of the preceding lines of the current rule.

4. If a DSLR line in a condition is written with a leading hyphen, the expanded result is inserted into the last line, which should contain a pattern CE, that is, a type name followed by a pair of parentheses. if this pair is empty, the expanded line (which should contain a valid constraint) is simply inserted, otherwise a comma (",") is inserted beforehand.

   If a DSLR line in a consequence is written with a leading hyphen, the expanded result is inserted into the last line, which should contain a "modify" statement, ending in a pair of braces ("{" and "}"). If this pair is empty, the expanded line (which should contain a valid method call) is simply inserted, otherwise a comma (",") is inserted beforehand.

NOTE

It is currently *not* possible to use a line with a leading hyphen to insert text into other conditional element forms (e.g., "accumulate") or it may only work for the first insertion (e.g., "eval").

Report a bug

7.5.15. String Transformation Functions

Table 7.6. String Transformation Functions

Name	Description
uc	Converts all letters to upper case.
lc	Converts all letters to lower case.
ucfirst	Converts the first letter to upper case, and all other letters to lower case.
num	Extracts all digits and "-" from the string. If the last two digits in the original string are preceded by "." or ",", a decimal period is inserted in the corresponding position.
a?b/c	Compares the string with string *a*, and if they are equal, replaces it with *b*, otherwise with *c*. But *c* can be another triplet *a*, *b*, *c*, so that the entire structure is, in fact, a translation table.

Report a bug

7.5.16. Stringing DSL Transformation Functions

Table 7.7. Stringing DSL Transformation Functions

Name	Description	Example

Name	Description	Example

.dsl	A file containing a DSL definition is customarily given the extension **`.dsl`**. It is passed to the Knowledge Builder with **`ResourceType.DSL`**. For a file using DSL definition, the extension **`.dslr`** should be used. The Knowledge Builder expects **`ResourceType.DSLR`**. The IDE, however, relies on file extensions to correctly recognize and work with your rules file.	`# definitions for conditions`
		`[when][]There is an? {entity}=${entity!lc}: {entity!ucfirst}()`
		`[when][]- with an? {attr} greater than {amount}={attr} <= {amount!num}`
		`[when][]- with a {what} {attr}={attr} {what!positive? >0/negative? %lt;0/zero? ==0/ERROR}`

DSL passing	The DSL must be passed to the Knowledge Builder ahead of any rules file using the DSL.
	
	For parsing and expanding a DSLR file the DSL configuration is read and supplied to the parser. Thus, the parser can "recognize" the DSL expressions and transform them into native rule language expressions.
		`KnowledgeBuilder kBuilder = new KnowledgeBuilder();`
		`Resource dsl = ResourceFactory.newClassPathResource( dslPath, getClass() );`
		`kBuilder.add( dsl, ResourceType.DSL );`
		`Resource dslr = ResourceFactory.newClassPathResource( dslrPath, getClass() );`
		`kBuilder.add( dslr, ResourceType.DSLR );`

Report a bug

CHAPTER 8. RULE COMMANDS

8.1. AVAILABLE API

XML marshalling/unmarshalling of the Drools Commands requires the use of special classes, which are going to be described in the following sections.

The following urls show sample script examples for jaxb, xstream and json marshalling using:

- http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/jaxb.mvt?r=HEAD
- http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/jaxb.mvt?r=HEAD
- http://fisheye.jboss.org/browse/JBossRules/trunk/drools-camel/src/test/resources/org/drools/camel/component/xstream.mvt?r=HEAD

XStream

To use the XStream commands marshaller you need to use the DroolsHelperProvider to obtain an XStream instance. We need to use this because it has the commands converters registered.

- Marshalling

  BatchExecutionHelperProviderImpl.newXStreamMarshaller().toXML(command);

- Unmarshalling

  BatchExecutionHelperProviderImpl.newXStreamMarshaller().fromXML(xml)

JSON

JSON API to marshalling/unmarshalling is similar to XStream API:

- Marshalling

  BatchExecutionHelper.newJSonMarshaller().toXML(command);

- Unmarshalling

  BatchExecutionHelper.newJSonMarshaller().fromXML(xml)

JAXB

There are two options for using JAXB, you can define your model in an XSD file or you can have a POJO model. In both cases you have to declare your model inside JAXBContext, and in order to do that you need to use Drools Helper classes. Once you have the JAXBContext you need to create the Unmarshaller/Marshaller as needed.

Using an XSD file to define the model

With your model defined in a XSD file you need to have a KnowledgeBase that has your XSD model added as a resource.

To do this, the XSD file must be added as a XSD ResourceType into the KnowledgeBuilder. Finally you can create the JAXBContext using the KnowledgeBase created with the KnowledgeBuilder

```
Options xjcOpts = new Options();
xjcOpts.setSchemaLanguage(Language.XMLSCHEMA);
```

```
JaxbConfiguration jaxbConfiguration =
KnowledgeBuilderFactory.newJaxbConfiguration( xjcOpts, "xsd" );
kbuilder.add(ResourceFactory.newClassPathResource("person.xsd",
getClass()), ResourceType.XSD, jaxbConfiguration);
KnowledgeBase kbase = kbuilder.newKnowledgeBase();

List<String> classesName = new ArrayList<String>();
classesName.add("org.drools.compiler.test.Person");

JAXBContext jaxbContext =
KnowledgeBuilderHelper.newJAXBContext(classesName.toArray(new
String[classesName.size()]), kbase);
```

Using a POJO model

In this case you need to use DroolsJaxbHelperProviderImpl to create the JAXBContext. This class has two parameters:

1. classNames: A List with the canonical name of the classes that you want to use in the marshalling/unmarshalling process.
2. properties: JAXB custom properties

```
List<String> classNames = new ArrayList<String>();
classNames.add("org.drools.compiler.test.Person");
JAXBContext jaxbContext =
DroolsJaxbHelperProviderImpl.createDroolsJaxbContext(classNames, null);
Marshaller marshaller = jaxbContext.createMarshaller();
```

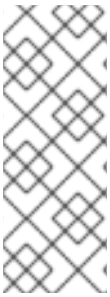
Report a bug

8.2. COMMANDS SUPPORTED

Currently, the following commands are supported:

- BatchExecutionCommand
- InsertObjectCommand
- RetractCommand
- ModifyCommand
- GetObjectCommand
- InsertElementsCommand
- FireAllRulesCommand
- StartProcessCommand
- SignalEventCommand
- CompleteWorkItemCommand
- AbortWorkItemCommand

- QueryCommand
- SetGlobalCommand
- GetGlobalCommand
- GetObjectsCommand



NOTE

In the next snippets code we are going to use a POJO org.drools.compiler.test.Person that has two fields

- name: String
- age: Integer

NOTE

In the next examples, to marshall the commands we have used the next snippet codes:

- XStream

```
String xml =
BatchExecutionHelper.newXStreamMarshaller().toXML(command
);
```

- JSON

```
String xml =
BatchExecutionHelper.newJSonMarshaller().toXML(command);
```

- JAXB

```
Marshaller marshaller = jaxbContext.createMarshaller();
StringWriter xml = new StringWriter();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
true);
marshaller.marshal(command, xml);
```

Report a bug

8.3. COMMANDS

8.3.1. BatchExecutionCommand

- Description: The command that contains a list of commands, which will be sent and executed.
- Attributes

Table 8.1. BatchExecutionCommand attributes

Name	Description	required
lookup	Sets the knowledge session id on which the commands are going to be executed	true
commands	List of commands to be executed	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
InsertObjectCommand insertObjectCommand = new
InsertObjectCommand(new Person("john", 25));
FireAllRulesCommand fireAllRulesCommand = new FireAllRulesCommand();
command.getCommands().add(insertObjectCommand);
command.getCommands().add(fireAllRulesCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <insert>
    <org.drools.compiler.test.Person>
      <name>john</name>
      <age>25</age>
    </org.drools.compiler.test.Person>
  </insert>
  <fire-all-rules/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":[{"insert":
{"object":{"org.drools.compiler.test.Person":
{"name":"john","age":25}}}},{"fire-all-rules":""}]}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <insert>
        <object xsi:type="person"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <age>25</age>
            <name>john</name>
        </object>
    </insert>
    <fire-all-rules max="-1"/>
</batch-execution>
```

Report a bug

8.3.2. InsertObjectCommand

- Description: Insert an object in the knowledge session.
- Attributes

Table 8.2. InsertObjectCommand attributes

Name	Description	required
object	The object to be inserted	true
outIdentifier	Id to identify the FactHandle created in the object insertion and added to the execution results	false
returnObject	Boolean to establish if the object must be returned in the execution results. Default value: true	false
entryPoint	Entrypoint for the insertion	false

- Command creation

```
List<Command> cmds = ArrayList<Command>();

Command insertObjectCommand = CommandFactory.newInsert(new
Person("john", 25), "john", false, null);
cmds.add( insertObjectCommand );

BatchExecutionCommand command =
CommandFactory.createBatchExecution(cmds, "ksession1" );
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <insert out-identifier="john" entry-point="my stream" return-
object="false">
    <org.drools.compiler.test.Person>
      <name>john</name>
      <age>25</age>
    </org.drools.compiler.test.Person>
  </insert>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"insert":
{"entry-point":"my stream", "out-identifier":"john","return-
object":false,"object":{"org.drools.compiler.test.Person":
{"name":"john","age":25}}}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <insert out-identifier="john" entry-point="my stream" >
        <object xsi:type="person"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <age>25</age>
            <name>john</name>
        </object>
    </insert>
</batch-execution>
```

Report a bug

8.3.3. RetractCommand

- Description: Retract an object from the knowledge session.
- Attributes

Table 8.3. RetractCommand attributes

Name	Description	required
handle	The FactHandle associated to the object to be retracted	true

- Command creation: we have two options, with the same output result:

1. Create the Fact Handle from a string

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
RetractCommand retractCommand = new RetractCommand();
retractCommand.setFactHandleFromString("123:234:345:456:567");
command.getCommands().add(retractCommand);
```

2. Set the Fact Handle that you received when the object was inserted

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
RetractCommand retractCommand = new RetractCommand(factHandle);
command.getCommands().add(retractCommand);
```

- XML output

- XStream

```
<batch-execution lookup="ksession1">
  <retract fact-handle="0:234:345:456:567"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"retract":
{"fact-handle":"0:234:345:456:567"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <retract fact-handle="0:234:345:456:567"/>
</batch-execution>
```

Report a bug

8.3.4. ModifyCommand

- Description: Allows you to modify a previously inserted object in the knowledge session.
- Attributes

Table 8.4. ModifyCommand attributes

Name	Description	required
handle	The FactHandle associated to the object to be retracted	true
setters	List of setters object's modifications	true

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
ModifyCommand modifyCommand = new ModifyCommand();
modifyCommand.setFactHandleFromString("123:234:345:456:567");
List<Setter> setters = new ArrayList<Setter>();
setters.add(new SetterImpl("age", "30"));
modifyCommand.setSetters(setters);
command.getCommands().add(modifyCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
```

```
  <modify fact-handle="0:234:345:456:567">
    <set accessor="age" value="30"/>
  </modify>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"modify":
{"fact-handle":"0:234:345:456:567","setters":
{"accessor":"age","value":30}}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <modify fact-handle="0:234:345:456:567">
        <set value="30" accessor="age"/>
    </modify>
</batch-execution>
```

Report a bug

8.3.5. GetObjectCommand

- Description: Used to get an object from a knowledge session
- Attributes

Table 8.5. GetObjectCommand attributes

Name	Description	required
factHandle	The FactHandle associated to the object to be retracted	true
outIdentifier	Id to identify the FactHandle created in the object insertion and added to the execution results	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
GetObjectCommand getObjectCommand = new GetObjectCommand();
getObjectCommand.setFactHandleFromString("123:234:345:456:567");
getObjectCommand.setOutIdentifier("john");
command.getCommands().add(getObjectCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <get-object fact-handle="0:234:345:456:567" out-
identifier="john"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"get-
object":{"fact-handle":"0:234:345:456:567","out-
identifier":"john"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <get-object out-identifier="john" fact-
handle="0:234:345:456:567"/>
</batch-execution>
```

Report a bug

8.3.6. InsertElementsCommand

- Description: Used to insert a list of objects.
- Attributes

Table 8.6. InsertElementsCommand attributes

Name	Description	required
objects	The list of objects to be inserted on the knowledge session	true
outIdentifier	Id to identify the FactHandle created in the object insertion and added to the execution results	false
returnObject	Boolean to establish if the object must be returned in the execution results. Default value: true	false
entryPoint	Entrypoint for the insertion	false

- Command creation

```
List<Command> cmds = ArrayList<Command>();
```

```
List<Object> objects = new ArrayList<Object>();
objects.add(new Person("john", 25));
objects.add(new Person("sarah", 35));

Command insertElementsCommand = CommandFactory.newInsertElements(
objects );
cmds.add( insertElementsCommand );

BatchExecutionCommand command =
CommandFactory.createBatchExecution(cmds, "ksession1" );
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <insert-elements>
    <org.drools.compiler.test.Person>
      <name>john</name>
      <age>25</age>
    </org.drools.compiler.test.Person>
    <org.drools.compiler.test.Person>
      <name>sarah</name>
      <age>35</age>
    </org.drools.compiler.test.Person>
  </insert-elements>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"insert-
elements":{"objects":[{"containedObject":
{"@class":"org.drools.compiler.test.Person","name":"john","age":2
5}},{"containedObject":
{"@class":"Person","name":"sarah","age":35}}]}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <insert-elements return-objects="true">
        <list>
            <element xsi:type="person"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
                <age>25</age>
                <name>john</name>
            </element>
            <element xsi:type="person"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
                <age>35</age>
                <name>sarah</name>
            </element>
        <list>
    </insert-elements>
</batch-execution>
```

Report a bug

8.3.7. FireAllRulesCommand

- Description: Allow execution of the rules activations created.
- Attributes

Table 8.7. FireAllRulesCommand attributes

Name	Description	required
max	The max number of rules activations to be executed. default is -1 and will not put any restriction on execution	false
outIdentifier	Add the number of rules activations fired on the execution results	false
agendaFilter	Allow the rules execution using an Agenda Filter	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
FireAllRulesCommand fireAllRulesCommand = new FireAllRulesCommand();
fireAllRulesCommand.setMax(10);
fireAllRulesCommand.setOutIdentifier("firedActivations");
command.getCommands().add(fireAllRulesCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <fire-all-rules max="10" out-identifier="firedActivations"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"fire-all-
rules":{"max":10,"out-identifier":"firedActivations"}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
   <fire-all-rules out-identifier="firedActivations" max="10"/>
```

```
</batch-execution>
```

Report a bug

8.3.8. StartProcessCommand

- Description: Allows you to start a process using the ID. Also you can pass parameters and initial data to be inserted.
- Attributes

Table 8.8. StartProcessCommand attributes

Name	Description	required
processId	The ID of the process to be started	true
parameters	A Map<String, Object> to pass parameters in the process startup	false
data	A list of objects to be inserted in the knowledge session before the process startup	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
StartProcessCommand startProcessCommand = new StartProcessCommand();
startProcessCommand.setProcessId("org.drools.task.processOne");
command.getCommands().add(startProcessCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <start-process processId="org.drools.task.processOne"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"start-
process":{"process-id":"org.drools.task.processOne"}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <start-process processId="org.drools.task.processOne">
```

```
        <parameter/>
    </start-process>
</batch-execution>
```

Report a bug

8.3.9. SignalEventCommand

- Description: Send a signal event.
- Attributes

Table 8.9. SignalEventCommand attributes

Name	Description	required
event-type		true
processInstanceId		false
event		false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
SignalEventCommand signalEventCommand = new SignalEventCommand();
signalEventCommand.setProcessInstanceId(1001);
signalEventCommand.setEventType("start");
signalEventCommand.setEvent(new Person("john", 25));
command.getCommands().add(signalEventCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <signal-event process-instance-id="1001" event-type="start">
    <org.drools.pipeline.camel.Person>
      <name>john</name>
      <age>25</age>
    </org.drools.pipeline.camel.Person>
  </signal-event>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"signal-
event":{"process-instance-id":1001,"@event-type":"start","event-
type":"start","object":{"org.drools.pipeline.camel.Person":
{"name":"john","age":25}}}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <signal-event event-type="start" process-instance-id="1001">
        <event xsi:type="person"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <age>25</age>
            <name>john</name>
        </event>
    </signal-event>
</batch-execution>
```

Report a bug

8.3.10. CompleteWorkItemCommand

- Description: Allows you to complete a WorkItem.
- Attributes

Table 8.10. CompleteWorkItemCommand attributes

Name	Description	required
workItemId	The ID of the WorkItem to be completed	true
results		false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
CompleteWorkItemCommand completeWorkItemCommand = new
CompleteWorkItemCommand();
completeWorkItemCommand.setWorkItemId(1001);
command.getCommands().add(completeWorkItemCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <complete-work-item id="1001"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"complete-
work-item":{"id":1001}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <complete-work-item id="1001"/>
</batch-execution>
```

Report a bug

8.3.11. AbortWorkItemCommand

- Description: Allows you abort an WorkItem. The same as session.getWorkItemManager().abortWorkItem(workItemId)
- Attributes

Table 8.11. AbortWorkItemCommand attributes

Name	Description	required
workItemId	The ID of the WorkItem to be completed	true

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
AbortWorkItemCommand abortWorkItemCommand = new
AbortWorkItemCommand();
abortWorkItemCommand.setWorkItemId(1001);
command.getCommands().add(abortWorkItemCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <abort-work-item id="1001"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"abort-work-
item":{"id":1001}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <abort-work-item id="1001"/>
</batch-execution>
```

Report a bug

8.3.12. QueryCommand

- Description: Executes a query defined in knowledge base.
- Attributes

  Table 8.12. QueryCommand attributes

Name	Description	required
name	The query name	true
outIdentifier	The identifier of the query results. The query results are going to be added in the execution results with this identifier	false
arguments	A list of objects to be passed as a query parameter	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
QueryCommand queryCommand = new QueryCommand();
queryCommand.setName("persons");
queryCommand.setOutIdentifier("persons");
command.getCommands().add(queryCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <query out-identifier="persons" name="persons"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"query":
{"out-identifier":"persons","name":"persons"}}}}
```

  - JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <query name="persons" out-identifier="persons"/>
</batch-execution>
```

Report a bug

8.3.13. SetGlobalCommand

- Description: Allows you to set a global.
- Attributes

  Table 8.13. SetGlobalCommand attributes

Name	Description	required
identifier	The identifier of the global defined in the knowledge base	true
object	The object to be set into the global	false
out	A boolean to add, or not, the set global result into the execution results	false
outIdentifier	The identifier of the global execution result	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
SetGlobalCommand setGlobalCommand = new SetGlobalCommand();
setGlobalCommand.setIdentifier("helper");
setGlobalCommand.setObject(new Person("kyle", 30));
setGlobalCommand.setOut(true);
setGlobalCommand.setOutIdentifier("output");
command.getCommands().add(setGlobalCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <set-global identifier="helper" out-identifier="output">
    <org.drools.compiler.test.Person>
      <name>kyle</name>
      <age>30</age>
    </org.drools.compiler.test.Person>
  </set-global>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"set-
```

```
global":{"identifier":"helper","out-
identifier":"output","object":{"org.drools.compiler.test.Person":
{"name":"kyle","age":30}}}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <set-global out="true" out-identifier="output"
identifier="helper">
        <object xsi:type="person"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
            <age>30</age>
            <name>kyle</name>
        </object>
    </set-global>
</batch-execution>
```

Report a bug

8.3.14. GetGlobalCommand

- Description: Allows you to get a global previously defined.
- Attributes

Table 8.14. GetGlobalCommand attributes

Name	Description	required
identifier	The identifier of the global defined in the knowledge base	true
outIdentifier	The identifier to be used in the execution results	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
GetGlobalCommand getGlobalCommand = new GetGlobalCommand();
getGlobalCommand.setIdentifier("helper");
getGlobalCommand.setOutIdentifier("helperOutput");
command.getCommands().add(getGlobalCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <get-global identifier="helper" out-identifier="helperOutput"/>
</batch-execution>
```

- JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"get-
global":{"identifier":"helper","out-
identifier":"helperOutput"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <get-global out-identifier="helperOutput"
identifier="helper"/>
</batch-execution>
```

Report a bug

8.3.15. GetObjectsCommand

- Description: Returns all the objects from the current session as a Collection.
- Attributes

Table 8.15. GetObjectsCommand attributes

Name	Description	required
objectFilter	An ObjectFilter to filter the objects returned from the current session	false
outIdentifier	The identifier to be used in the execution results	false

- Command creation

```
BatchExecutionCommand command = new BatchExecutionCommand();
command.setLookup("ksession1");
GetObjectsCommand getObjectsCommand = new GetObjectsCommand();
getObjectsCommand.setOutIdentifier("objects");
command.getCommands().add(getObjectsCommand);
```

- XML output
  - XStream

```
<batch-execution lookup="ksession1">
  <get-objects out-identifier="objects"/>
</batch-execution>
```

  - JSON

```
{"batch-execution":{"lookup":"ksession1","commands":{"get-
```

```
objects":{"out-identifier":"objects"}}}}
```

- JAXB

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<batch-execution lookup="ksession1">
    <get-objects out-identifier="objects"/>
</batch-execution>
```

Report a bug

CHAPTER 9. XML

9.1. THE XML FORMAT

WARNING

The XML rule language has not been updated to support functionality introduced in Drools 5.x and is consider a deprecated feature.

As an option, JBoss Rules supports a "native" rule language as an alternative to DRL. This allows you to capture and manage your rules as XML data. Just like the non-XML DRL format, the XML format is parsed into the internal "AST" representation as fast as possible (using a SAX parser). There is no external transformation step required.

Report a bug

9.2. XML RULE EXAMPLE

This is what a rule looks like in XML:

```
<?xml version="1.0" encoding="UTF-8"?>

<package name="com.sample"
         xmlns="http://drools.org/drools-5.0"
         xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
         xs:schemaLocation="http://drools.org/drools-5.0 drools-5.0.xsd">

<import name="java.util.HashMap" />
<import name="org.drools.*" />

<global identifier="x" type="com.sample.X" />
<global identifier="yada" type="com.sample.Yada" />

<function return-type="void" name="myFunc">
    <parameter identifier="foo" type="Bar" />
    <parameter identifier="bada" type="Bing" />

    <body>
     System.out.println("hello world");
    </body>
</function>

<rule name="simple_rule">
<rule-attribute name="salience" value="10" />
<rule-attribute name="no-loop" value="true" />
<rule-attribute name="agenda-group" value="agenda-group" />
<rule-attribute name="activation-group" value="activation-group" />

<lhs>
```

```
    <pattern identifier="foo2" object-type="Bar" >
            <or-constraint-connective>
                <and-constraint-connective>
                    <field-constraint field-name="a">
                        <or-restriction-connective>
                            <and-restriction-connective>
                                <literal-restriction evaluator=">"
value="60" />
                                <literal-restriction evaluator="<"
value="70" />
                            </and-restriction-connective>
                            <and-restriction-connective>
                                <literal-restriction evaluator="<"
value="50" />
                                <literal-restriction evaluator=">"
value="55" />
                            </and-restriction-connective>
                        </or-restriction-connective>
                    </field-constraint>

                    <field-constraint field-name="a3">
                        <literal-restriction evaluator="==" value="black"
/>
                    </field-constraint>
                </and-constraint-connective>

                <and-constraint-connective>
                    <field-constraint field-name="a">
                        <literal-restriction evaluator="==" value="40" />
                    </field-constraint>

                    <field-constraint field-name="a3">
                        <literal-restriction evaluator="==" value="pink"
/>
                    </field-constraint>
                </and-constraint-connective>

                <and-constraint-connective>
                    <field-constraint field-name="a">
                        <literal-restriction evaluator="==" value="12"/>
                    </field-constraint>

                    <field-constraint field-name="a3">
                        <or-restriction-connective>
                            <literal-restriction evaluator="=="
value="yellow"/>
                            <literal-restriction evaluator="=="
value="blue" />
                        </or-restriction-connective>
                    </field-constraint>
                </and-constraint-connective>
            </or-constraint-connective>
        </pattern>

        <not>
            <pattern object-type="Person">
```

```
                <field-constraint field-name="likes">
                    <variable-restriction evaluator="==" 
identifier="type"/>
                </field-constraint>
            </pattern>

            <exists>
                <pattern object-type="Person">
                    <field-constraint field-name="likes">
                        <variable-restriction evaluator="==" 
identifier="type"/>
                    </field-constraint>
                </pattern>
            </exists>
        </not>

        <or-conditional-element>
            <pattern identifier="foo3" object-type="Bar" >
                <field-constraint field-name="a">
                    <or-restriction-connective>
                        <literal-restriction evaluator="==" value="3" />
                        <literal-restriction evaluator="==" value="4" />
                    </or-restriction-connective>
                </field-constraint>
                <field-constraint field-name="a3">
                    <literal-restriction evaluator="==" value="hello" />
                </field-constraint>
                <field-constraint field-name="a4">
                    <literal-restriction evaluator="==" value="null" />
                </field-constraint>
            </pattern>

            <pattern identifier="foo4" object-type="Bar" >
                <field-binding field-name="a" identifier="a4" />
                <field-constraint field-name="a">
                    <literal-restriction evaluator="!=" value="4" />
                    <literal-restriction evaluator="!=" value="5" />
                </field-constraint>
            </pattern>
        </or-conditional-element>

        <pattern identifier="foo5" object-type="Bar" >
            <field-constraint field-name="b">
                <or-restriction-connective>
                    <return-value-restriction evaluator="==" >a4 + 
1</return-value-restriction>
                    <variable-restriction evaluator=">" identifier="a4" 
/>
                    <qualified-identifier-restriction evaluator="==">
                        org.drools.Bar.BAR_ENUM_VALUE
                    </qualified-identifier-restriction>
                </or-restriction-connective>
            </field-constraint>
        </pattern>

        <pattern identifier="foo6" object-type="Bar" >
```

```
            <field-binding field-name="a" identifier="a4" />
            <field-constraint field-name="b">
                <literal-restriction evaluator="==" value="6" />
            </field-constraint>
        </pattern>
  </lhs>
 <rhs>
    if ( a == b ) {
      assert( foo3 );
    } else {
      retract( foo4 );
    }
    System.out.println( a4 );
   </rhs>
</rule>

</package>
```

Report a bug

9.3. XML ELEMENTS

Table 9.1. XML Elements

Name	Description
global	Defines global objects that can be referred to in the rules.
function	Contains a function declaration for a function to be used in the rules. You have to specify a return type, a unique name and parameters, in the body goes a snippet of code.
import	Imports the types you wish to use in the rule.

Report a bug

9.4. DETAIL OF A RULE ELEMENT

This example rule has LHS and RHS (conditions and consequence) sections. The RHS is a block of semantic code that will be executed when the rule is activated. The LHS is slightly more complicated as it contains nested elements for conditional elements, constraints and restrictions:

```
<rule name="simple_rule">
<rule-attribute name="salience" value="10" />
<rule-attribute name="no-loop" value="true" />
<rule-attribute name="agenda-group" value="agenda-group" />
<rule-attribute name="activation-group" value="activation-group" />

<lhs>
    <pattern identifier="cheese" object-type="Cheese">
```

```
        <from>
            <accumulate>
                <pattern object-type="Person"></pattern>
                <init>
                    int total = 0;
                </init>
                <action>
                    total += $cheese.getPrice();
                </action>
                <result>
                    new Integer( total ) );
                </result>
            </accumulate>
        </from>
    </pattern>

    <pattern identifier="max" object-type="Number">
        <from>
            <accumulate>
                <pattern identifier="cheese" object-type="Cheese">
</pattern>
                <external-function evaluator="max" expression="$price"/>
            </accumulate>
        </from>
    </pattern>
</lhs>
<rhs>
    list1.add( $cheese );
</rhs>
</rule>
```

Report a bug

9.5. XML RULE ELEMENTS

Table 9.2. XML Rule Elements

Element	Description
Pattern	This allows you to specify a type (class) and perhaps bind a variable to an instance of that class. Nested under the pattern object are constraints and restrictions that have to be met. The Predicate and Return Value constraints allow Java expressions to be embedded.
Conditional elements (not, exists, and, or)	These work like their DRL counterparts. Elements that are nested under and an "and" element are logically "anded" together. Likewise with "or" (and you can nest things further). "Exists" and "Not" work around patterns, to check for the existence or nonexistence of a fact meeting the pattern's constraints.

Element	Description
Eval	Allows the execution of a valid snippet of Java code as long as it evaluates to a boolean (do not end it with a semi-colon, as it is just a fragment). This can include calling a function. The Eval is less efficient than the columns, as the rule engine has to evaluate it each time, but it is a "catch all" feature for when you can express what you need to do with Column constraints.

Report a bug

9.6. AUTOMATIC TRANSFORMING BETWEEN XML AND DRL

JBoss Rules comes with some utility classes to transform between formats. This works by parsing the rules from the source format into the AST and then "dumping" out to the appropriate target format. This allows you to, for example, write rules in DRL and export them to XML.

Report a bug

9.7. CLASSES FOR AUTOMATIC TRANSFORMING BETWEEN XML AND DRL

These are the classes to use when transforming between XML and DRL files. Using combinations of these, you can convert between any format (including round trip):

- DrlDumper - for exporting DRL
- DrlParser - for reading DRL
- XmlPackageReader - for reading XML

NOTE

DSLs will not be preserved (from DRLs that are using a DSL) - but they will be able to be converted.

Report a bug

CHAPTER 10. OBJECTS AND INTERFACES

10.1. GLOBALS

Globals are named objects that are made visible to the rule engine, but unlike facts, changes in the object backing a global do not trigger reevaluation of rules. Globals are useful for providing static information, as an object offering services that are used in the RHS of a rule, or as a means to return objects from the rule engine. When you use a global on the LHS of a rule, make sure it is immutable, or else your changes will not have any effect on the behavior of your rules.

Report a bug

10.2. WORKING WITH GLOBALS

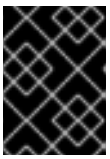
Procedure 10.1. Task

1. To start implementing globals into the Working Memory, declare a global in a rules file and back it up with a Java object:

```
global java.util.List list
```

2. With the Knowledge Base now aware of the global identifier and its type, you can call **`ksession.setGlobal()`** with the global's name and an object (for any session) to associate the object with the global:

```
List list = new ArrayList();
ksession.setGlobal("list", list);
```



IMPORTANT

Failure to declare the global type and identifier in DRL code will result in an exception being thrown from this call.

3. Set the global before it is used in the evaluation of a rule. Failure to do so results in a **`NullPointerException`**.

Report a bug

10.3. RESOLVING GLOBALS

Globals can be resolved in three ways:

getGlobals()

The Stateless Knowledge Session method **`getGlobals()`** returns a Globals instance which provides access to the session's globals. These are shared for all execution calls. Exercise caution regarding mutable globals because execution calls can be executing simultaneously in different threads.

Delegates

Using a delegate is another way of providing global resolution. Assigning a value to a global (with

`setGlobal(String, Object)`) results in the value being stored in an internal collection mapping identifiers to values. Identifiers in this internal collection will have priority over any supplied delegate. If an identifier cannot be found in this internal collection, the delegate global (if any) will be used.

Execution

Execution scoped globals use a `Command` to set a global which is then passed to the `CommandExecutor`.

Report a bug

10.4. SESSION SCOPED GLOBAL EXAMPLE

This is what a session scoped Global looks like:

```
StatelessKieSession ksession = kbase.newStatelessKnowledgeSession();
// Set a global hbnSession, that can be used for DB interactions in the
rules.
ksession.setGlobal( "hbnSession", hibernateSession );
// Execute while being able to resolve the "hbnSession" identifier.
ksession.execute( collection );
```

Report a bug

10.5. STATEFULRULESESSIONS

The `StatefulRuleSession` property is inherited by the `StatefulKnowledgeSession` and provides the rule-related methods that are relevant from outside of the engine.

Report a bug

10.6. AGENDAFILTER OBJECTS

`AgendaFilter` objects are optional implementations of the filter interface which are used to allow or deny the firing of an activation. What is filtered depends on the implementation.

Report a bug

10.7. USING THE AGENDAFILTER

Procedure 10.2. Task

- To use a filter specify it while calling `fireAllRules()`. The following example permits only rules ending in the string `"Test"`. All others will be filtered out:

```
ksession.fireAllRules( new RuleNameEndsWithAgendaFilter( "Test" ) );
```

Report a bug

10.8. RULE ENGINE PHASES

The engine cycles repeatedly through two phases:

Working Memory Actions

This is where most of the work takes place, either in the Consequence (the RHS itself) or the main Java application process. Once the Consequence has finished or the main Java application process calls `fireAllRules()` the engine switches to the Agenda Evaluation phase.

Agenda Evaluation

This attempts to select a rule to fire. If no rule is found it exits. Otherwise it fires the found rule, switching the phase back to Working Memory Actions.

The process repeats until the agenda is clear, in which case control returns to the calling application. When Working Memory Actions are taking place, no rules are being fired.

Report a bug

10.9. THE EVENT MODEL

The event package provides means to be notified of rule engine events, including rules firing, objects being asserted, etc. This allows you, for instance, to separate logging and auditing activities from the main part of your application (and the rules).

Report a bug

10.10. THE KNOWLEGERUNTIMEEVENTMANAGER

The **`KnowlegeRuntimeEventManager`** interface is implemented by the **`KnowledgeRuntime`** which provides two interfaces, **`WorkingMemoryEventManager`** and **`ProcessEventManager`**.

Report a bug

10.11. THE WORKINGMEMORYEVENTMANAGER

The **`WorkingMemoryEventManager`** allows for listeners to be added and removed, so that events for the working memory and the agenda can be listened to.

Report a bug

10.12. ADDING AN AGENDAEVENTLISTENER

The following code snippet shows how a simple agenda listener is declared and attached to a session. It will print activations after they have fired:

```
ksession.addEventListener( new DefaultAgendaEventListener() {
   public void afterActivationFired(AfterActivationFiredEvent event) {
       super.afterActivationFired( event );
       System.out.println( event );
   }
});
```

Report a bug

10.13. PRINTING WORKING MEMORY EVENTS

This code lets you print all Working Memory events by adding a listener:

```
ksession.addEventListener( new DebugWorkingMemoryEventListener() );
```

Report a bug

10.14. KNOWLEGERUNTIMEEVENTS

All emitted events implement the **`KnowlegeRuntimeEvent`** interface which can be used to retrieve the actual **`KnowlegeRuntime`** the event originated from.

Report a bug

10.15. SUPPORTED EVENTS FOR THE KNOWLEDGERUNTIMEEVENT INTERFACE

The events currently supported are:

- ActivationCreatedEvent
- ActivationCancelledEvent
- BeforeActivationFiredEvent
- AfterActivationFiredEvent
- AgendaGroupPushedEvent
- AgendaGroupPoppedEvent
- ObjectInsertEvent
- ObjectRetractedEvent
- ObjectUpdatedEvent
- ProcessCompletedEvent
- ProcessNodeLeftEvent
- ProcessNodeTriggeredEvent
- ProcessStartEvent

Report a bug

10.16. THE KNOWLEDGERUNTIMELOGGER

The KnowledgeRuntimeLogger uses the comprehensive event system in JBoss Rules to create an audit log that can be used to log the execution of an application for later inspection, using tools such as the Eclipse audit viewer.

Report a bug

10.17. ENABLING A FILELOGGER

To enable a FileLogger to track your files, use this code:

```
KnowledgeRuntimeLogger logger =
  KnowledgeRuntimeLoggerFactory.newFileLogger(ksession,
"logdir/mylogfile");
...
logger.close();
```

Report a bug

10.18. USING STATELESSKNOWLEDGESESSION IN JBOSS RULES

The **StatelessKnowledgeSession** wraps the **StatefulKnowledgeSession**, instead of extending it. Its main focus is on decision service type scenarios. It avoids the need to call **dispose()**. Stateless sessions do not support iterative insertions and the method call **fireAllRules()** from Java code. The act of calling **execute()** is a single-shot method that will internally instantiate a **StatefulKnowledgeSession**, add all the user data and execute user commands, call **fireAllRules()**, and then call **dispose()**. While the main way to work with this class is via the **BatchExecution** (a subinterface of **Command**) as supported by the **CommandExecutor** interface, two convenience methods are provided for when simple object insertion is all that's required. The **CommandExecutor** and **BatchExecution** are talked about in detail in their own section.

Report a bug

10.19. PERFORMING A STATELESSKNOWLEDGESESSION EXECUTION WITH A COLLECTION

This the code for performing a StatelessKnowledgeSession execution with a collection:

```
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();
kbuilder.add( ResourceFactory.newFileSystemResource( fileName ),
ResourceType.DRL );
if (kbuilder.hasErrors() ) {
    System.out.println( kbuilder.getErrors() );
} else {
    Kie kbase = KnowledgeBaseFactory.newKnowledgeBase();
    kbase.addKnowledgePackages( kbuilder.getKnowledgePackages() );
    StatelessKieSession ksession = kbase.newStatelessKnowledgeSession();
    ksession.execute( collection );
}
```

Report a bug

10.20. PERFORMING A STATELESSKNOWLEDGESESSION EXECUTION WITH THE INSERTELEMENTS COMMAND

This is the code for performing a StatelessKnowledgeSession execution with the InsertElements Command:

```
ksession.execute( CommandFactory.newInsertElements( collection ) );
```

NOTE

To insert the collection and its individual elements, use **`CommandFactory.newInsert(collection)`**.

Report a bug

10.21. THE BATCHEXECUTIONHELPER

Methods of the **`CommandFactory`** create the supported commands, all of which can be marshaled using XStream and the **`BatchExecutionHelper`**. **`BatchExecutionHelper`** provides details on the XML format as well as how to use JBoss Rules Pipeline to automate the marshaling of **`BatchExecution`** and **`ExecutionResults`**.

Report a bug

10.22. THE COMMANDEXECUTOR INTERFACE

The **`CommandExecutor`** interface allows users to export data using "out" parameters. This means that inserted facts, globals and query results can all be returned using this interface.

Report a bug

10.23. OUT IDENTIFIERS

This is an example of what out identifiers look like:

```
// Set up a list of commands
List cmds = new ArrayList();
cmds.add( CommandFactory.newSetGlobal( "list1", new ArrayList(), true ) );
cmds.add( CommandFactory.newInsert( new Person( "jon", 102 ), "person" ) );
cmds.add( CommandFactory.newQuery( "Get People" "getPeople" );

// Execute the list
ExecutionResults results =
  ksession.execute( CommandFactory.newBatchExecution( cmds ) );

// Retrieve the ArrayList
results.getValue( "list1" );
// Retrieve the inserted Person fact
results.getValue( "person" );
// Retrieve the query as a QueryResults instance.
results.getValue( "Get People" );
```

Report a bug

CHAPTER 11. COMPLEX EVENT PROCESSING

11.1. INTRODUCTION TO COMPLEX EVENT PROCESSING

JBoss BRMS Complex Event Processing provides the JBoss Enterprise BRMS Platform with complex event processing capabilities.

For the purpose of this guide, *Complex Event Processing*, or CEP, refers to the ability to process multiple events and detect interesting events from within a collection of events, uncover relationships that exist between events, and infer new data from the events and their relationships.

An *event* can best be described as a record of a significant change of state in the application domain. Depending on how the domain is modeled, the change of state may be represented by a single event, multiple atomic events, or even hierarchies of correlated events. Using a stock broker application as an example, a change in security prices, a change in ownership from seller to buyer, or a change in an account holder's balance are all considered to be events as a change has occurred in the state of the application domain.

Event processing use cases, in general, share several requirements and goals with *business rules use cases*.

From a business perspective, business rule definitions are often defined based on the occurrence of scenarios triggered by events. For example:

- On an algorithmic trading application: Take an action if the security price increases X% above the day's opening price.

  The price increases are denoted by events on a stock trade application.

- On a monitoring application: Take an action if the temperature in the server room increases X degrees in Y minutes.

  The sensor readings are denoted by events.

Both business rules and event processing queries change frequently and require an immediate response for the business to adapt to new market conditions, regulations, and corporate policies.

From a technical perspective:

- Both business rules and event processing require seamless integration with the enterprise infrastructure and applications. This is particularly important with regard to life-cycle management, auditing, and security.

- Both business rules and event processing have functional requirements like *pattern matching* and non-functional requirements like response time limits and query/rule explanations.

NOTE

JBoss BRMS Complex Event Processing provides the complex event processing capabilities of JBoss Business Rules Management System. The Business Rules Management and Business Process Management capabilities are provided by other modules.

Complex event processing scenarios share these distinguishing characteristics:

- They usually process large numbers of events, but only a small percentage of the events are of interest.
- The events are usually immutable, as they represent a record of change in state.
- The rules and queries run against events and must react to detected event patterns.
- There are usually strong temporal relationships between related events.
- Individual events are not important. The system is concerned with patterns of related events and the relationships between them.
- It is often necessary to perform composition and aggregation of events.

As such, JBoss BRMS Complex Event Processing supports the following behaviors:

- Support events, with their proper semantics, as *first class citizens*.
- Allow detection, correlation, aggregation, and composition of events.
- Support processing streams of events.
- Support temporal constraints in order to model the temporal relationships between events.
- Support *sliding windows* of interesting events.
- Support a *session-scoped* unified clock.
- Support the required volumes of events for complex event processing use cases.
- Support reactive rules.
- Support adapters for event input into the engine (pipeline).

The rest of this guide describes each of the features that JBoss BRMS Complex Event Processing provides.

Report a bug

CHAPTER 12. FEATURES OF JBOSS BRMS COMPLEX EVENT PROCESSING

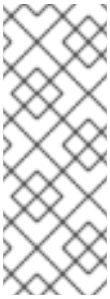
12.1. EVENTS

Events are a record of significant change of state in the application domain. From a complex event processing perspective, an event is a special type of fact or object. A fact is a known piece of data. For instance, a fact could be a stock's opening price. A rule is a definition of how to react to the data. For instance, if a stock price reaches $X, sell the stock.

The defining characteristics of events are the following:

Events are *immutable*

An event is a record of change which has occurred at some time in the past, and as such it cannot be changed.



NOTE

The rules engine does not enforce immutability on the Java objects representing events; this makes *event data enrichment* possible.

The application should be able to populate un-populated event attributes, which can be used to enrich the event with inferred data; however, event attributes that have already been populated should not be changed.

Events have strong *temporal constraints*

Rules involving events usually require the correlation of multiple events that occur at different points in time relative to each other.

Events have *managed life-cycles*

Because events are immutable and have temporal constraints, they are usually only of interest for a specified period of time. This means the engine can automatically manage the life-cycle of events.

Events can use *sliding windows*

It is possible to define and use sliding windows with events since all events have timestamps associated with them. Therefore, sliding windows allow the creation of rules on aggregations of values over a time period.

Events can be declared as either *interval-based* events or *point-in-time* events. Interval-based events have a duration time and persist in working memory until their duration time has lapsed. Point-in-time events have no duration and can be thought of as interval-based events with a duration of zero.

Report a bug

12.2. EVENT DECLARATION

To declare a fact type as an event, assign the `@role` meta-data tag to the fact with the `event` parameter. The @role meta-data tag can accept two possible values:

- **fact**: Assigning the fact role declares the type is to be handled as a regular fact. Fact is the default role.
- **event**: Assigning the event role declares the type is to be handled as an event.

This example declares that a stock broker application's **StockTick** fact type will be handled as an event:

Example 12.1. Declaring a Fact Type as an Event

```
import some.package.StockTick

declare StockTick
    @role( event )
end
```

Facts can also be declared inline. If **StockTick** was a fact type declared in the DRL instead of in a pre-existing class, the code would be as follows:

Example 12.2. Declaring a Fact Type and Assigning it to an Event Role

```
declare StockTick
    @role( event )

    datetime : java.util.Date
    symbol : String
    price : double
end
```

For more information on *type declarations*, please refer to the Rule Language section of the *JBoss Rules Reference Guide*.

Report a bug

12.3. EVENT META-DATA

Every event has associated meta-data. Typically, the meta-data is automatically added as each event is inserted into working memory. The meta-data defaults can be changed on an event-type basis using the meta-data tags:

- @role
- @timestamp
- @duration
- @expires

The following examples assume the application domain model includes the following class:

Example 12.3. The VoiceCall Fact Class

```
/**
 * A class that represents a voice call in
 * a Telecom domain model
 */
public class VoiceCall {
    private String  originNumber;
    private String  destinationNumber;
    private Date    callDateTime;
    private long    callDuration;          // in milliseconds

    // constructors, getters, and setters
}
```

@role

The @role meta-data tag indicates whether a given fact type is either a regular fact or an event. It accepts either **fact** or **event** as a parameter. The default is **fact**.

```
@role( <fact|event> )
```

Example 12.4. Declaring VoiceCall as an Event Type

```
declare VoiceCall
    @role( event )
end
```

@timestamp

A timestamp is automatically assigned to every event. By default, the time is provided by the session clock and assigned to the event at insertion into the working memory. Events can have their own timestamp attribute, which can be included by telling the engine to use the attribute's timestamp instead of the session clock.

To use the attribute's timestamp, use the attribute name as the parameter for the **@timestamp** tag.

```
@timestamp( <attributeName> )
```

Example 12.5. Declaring the VoiceCall Timestamp Attribute

```
declare VoiceCall
    @role( event )
    @timestamp( callDateTime )
end
```

@duration

JBoss BRMS Complex Event Processing supports both point-in-time and interval-based events. A point-in-time event is represented as an interval-based event with a duration of zero time units. By default, every event has a duration of zero. To assign a different duration to an event, use the attribute name as the parameter for the **@duration** tag.

```
@duration( <attributeName> )
```

Example 12.6. Declaring the VoiceCall Duration Attribute

```
declare VoiceCall
    @role( event )
    @timestamp( callDateTime )
    @duration( callDuration )
end
```

@expires

Events may be set to expire automatically after a specific duration in the working memory. By default, this happens when the event can no longer match and activate any of the current rules. You can also explicitly define when an event should expire. The @expires tag is only used when the engine is running in *stream* mode.

```
@expires( <timeOffset> )
```

The value of **`timeOffset`** is a temporal interval that sets the relative duration of the event.

```
[#d][#h][#m][#s][#[ms]]
```

All parameters are optional and the # parameter should be replaced by the appropriate value.

To declare that the **`VoiceCall`** facts should expire one hour and thirty-five minutes after insertion into the working memory, use the following:

Example 12.7. Declaring the Expiration Offset for the VoiceCall Events

```
declare VoiceCall
    @role( event )
    @timestamp( callDateTime )
    @duration( callDuration )
    @expires( 1h35m )
end
```

See Also:

- Section 12.6, “Event Processing Modes”
- Section 12.14.2, “Explicit Expiration”

Report a bug

12.4. SESSION CLOCK

Events have strong temporal constraints making it is necessary to use a reference clock. If a rule needs to determine the average price of a given stock over the last sixty minutes, it is necessary to compare

the stock price event's timestamp with the current time. The reference clock provides the current time.

Because the rules engine can simultaneously run an array of different scenarios that require different clocks, multiple clock implementations can be used by the engine.

Scenarios that require different clocks include the following:

- Rules testing: Testing always requires a controlled environment, and when the tests include rules with temporal constraints, it is necessary to control the input rules, facts, and the flow of time.
- Regular execution: A rules engine that reacts to events in real time needs a real-time clock.
- Special environments: Specific environments may have specific time control requirements. For instance, clustered environments may require clock synchronization or JEE environments may require you to use an application server-provided clock.
- Rules replay or simulation: In order to replay or simulate scenarios, it is necessary that the application controls the flow of time.

Report a bug

12.5. AVAILABLE CLOCK IMPLEMENTATIONS

JBoss BRMS Complex Event Processing comes equipped with two clock implementations:

Real-Time Clock

The real-time clock is the default implementation based on the system clock. The real-time clock uses the system clock to determine the current time for timestamps.

To explicitly configure the engine to use the real-time clock, set the session configuration parameter to ***realtime***:

```
KieSessionConfiguration config =
KieServices.Factory.get().newKieSessionConfiguration()
   config.setOption( ClockTypeOption.get("realtime") );
```

Pseudo-Clock

The pseudo-clock is useful for testing temporal rules since it can be controlled by the application.

To explicitly configure the engine to use the pseudo-clock, set the session configuration parameter to ***pseudo***:

```
KieSessionConfiguration config =
KieServices.Factory.get().newKieSessionConfiguration();
   config.setOption( ClockTypeOption.get("pseudo") );
```

This example shows how to control the pseudo-clock:

```
KieSessionConfiguration conf =
KieServices.Factory.get().newKieSessionConfiguration();
   conf.setOption( ClockTypeOption.get( "pseudo" ) );
   KieSession session = kbase.newKieSession( conf, null );
```

```
SessionPseudoClock clock = session.getSessionClock();

// then, while inserting facts, advance the clock as necessary:
FactHandle handle1 = session.insert( tick1 );
clock.advanceTime( 10, TimeUnit.SECONDS );
FactHandle handle2 = session.insert( tick2 );
clock.advanceTime( 30, TimeUnit.SECONDS );
FactHandle handle3 = session.insert( tick3 );
```

Report a bug

12.6. EVENT PROCESSING MODES

Rules engines process facts and rules to provide applications with results. Regular facts (facts with no temporal constraints) are processed independent of time and in no particular order. JBoss BRMS processes facts of this type in cloud mode. Events (facts which have strong temporal constraints) must be processed in real-time or near real-time. JBoss BRMS processes these events in stream mode. Stream mode deals with synchronization and makes it possible for JBoss BRMS to process events.

Report a bug

12.7. CLOUD MODE

Cloud mode is the default operating mode of JBoss Business Rules Management System.

Running in Cloud mode, the engine applies a many-to-many pattern matching algorithm, which treats the events as an unordered cloud. Events still have timestamps, but there is no way for the rules engine running in Cloud mode to draw relevance from the timestamp because Cloud mode is unaware of the present time.

This mode uses the rules constraints to find the matching tuples, activate, and fire rules.

Cloud mode does not impose any kind of additional requirements on facts; however, because it has no concept of time, it cannot take advantage of temporal features such as *sliding windows* or *automatic life-cycle management*. In Cloud mode, it is necessary to explicitly retract events when they are no longer needed.

Certain requirements that are not imposed include the following:

- No need for clock synchronization since there is no notion of time.
- No requirement on ordering events since the engine looks at the events as an unordered cloud against which the engine tries to match rules.

Cloud mode can be specified either by setting a system property, using configuration property files, or via the API.

The API call follows:

```
KieBaseConfiguration config =
KieServices.Factory.get().newKieBaseConfiguration();
   config.setOption( EventProcessingOption.CLOUD );
```

The equivalent property follows:

```
drools.eventProcessingMode = cloud
```

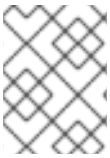
Report a bug

12.8. STREAM MODE

Stream mode processes events chronologically as they are inserted into the rules engine. Stream mode uses a session clock that enables the rules engine to process events as they occur in time. The session clock enables processing events as they occur based on the age of the events. Stream mode also synchronizes streams of events (so events in different streams can be processed in chronological order), implements sliding windows of interest, and enables automatic life-cycle management.

The requirements for using stream mode are the following:

- Events in each stream must be ordered chronologically.
- A session clock must be present to synchronize event streams.



NOTE

The application does not need to enforce ordering events between streams, but the use of event streams that have not been synchronized may cause unexpected results.

Stream mode can be enabled by setting a system property, using configuration property files, or via the API.

The API call follows:

```
KieBaseConfiguration config =
KieServices.Factory.get().newKieBaseConfiguration();
   config.setOption( EventProcessingOption.STREAM );
```

The equivalent property follows:

```
drools.eventProcessingMode = stream
```

Report a bug

12.9. SUPPORT FOR EVENT STREAMS

*Complex event processing use cases*deal with streams of events. The streams can be provided to the application via JMS queues, flat text files, database tables, raw sockets, or even web service calls.

Streams share a common set of characteristics:

- Events in the stream are ordered by timestamp. The timestamps may have different semantics for different streams, but they are always ordered internally.
- There is usually a high volume of events in the stream.
- Atomic events contained in the streams are rarely useful by themselves.

- Streams are either homogeneous (they contain a single type of event) or heterogeneous (they contain events of different types).

A stream is also known as an *entry point*.

Facts from one entry point, or stream, may join with facts from any other entry point in addition to facts already in working memory. Facts always remain associated with the entry point through which they entered the engine. Facts of the same type may enter the engine through several entry points, but facts that enter the engine through entry point A will never match a pattern from entry point B.

See Also:

- Section 12.10, “Declaring and Using Entry Points”

Report a bug

12.10. DECLARING AND USING ENTRY POINTS

Entry points are declared implicitly by making direct use of them in rules. Referencing an entry point in a rule will make the engine, at compile time, identify and create the proper internal structures to support that entry point.

For example, a banking application that has transactions fed into the engine via streams could have one stream for all of the transactions executed at ATMs. A rule for this scenario could state, "A withdrawal is only allowed if the account balance is greater than the withdrawal amount the customer has requested."

Example 12.8. Example ATM Rule

```
rule "authorize withdraw"
   when
       WithdrawRequest( $ai : accountId, $am : amount ) from entry-point
"ATM Stream"
       CheckingAccount( accountId == $ai, balance > $am )
   then
       // authorize withdraw
   end
```

When the engine compiles this rule, it will identify that the pattern is tied to the entry point "ATM Stream." The engine will create all the necessary structures for the rule-base to support the "ATM Stream", and this rule will only match **`WithdrawRequest`** events coming from the "ATM Stream."

Note the ATM example rule joins the event (**`WithdrawalRequest`**) from the stream with a fact from the main working memory (**`CheckingAccount`**).

The banking application may have a second rule that states, "A fee of $2 must be applied to a withdraw request made via a branch teller."

Example 12.9. Using Multiple Streams

```
rule "apply fee on withdraws on branches"
   when
       WithdrawRequest( $ai : accountId, processed == true ) from entry-
```

```
point "Branch Stream"
       CheckingAccount( accountId == $ai )
   then
       // apply a $2 fee on the account
   end
```

This rule matches events of the same type (**`WithdrawRequest`**) as the example ATM rule but from a different stream. Events inserted into the "ATM Stream" will never match the pattern on the second rule, which is tied to the "Branch Stream;" accordingly, events inserted into the "Branch Stream" will never match the pattern on the example ATM rule, which is tied to the "ATM Stream".

Declaring the stream in a rule states that the rule is only interested in events coming from that stream.

Events can be inserted manually into an entry point instead of directly into the working memory.

Example 12.10. Inserting Facts into an Entry Point

```
   // create your rulebase and your session as usual
   KieSession session = ...

   // get a reference to the entry point
   WorkingMemoryEntryPoint atmStream =
session.getWorkingMemoryEntryPoint( "ATM Stream" );

   // and start inserting your facts into the entry point
   atmStream.insert( aWithdrawRequest );
```

Report a bug

12.11. NEGATIVE PATTERN IN STREAM MODE

A *negative pattern* is concerned with conditions that are not met. Negative patterns make reasoning in the absence of events possible. For instance, a safety system could have a rule that states, "If a fire is detected and the sprinkler is *not* activated, sound the alarm."

In Cloud mode, the engine assumes all facts (regular facts and events) are known in advance and evaluates negative patterns immediately.

Example 12.11. A Rule with a Negative Pattern

```
rule "Sound the alarm"
when
    $f : FireDetected( )
    not( SprinklerActivated( ) )
then
    // sound the alarm
end
```

An example in stream mode is displayed below. This rule keeps consistency when dealing with negative patterns and temporal constraints at the same time interval.

Example 12.12. A Rule with a Negative Pattern, Temporal Constraints, and an Explicit Duration Parameter.

```
rule "Sound the alarm"
    duration( 10s )
when
    $f : FireDetected( )
    not( SprinklerActivated( this after[0s,10s] $f ) )
then
    // sound the alarm
end
```

In stream mode, negative patterns with temporal constraints may force the engine to wait for a set time before activating a rule. A rule may be written for an alarm system that states, "If a fire is detected and the sprinkler is *not* activated after 10 seconds, sound the alarm." Unlike the previous stream mode example, this one does not require the user to calculate and write the duration parameter.

Example 12.13. A Rule with a Negative Pattern with Temporal Constraints

```
rule "Sound the alarm"
when
    $f : FireDetected( )
    not( SprinklerActivated( this after[0s,10s] $f ) )
then
    // sound the alarm
end
```

The rule depicted below expects one "Heartbeat" event to occur every 10 seconds; if not, the rule fires. What is special about this rule is that it uses the same type of object in the first pattern and in the negative pattern. The negative pattern has the temporal constraint to wait between 0 to 10 seconds before firing, and it excludes the Heartbeat bound to $h. Excluding the bound Heartbeat is important since the temporal constraint [0s, ...] does not exclude by itself the bound event $h from being matched again, thus preventing the rule to fire.

Example 12.14. Excluding Bound Events in Negative Patterns

```
rule "Sound the alarm"
when
    $h: Heartbeat( ) from entry-point "MonitoringStream"
    not( Heartbeat( this != $h, this after[0s,10s] $h ) from entry-point
"MonitoringStream" )
then
    // Sound the alarm
end
```

Report a bug

12.12. TEMPORAL REASONING

12.12.1. Temporal Reasoning

Complex Event Processing requires the rules engine to engage in temporal reasoning. Events have strong temporal constraints so it is vital the rules engine can determine and interpret an event's temporal attributes, both as they relate to other events and the 'flow of time' as it appears to the rules engine. This makes it possible for rules to take time into account; for instance, a rule could state, "Calculate the average price of a stock over the last 60 minutes."

NOTE

JBoss BRMS Complex Event Processing implements interval-based time events, which have a duration attribute that is used to indicate how long an event is of interest. Point-in-time events are also supported and treated as interval-based events with a duration of 0 (zero).

Report a bug

12.12.2. Temporal Operations

12.12.2.1. Temporal Operations

JBoss BRMS Complex Event Processing implements 13 temporal operators and their logical complements (negation). The 13 temporal operators are the following:

- After
- Before
- Coincides
- During
- Finishes
- Finishes By
- Includes
- Meets
- Met By
- Overlaps
- Overlapped By
- Starts
- Started By

Report a bug

12.12.2.2. After

The **after** operator correlates two events and matches when the temporal distance (the time between the two events) from the current event to the event being correlated falls into the distance range declared for the operator.

For example:

```
$eventA : EventA( this after[ 3m30s, 4m ] $eventB )
```

This pattern only matches if the temporal distance between the time when **$eventB** finished and the time when **$eventA** started is between the lower limit of three minutes and thirty seconds and the upper limit of four minutes.

This can also be represented as follows:

```
  3m30s <= $eventA.startTimestamp - $eventB.endTimeStamp <= 4m
```

The **after** operator accepts one or two optional parameters:

- If two values are defined, the interval starts on the first value (3 minutes and 30 seconds in the example) and ends on the second value (4 minutes in the example).
- If only one value is defined, the interval starts on the provided value and runs indefinitely with no end time.
- If no value is defined, the interval starts at one millisecond and runs indefinitely with no end time.

The **after** operator also accepts negative temporal distances.

For example:

```
$eventA : EventA( this after[ -3m30s, -2m ] $eventB )
```

If the first value is greater than the second value, the engine will automatically reverse them.

The following two patterns are equivalent to each other:

```
$eventA : EventA( this after[ -3m30s, -2m ] $eventB )
   $eventA : EventA( this after[ -2m, -3m30s ] $eventB )
```

Report a bug

12.12.2.3. Before

The **before** operator correlates two events and matches when the temporal distance (time between the two events) from the event being correlated to the current event falls within the distance range declared for the operator.

For example:

```
$eventA : EventA( this before[ 3m30s, 4m ] $eventB )
```

This pattern only matches if the temporal distance between the time when **$eventA** finished and the time when **$eventB** started is between the lower limit of three minutes and thirty seconds and the upper limit of four minutes.

This can also be represented as follows:

```
3m30s <= $eventB.startTimestamp - $eventA.endTimeStamp <= 4m
```

The **before** operator accepts one or two optional parameters:

- If two values are defined, the interval starts on the first value (3 minutes and 30 seconds in the example) and ends on the second value (4 minutes in the example).
- If only one value is defined, the interval starts on the provided value and runs indefinitely with no end time.
- If no value is defined, the interval starts at one millisecond and runs indefinitely with no end time.

The **before** operator also accepts negative temporal distances.

For example:

```
$eventA : EventA( this before[ -3m30s, -2m ] $eventB )
```

If the first value is greater than the second value, the engine will automatically reverse them.

The following two patterns are equivalent to each other:

```
$eventA : EventA( this before[ -3m30s, -2m ] $eventB )
   $eventA : EventA( this before[ -2m, -3m30s ] $eventB )
```

Report a bug

12.12.2.4. Coincides

The **coincides** operator correlates two events and matches when both events happen at the same time.

For example:

```
$eventA : EventA( this coincides $eventB )
```

This pattern only matches if both the start timestamps of **$eventA** and **$eventB** are identical and the end timestamps of both **$eventA** and **$eventB** are also identical.

The **coincides** operator accepts optional thresholds for the distance between the events' start times and the events' end times, so the events do not have to start at exactly the same time or end at exactly the same time, but they need to be within the provided thresholds.

The following rules apply when defining thresholds for the **coincides** operator:

- If only one parameter is given, it is used to set the threshold for both the start and end times of both events.

- If two parameters are given, the first is used as a threshold for the start time and the second one is used as a threshold for the end time.

For example:

```
$eventA : EventA( this coincides[15s, 10s] $eventB )
```

This pattern will only match if the following conditions are met:

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 15s &&
   abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 10s
```

WARNING

The **coincides** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance internals.

Report a bug

12.12.2.5. During

The **during** operator correlates two events and matches when the current event happens during the event being correlated.

For example:

```
$eventA : EventA( this during $eventB )
```

This pattern only matches if **$eventA** starts after **$eventB** and ends before **$eventB** ends.

This can also be represented as follows:

```
$eventB.startTimestamp < $eventA.startTimestamp <= $eventA.endTimestamp <
$eventB.endTimestamp
```

The **during** operator accepts one, two, or four optional parameters:

The following rules apply when providing parameters for the **during** operator:

- If one value is defined, this value will represent the maximum distance between the start times of the two events and the maximum distance between the end times of the two events.
- If two values are defined, these values represent a threshold that the current event's start time and end time must occur between in relation to the correlated event's start and end times.

  If the values 5s and 10s are provided, the current event must start between 5 and 10 seconds after the correlated event, and similarly the current event must end between 5 and 10 seconds before the correlated event.

- If four values are defined, the first and second values will be used as the minimum and maximum distances between the starting times of the events, and the third and fourth values will be used as the minimum and maximum distances between the end times of the two events.

Report a bug

12.12.2.6. Finishes

The **finishes** operator correlates two events and matches when the current event's start timestamp post-dates the correlated event's start timestamp and both events end simultaneously.

For example:

```
$eventA : EventA( this finishes $eventB )
```

This pattern only matches if **$eventA** starts after **$eventB** starts and ends at the same time as **$eventB** ends.

This can be represented as follows:

```
$eventB.startTimestamp < $eventA.startTimestamp &&
   $eventA.endTimestamp == $eventB.endTimestamp
```

The **finishes** operator accepts one optional parameter. If defined, the optional parameter sets the maximum time allowed between the end times of the two events.

For example:

```
$eventA : EventA( this finishes[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
$eventB.startTimestamp < $eventA.startTimestamp &&
   abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 5s
```

WARNING

The **finishes** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

12.12.2.7. Finishes By

The **finishedby** operator correlates two events and matches when the current event's start time predates the correlated event's start time but both events end simultaneously. **finishedby** is the symmetrical opposite of the **finishes** operator.

For example:

```
$eventA : EventA( this finishedby $eventB )
```

This pattern only matches if **$eventA** starts before **$eventB** starts and ends at the same time as **$eventB** ends.

This can be represented as follows:

```
$eventA.startTimestamp < $eventB.startTimestamp &&
   $eventA.endTimestamp == $eventB.endTimestamp
```

The **finishedby** operator accepts one optional parameter. If defined, the optional parameter sets the maximum time allowed between the end times of the two events.

```
$eventA : EventA( this finishedby[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
$eventA.startTimestamp < $eventB.startTimestamp &&
   abs( $eventA.endTimestamp - $eventB.endTimestamp ) <= 5s
```

WARNING

The **finishedby** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

12.12.2.8. Includes

The **includes** operator examines two events and matches when the event being correlated happens during the current event. It is the symmetrical opposite of the **during** operator.

For example:

```
$eventA : EventA( this includes $eventB )
```

This pattern only matches if **$eventB** starts after **$eventA** and ends before **$eventA** ends.

This can be represented as follows:

```
$eventA.startTimestamp < $eventB.startTimestamp <= $eventB.endTimestamp <
$eventA.endTimestamp
```

The **includes** operator accepts 1, 2 or 4 optional parameters:

- If one value is defined, this value will represent the maximum distance between the start times of the two events and the maximum distance between the end times of the two events.
- If two values are defined, these values represent a threshold that the current event's start time and end time must occur between in relation to the correlated event's start and end times.

  If the values 5s and 10s are provided, the current event must start between 5 and 10 seconds after the correlated event, and similarly the current event must end between 5 and 10 seconds before the correlated event.
- If four values are defined, the first and second values will be used as the minimum and maximum distances between the starting times of the events, and the third and fourth values will be used as the minimum and maximum distances between the end times of the two events.

Report a bug

12.12.2.9. Meets

The **meets** operator correlates two events and matches when the current event ends at the same time as the correlated event starts.

For example:

```
$eventA : EventA( this meets $eventB )
```

This pattern matches if **$eventA** ends at the same time as **$eventB** starts.

This can be represented as follows:

```
abs( $eventB.startTimestamp - $eventA.endTimestamp ) == 0
```

The **meets** operator accepts one optional parameter. If defined, it determines the maximum time allowed between the end time of the current event and the start time of the correlated event.

For example:

```
$eventA : EventA( this meets[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
abs( $eventB.startTimestamp - $eventA.endTimestamp) <= 5s
```

WARNING

The **meets** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

12.12.2.10. Met By

The **metby** operator correlates two events and matches when the current event starts at the same time as the correlated event ends.

For example:

```
$eventA : EventA( this metby $eventB )
```

This pattern matches if **$eventA** starts at the same time as **$eventB** ends.

This can be represented as follows:

```
abs( $eventA.startTimestamp - $eventB.endTimestamp ) == 0
```

The **metby** operator accepts one optional parameter. If defined, it sets the maximum distance between the end time of the correlated event and the start time of the current event.

For example:

```
$eventA : EventA( this metby[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
abs( $eventA.startTimestamp - $eventB.endTimestamp) <= 5s
```

WARNING

The **metby** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

12.12.2.11. Overlaps

The **overlaps** operator correlates two events and matches when the current event starts before the correlated event starts and ends after the correlated event starts, but it ends before the correlated event ends.

For example:

```
$eventA : EventA( this overlaps $eventB )
```

This pattern matches if these conditions are met:

```
$eventA.startTimestamp < $eventB.startTimestamp < $eventA.endTimestamp <
$eventB.endTimestamp
```

The **overlaps** operator accepts one or two optional parameters:

- If one parameter is defined, it will define the maximum distance between the start time of the correlated event and the end time of the current event.
- If two values are defined, the first value will be the minimum distance, and the second value will be the maximum distance between the start time of the correlated event and the end time of the current event.

Report a bug

12.12.2.12. Overlapped By

The **overlappedby** operator correlates two events and matches when the correlated event starts before the current event, and the correlated event ends after the current event starts but before the current event ends.

For example:

```
$eventA : EventA( this overlappedby $eventB )
```

This pattern matches if these conditions are met:

```
$eventB.startTimestamp < $eventA.startTimestamp < $eventB.endTimestamp <
$eventA.endTimestamp
```

The **overlappedby** operator accepts one or two optional parameters:

- If one parameter is defined, it sets the maximum distance between the start time of the correlated event and the end time of the current event.
- If two values are defined, the first value will be the minimum distance, and the second value will be the maximum distance between the start time of the correlated event and the end time of the current event.

Report a bug

12.12.2.13. Starts

The **starts** operator correlates two events and matches when they start at the same time, but the current event ends before the correlated event ends.

For example:

```
$eventA : EventA( this starts $eventB )
```

This pattern matches if **$eventA** and **$eventB** start at the same time, and **$eventA** ends before **$eventB** ends.

This can be represented as follows:

```
$eventA.startTimestamp == $eventB.startTimestamp &&
   $eventA.endTimestamp < $eventB.endTimestamp
```

The **starts** operator accepts one optional parameter. If defined, it determines the maximum distance between the start times of events in order for the operator to still match:

```
$eventA : EventA( this starts[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 5s &&
   $eventA.endTimestamp < $eventB.endTimestamp
```

WARNING

The **starts** operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

12.12.2.14. Started By

The **startedby** operator correlates two events. It matches when both events start at the same time and the correlating event ends before the current event.

For example:

```
$eventA : EventA( this startedby $eventB )
```

This pattern matches if **$eventA** and **$eventB** start at the same time, and **$eventB** ends before **$eventA** ends.

This can be represented as follows:

```
$eventA.startTimestamp == $eventB.startTimestamp &&
   $eventA.endTimestamp > $eventB.endTimestamp
```

The **startedby** operator accepts one optional parameter. If defined, it sets the maximum distance between the start time of the two events in order for the operator to still match:

```
$eventA : EventA( this starts[ 5s ] $eventB )
```

This pattern matches if these conditions are met:

```
abs( $eventA.startTimestamp - $eventB.startTimestamp ) <= 5s &&
   $eventA.endTimestamp > $eventB.endTimestamp
```

WARNING

The `startsby` operator does not accept negative intervals, and the rules engine will throw an exception if an attempt is made to use negative distance intervals.

Report a bug

12.13. SLIDING WINDOWS

12.13.1. Sliding Time Windows

Stream mode allows events to be matched over a sliding time window. A *sliding window* is a time period that stretches back in time from the present. For instance, a sliding window of two minutes includes any events that have occurred in the past two minutes. As events fall out of the sliding time window (in this case because they occurred more than two minutes ago), they will no longer match against rules using this particular sliding window.

For example:

```
StockTick() over window:time( 2m )
```

JBoss BRMS Complex Event Processing uses the **`over`** keyword to associate windows with patterns.

Sliding time windows can also be used to calculate averages and over time. For instance, a rule could be written that states, "If the average temperature reading for the last ten minutes goes above a certain point, sound the alarm."

Example 12.15. Average Value over Time

```
rule "Sound the alarm in case temperature rises above threshold"
when
    TemperatureThreshold( $max : max )
    Number( doubleValue > $max ) from accumulate(
        SensorReading( $temp : temperature ) over window:time( 10m ),
        average( $temp ) )
then
    // sound the alarm
end
```

The engine will automatically discard any **`SensorReading`** more than ten minutes old and keep re-calculating the average.

Report a bug

12.13.2. Sliding Length Windows

Similar to Time Windows, Sliding Length Windows work in the same manner; however, they consider events based on order of their insertion into the session instead of flow of time.

The pattern below demonstrates this order by only considering the last 10 RHT Stock Ticks independent of how old they are. Unlike the previous StockTick from the Sliding Time Windows pattern, this pattern uses window:length.

```
StockTick( company == "RHT" ) over window:length( 10 )
```

The example below portrays window length instead of window time; that is, it allows the user to sound an alarm in case the average temperature over the last 100 readings from a sensor is above the threshold value.

Example 12.16. Average Value over Length

```
rule "Sound the alarm in case temperature rises above threshold"
when
    TemperatureThreshold( $max : max )
    Number( doubleValue > $max ) from accumulate(
        SensorReading( $temp : temperature ) over window:length( 100 ),
        average( $temp ) )
then
    // sound the alarm
end
```

NOTE

The engine disregards events that fall off a window when calculating that window, but it does not remove the event from the session based on that condition alone as there might be other rules that depend on that event.

NOTE

Length based windows do not define temporal constraints for event expiration from the session, and the engine will not consider them. If events have no other rules defining temporal constraints and no explicit expiration policy, the engine will keep them in the session indefinitely.

Report a bug

12.14. MEMORY MANAGEMENT FOR EVENTS

12.14.1. Memory Management for Events

Automatic memory management for events is available when running the rules engine in Stream mode. Events that no longer match any rule due to their temporal constraints can be safely retracted from the session by the rules engine without any side effects, releasing any resources held by the retracted events.

The rules engine has two ways of determining if an event is still of interest:

Explicitly

Event expiration can be explicitly set with the @expires

Implicitly

The rules engine can analyze the temporal constraints in rules to determine the window of interest for events.

Report a bug

12.14.2. Explicit Expiration

Explicit expiration is set with a **declare** statement and the metadata @expires tag.

For example:

Example 12.17. Declaring Explicit Expiration

```
declare StockTick
       @expires( 30m )
   end
```

Declaring expiration against an event-type will, in the above example **StockTick** events, remove any StockTick events from the session automatically after the defined expiration time if no rules still need the events.

Report a bug

12.14.3. Inferred Expiration

The rules engine can calculate the expiration offset for a given event implicitly by analyzing the temporal constraints in the rules.

For example:

Example 12.18. A Rule with Temporal Constraints

```
rule "correlate orders"
   when
       $bo : BuyOrder( $id : id )
       $ae : AckOrder( id == $id, this after[0,10s] $bo )
   then
       // do something
   end
```

For the example rule, the rules engine automatically calculates that whenever a **BuyOrder** event occurs it needs to store the event for up to ten seconds to wait for the matching **AckOrder** event, making the implicit expiration offset for **BuyOrder** events ten seconds. An **AckOrder** event can only match an existing **BuyOrder** event making its implicit expiration offset zero seconds.

The engine analyzes the entire rule-base to find the offset for every event-type. Whenever an implicit

expiration clashes with an explicit expiration the engine uses the greater value of the two.

Report a bug

APPENDIX A. REVISION HISTORY

Revision 1.0.0-10 **Mon Jun 30 2014** **Vikram Goyal**

Built from Content Specification: 22693, Revision: 662424 by vigoyal