Drools Usage Manual

version 2.0-beta-12
generated November 22, 2003

The Drools Project

drools.org

Contents

List of Figures

Chapter 1

Functional Overview

1.1 Rules

1.1.1 Declarative Form

Drools directly supports *declarative* rules, as opposed to *procedural* logic. Declarative rules typically take the form as follows:

> **if** *condition*
> **then** *consequence*

By "declarative', it is meant that the rules *declare*, by way of the *condition*, what should occur but do not specify the procedure for actually testing the conditions. For example, a procedural method for ensuring you have an umbrella if it is raining would be:

> Step outside and determine if it is raining. If it is raining, then go to the closet and get an umbrella.

In a declarative form, the above could be represented by two rules:

- **if** *it is raining*
  then *you need an umbrella*
- **if** *you need an umbrella*
  then *get one from the closet*

Given declarative rules, the knowledge that it is raining could produce two courses of action:

1. You already have an umbrella, perhaps because you always carry one, in which case, you're ready.
2. You don't have an umbrella, so you go get one from the closet.

1.1.2 Applicable Context

The set of rules to be considered at any point of time depend upon the current context. As a human, you have certain rules to think about when you dine at a fine restaurant, which are exclusive to the set of rules you consider when spending a sunny day at the swimming pool. Even so, there are some other overriding rules that may be important regardless of the context, such as laws against homocide.

So, in a given context, different sets of rules are pertinent. A single set of rules may be user in one context, while a different set is used in a different context. Within a rule-engine, available sets of rules are called *rule sets*. The set or sets of rules currently applicable given the context is called a *rule base*.

1.2 Knowledge

You become aware of knowledge, in the form of facts, over time. Likewise, over time, facts may change or cease to be true, in which case, they facts as you know them must be altered or purged from your memory. Likewise, within a rule-engine, there are operations for becoming aware of a fact, purging a fact, or modifying a known fact. These operations are as follow:

- **Assert** Add a fact to what is known.
  The weather is rainy. Betty is in the room.
- **Retract** Remove a fact from what is known.
  Removing the fact *Betty is in the room* when Betty leave the room.
- **Modify** Alter a fact from what is know.
  Changing the knowledge about the weather from *The weather is rainy* to *The weather is sunny* when the rain stops and the clouds go away.

Within a rule-engine, the collected knowledge is called the *working memory*. Knowledge is asserted, retracted and modified within the working memory and the rules are evaluated to determine what actions, if any, should be taken.

1.3 Why a rule-engine?

While the logic expressed in a rule can and has often been writen within the code a system, a rule-engine offers many benefits. Instead of locking the logic up in code written by developers, the logic can be moved out-board external to the actual application. In this way it is possible for non-developers to change the logic without having to rebuild the system. Additionally, by codifying all of the system rules in a central location, they are no longer scattered throughout the application. This allows for easier validation of the system's requirements and analysis of the logic of the system.

Additionally, a rule-engine such as Drools is built upon an intelligent algorithm that allows for the evaluation of many rules against many facts in an effecient manner. In a procedural system, a change in a single fact might require double-checking *every* rule to determine if any action needs to be taken. A rule-engine which uses the *Rete* algorithm is optimized to minimize the amount of processing effort that is required to evaluate the rules that may have been affected by a change in knowledge.

1.4 A note about "business rules"

A higher-level form of rule is the *business rule*. Business rules do not necessarily follow the *if-then* form, but may be specified in different formats that are not as closely linked to the underlying rule-engine implementation. Business rules tend to use the *must* or *must not* form to expression constraints or inferences.

- An order **must not** be billed before it ships.
- An applicant for store credit **must** be 18 years of age.

Drools does not directly support this level of business rules, but other projects built upon Drools[1] may easily support such notation.

[1] The **Fluxtapose** suite of tools from The Werken Company is one such product that supports business rules.

Chapter 2

Architecture

2.1 Rules, rule-sets, and rule-bases

Within Drools, the concepts of rules, rule-sets and rule-bases are directly modelled by the classes `Rule`, `RuleSet` and `RuleBase`. A `Rule` may be a member of multiple `RuleSet`s, and multiple `RuleSet`s may be active within a given `RuleBase`.

2.1.1 `Rule`, `Condition` and `Consequence`

A single `Rule` may have one-or-more `Condition`s associated with it. Each condition must be met before the rule is considered to be activated. Once activated the rule's `Consequence` is a candidate for being fired. In pattern parlance, the `Condition` class is simply a *predicate object* which evaluates itself against the known facts to return a boolean value of either *true* or *false*. The `Consequence` class is likewise simply a *functor* which objectifies a function and performs an arbitrary task when executed.

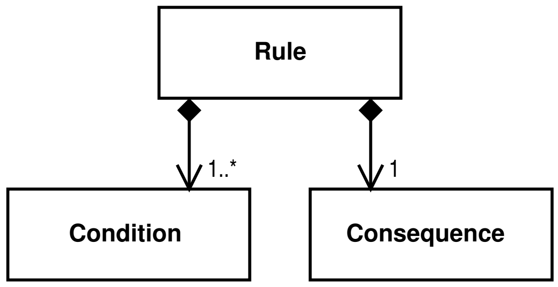

Figure 2.1: Object model for `Rule`, `Condition` and `Consequence`

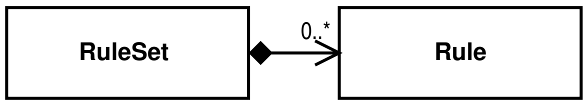

Figure 2.2: Object model for `RuleSet` and `Rule`

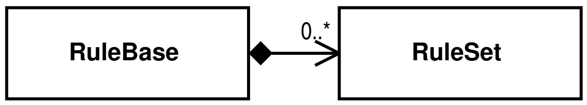

Figure 2.3: Object model for `RuleBase` and `RuleSet`

2.1.2 `RuleSet`

A `RuleSet` is simply a collection of `Rules`. It serves only to associate a group of rules with one another so that they may be worked with as a set.

2.1.3 `RuleBase`

A `RuleBase` is an *active* collection of `RuleSets`. A `RuleBase` contains rules that are all considered to be in effect for a given set of knowledge. Multiple `RuleSets` may be a part of a given `RuleBase`.

2.2 Knowledge

The set of knowledge that is examined is modelled by the class `WorkingMemory`, which is backed by a particular `RuleBase`. It is through the `WorkingMemory` that knowledge is *asserted*, *retracted* and *modified*. Each `WorkingMemory` is backed by exactly one `RuleBase` which determines which rules are evaluated as knowledge is manipulated.

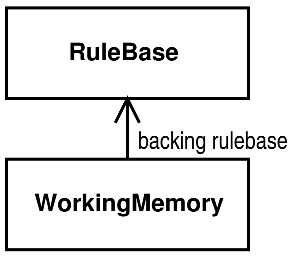

Figure 2.4: Object model for `WorkingMemory` and `RuleBase`

2.3 Complete Model

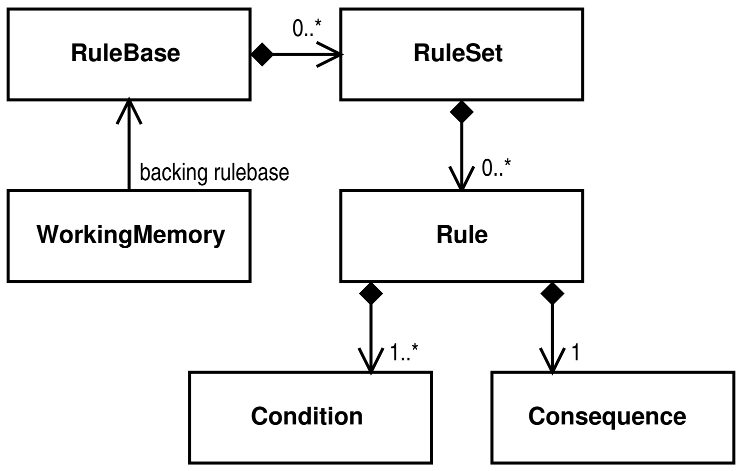

Figure 2.5: Complete object model

Chapter 3

Drools Rule Language

3.1 Introduction

The Drools Rule Language (DRL) is an XML-based language that uses an extensible tag mechanism. The available tags can be extended through the use of *semantic modules*. By itself the DRL is not fully usable. At least one semantic module must be selected and used. What follows is a reference for the available core DRL tags.

3.2 Core DRL namespace

The core DRL tags belong to the `http://drools.org/rules` namespace. In order to use the tags, the root tag must include an `xmlns` declaration binding the DRL namespace to either a prefix (Figure 3.1) or as the default namespace (Figure 3.2). Common practice is to bind the DRL namespace URI to the default namespace upon the root `<rule-set>` tag of a document.

```
<drl:rule-set xmlns:drl="http://drools.org/rules">
```

Figure 3.1: Binding the DRL namespace URI to a prefix

```
<rule-set xmlns="http://drools.org/rules">
```

Figure 3.2: Binding the DRL namespace URI as the default namespace

3.3 Core DRL tags

3.3.1 `<rule-set>`

The root tag of a DRL document is the `<rule-set>` tag, which serves to collect a group of rules. A rule-set must have a `name` attribute defining its name.

`<rule-set>`

Attribute	Description
`name`	Name of the rule-set.

Tag	Description
`<rule>`	One or more rule members of the set.

```
<rule-set xmlns="http://drools.org/rules"
          name="My First Rule Set">
    ...
</rule-set>
```

3.3.2 `<rule>`

A `<rule>` tag defines a single rule within the rule-set. A rule must have at least one parameter[1]. A rule contains conditions, extractors and exactly one consequence.

`<rule>`

Attribute	Description
`name`	Name of the rule.

Tag	Description
`<parameter>`	Parameter declaration.
`<declaration>`	Additional local declaration.
`<ns:condition>`	Semantic condition.
`<ns:extractor>`	Semantic extractor.
`<ns:consequence>`	Semantic consequence.
`<duration>`	Truthness duration.

[1]Semantic modules may add implicit parameters to rules, so it's not strictly true that each `<rule>` must have a `<parameter>` or a `<consequence>`

```
<rule-set xmlns="http://drools.org/rules"
          name="My First Rule Set">

    <rule name="My First Rule">
        ...
    </rule>

</rule-set>
```

3.3.3 `<parameter>`

A `<parameter>` defines an input parameter for a rule. Each input parameter will be supplied an object from the working memory in order to attempt satisfying the conditions. Each `<parameter>` is required to have an `identifier` that is unique within the rule and a semantic object type as its child element.

`<parameter>`

Attribute	Description
`identifier`	Parameter identifier.

Tag	Description
`<ns:objectType>`	Object type of the parameter.

```
<rule name="My First Rule">

    <parameter identifier="factOne">
        ...
    </parameter>
    <parameter identifier="factTwo">
        ...
    </parameter>

</rule>
```

3.3.4 `<declaration>`

A `<declaration>` defines an addition *local* declared fact which may be the target of an *extractor*. Its `identifier` must be unique across all `<parameter>` and `<declaration>` tags. As with a `<parameter>`, it must have a semantic object type as its child element.

`<declaration>`

Attribute	Description
`identifier`	Declaration identifier.

Tag	Description
`<ns:objectType>`	Object type of the local declaration.

```
<rule name="My First Rule">

    <declaration identifier="factOne">
        ...
    </declaration>
    <declaration identifier="factTwo">
        ...
    </declaration>

</rule>
```

3.3.5 `<duration>`

The `<duration>` tag is optionally used to specify the truthness duration of the rule. The truthness duration determines how long the rule's conditions must be *continually* true in order to activate the rule.

`<declaration>`

Attribute	Description
`weeks`	Number of weeks.
`days`	Number of days.
`hours`	Number of hours.
`minutes`	Number of minutes.
`seconds`	Number of seconds.

```
<rule name="My First Rule">

    <duration hours="2"
              minutes="30"/>

</rule>
```

Chapter 4

Algorithms

4.1 Efficient Matching

While it may be simple to create a rules engine that allows specification of business logic in a format that is comfortable to business analysts, the matching of the rules may still be problematic without a good algorithm.

The rules engine must be made aware of its environment, typically through a process called *fact assertion.* Fact assertion consists of the program asserting facts into a rules session, or *working memory.*

Whenever a fact is asserted, retracted or modified within the working memory, many rules may become candidates for firing, or may have become invalidated. A simplistic approach is to reevaluate all rules against the entirety of the working memory. This method is guaranteed to be correct but will also certainly be sub-optimal. Any individual fact modification only affects a small number of conditions in a small number of rules.

Variations of the Rete algorithm allow the rules engine to maintain a memory of the results of partial rule matches across time. Reevaluation of each condition is no longer necessary, as the engine knows which conditions might possibly change for each fact, and only those must be reevaluated.

4.2 Rete

Charles Forgy created the original Rete algorithm [**?**] around 1982 as part of his DARPA-funded research. Compared to many previous production-matching algorithms, Rete was very advanced. Even today, there have been few improvements to it in the general case[1]. Variations on Rete, such as TREAT [**?**], may have different performance characteristics depending on the environment. Some perform better with large rule sets but small numbers of objects, while other

[1]Both ILOG and Haley claim to have optimized Rete algorithms, but details are not currently public.

perform well for steady-state environments, but react poorly to numerous successive changes in the data.

A *Rete network* is a graph through which data flows. Originally, data was specified using Cambridge-prefix tuples since Lisp-like languages were in style for logic programming.[2] The tuples were used to express attributes about objects. For example, tuples may be used to express a person's name and her pets. The tuples are dropped into the Rete network, and those that reach the far end cause the firing of a rule. The original production-matching was based upon matches against tuple patterns.

The Rete network is comprise of two types of nodes:

- **1-input/1-output nodes**
  The *1/1* nodes are constrictive nodes that only allow matching tuples to flow through. Any tuples that do not match are discarded by the node.
- **2-input/1-output nodes**
  The *2/1* nodes simply connect the output arcs from two other nodes (either *1/1* nodes or *2/1* nodes) merging tuples from both the left and right incoming arcs into a single tuple on the outgoing arc. Maintains a memory of tuples for matching against future facts.

A forest of *1/1* nodes acts as the entry-point into the entire Rete network for any incoming tuple. The network-entry nodes filter tuples purely by their type. Tuples about dogs and tuples about cats may each have a different type and may be differentiated from each other by the *1/1* network-entry nodes.

Each condition of a rule is merely a pattern for a particular tuple type. The condition describes the attributes that a tuple must have and acts as a filter. Each condition is transformed into a *1/1* node that only allows tuples matching the specified attributes to pass. An attribute value may be specified as a variable and implies that the variable must hold the same value in all occurrences. The *1/1* filter nodes are attached to the network downstream from the *1/1* entry-node that differentiates their tuple type.

Consider a condition such as "For any person who has a dog that has the same name as that person's sister's cat, then..." This could be expressed with the condition patterns of:

```
(1)  ( person  name=person?  sister=sister?  )
(2)  ( person  name=person?  dog=petName?  )
(3)  ( person  name=sister?  cat=petName?  )
```

Condition *#1* models the sister relationship so that the rule only applies to two people who are sisters. The `person?` and `sister?` tokens are variables that must be consistent across any set of tuples that match this rule.

Conditions *#2* and *#3* serve two roles. The `dog` and `cat` attributes share the same `petName?` variable and serve to identify two people who have a cat

[2] As it is for many artificial intelligence projects.

and a dog with the same name. They each contain a `name` attribute with either the variable `person?` or `sister?` which ties the last two conditions back to the first two.

$$\begin{tikzcd}
\bullet \arrow[d] & & \\
type(person) \arrow[d] \arrow[dr] \arrow[drr] & & \\
condition(1) \arrow[dr] & condition(2) \arrow[d] & condition(3) \arrow[dd] \\
& join(1) \arrow[dr] & \\
& & join(2) \arrow[d] \\
& & terminal
\end{tikzcd}$$

Figure 4.1: Rete network

type	person	sister	cat	dog
tuple set # 1				
person	rebecca	jeannie	zoomie	*null*
person	jeannie	rebecca	*null*	zoomie
tuple set # 2				
person	rebecca	jeannie	zoomie	*null*
person	jeannie	rebecca	*null*	toby

Figure 4.2: Example tuple sets

If two sets of tuples (see Figure 4.2) were asserted against the rule, *tuple set #1* would cause a firing of the rule, where *tuple set #2* would not. In both cases, the two tuples would pass node *condition(1)*, as the nodes simply associate the `person?` and `sister?` variables with the appropriate values from each tuple.

The *join(1)* node would allow both tuples to merge and propagate past it in both the first and second case. Additionally, for both cases, the *rebecca* tuple would pass node *condition(2)* and the *jeannie* tuple would pass node *condition(3)*.

The *join(2)* node is where the two cases differ. In the first case, nodes *condition(2)* and *condition(3)* have each associated the value of “ugly” to the `petName?` variable. In the second case, the two nodes has assigned different values to the variable. The *join(2)* node only allows those tuples that have consistent associations with all variables to pass.

4.3 Rete-OO

The Rete algorithm works wonderfully in language systems such as Lisp where pertinent attributes about objects are directly asserted to the rules engine. In an object-oriented language, such as C++ or Java, and entire graph of objects can be reachable from a single named root object. Expressing highly complex relationships between entities using Cambridge-prefix notation may require many separate assertions. In an OO language, the single root object is all that should be asserted, since attributes and relationships can be *extracted* using normal language constructs.

Bob McWhirter of The Werken Company adapted Forgy’s original Rete algorithm to object-oriented constructs, creating the Rete-OO algorithm. As with Rete, there are *1/1* nodes and *2/1* nodes. Unlike Rete, there are nodes that exist simply to extract reachable attributes and add columns to passing tuples. Rete always constructs the condition *1/1* nodes toward the root of the tree leaving the bottom portion to be comprised of purely aggregating *2/1 join* nodes. Rete-OO must interleave both *1/1* and *2/1* nodes.

The same example as in section 4.2, the conditions could be expressed in terms of object-oriented language boolean and assignment expressions. The choice of Java as the expression language is purely arbitrary.

(0) `Person personOne, personTwo`
(1) `personOne.hasSister(personTwo)`
(3) `petName = personOne.getCat().getName()`
(3) `petName = personTwo.getDog().getName()`

Rete-OO adds the concept of *root object declaration*, where the root objects of the condition are declared with a name and type. The object’s type maps directly to the tuple type in Rete. The root object name has no direct mapping in Rete and causes the addition of a *parameter node* in Rete-OO. Boolean expressions in Rete-OO conditions are equivalent to Rete’s condition patterns

$$\begin{tikzcd}
\bullet \arrow[d] & \\
type(Person) \arrow[d] \arrow[dr] & \\
parameter(personOne) \arrow[d] & parameter(personTwo) \arrow[d] \\
extraction(petName) \arrow[dr] & extraction(petName) \arrow[d] \\
& join \arrow[d] \\
& condition(hasSister) \arrow[d] \\
& terminal
\end{tikzcd}$$

Figure 4.3: Rete-OO network

against attributes. The assignment expressions map to place-holder variables in Forgy's algorithm.

The types of nodes used in Rete-OO graph construction are listed here. Those that are new or different from Rete are denoted with a '*'.

- **Object type**
  Object type nodes differentiate objects by filtering on their defined type.
- **Parameter***
  Parameter nodes create a tuple with a single entry binding the object to the name.
- **Condition**
  Condition nodes simply tests a tuple against an a boolean expression.
- **Extraction***
  Extraction nodes extract new attributes, create new columns on tuples, and store the results.
- **Join**
  Join nodes connect the output arcs from two other nodes and allows consistent tuples to be merged and passed through.
- **Terminal**
  Terminal nodes fire to indicate a successful match for the rule.

The resulting Rete-OO graph is constructed in a different manner than the equivalent Rete graph, due to the addition and rearrangement of some nodes.

Appendix A

Project Information

A.1 Web Site

All development resources related to Drools are hosted by **The Codehaus**, the open-source arm of The Werken Company. Drools maintains a website at:

```
    http://drools.org/
```

A.2 Mailing Lists

The drools project maintains two mailing lists. The first, known as `drools-interest` is for general discussion by users and developers of drools. The second list is `drools-cvs` which simply tracks changes made to the source-code through the CVS repository. For information about subscribing to each list or access to the list archives:

```
    http://lists.codehaus.org/listinfo/drools-interest
    http://lists.codehaus.org/listinfo/drools-cvs
```

A.3 Source Repository

The drools project maintains a revision control repository using CVS. To checkout the latest sources, you must issue two CVS commands. The first is used to login. When presented with a prompt for a password, simply press *ENTER*.

```
  cvs -d:pserver:anonymous@cvs.codehaus.org:/scm/cvspublic login
  cvs -d:pserver:anonymous@cvs.codehaus.org:/scm/cvspublic co drools
```

A.4 Internet Relay Chat

There is a dedicated channel on The Werken Company's IRC server for drools:

address	`irc.codehaus.org`
port	`6667`
channel	`#drools`
url	`irc://irc.codehaus.org:6667/drools`

A.5 Bug, Issue & Feature Tracking

For bug, issue and feature tracking, the Drools project uses the Jira project management system provided by The Codehaus.

`http://jira.codehaus.org/`

A.6 Project Team

A.6.1 Bob McWhirter

Bob McWhirter originally founded the Drools project in 2000 and developed the Rete-OO algorithm used by the engine. Bob is also the founder of The Werken Company and the chief architect behind the commercial **Fluxtapose** suite of tools which build upon Drools to provide a complete solution for implementing business rules.

A.6.2 Thomas Diesler

Thomas Diesler researched and supplied the JSR-94 Rule-Engine API bindings for Drools.

A.6.3 Roger F. Gay

Roger F. Gay devised the XML Schemas for the core DRL syntax and each semantic module.

A.6.4 Contributors

Others have contributed ideas, patches and testing assistance over the years:

- Dave Cramer *(eBox)*
- Martin Hald
- Matt Ho
- Pete Kazmier *(iBasis)*
- Christiaan ten Klooster
- James Roome
- Bart Selders *(iBanx)*
- James Strachan *(CoreDevelopers Network)*
- Tom Vasak

Appendix B

Licensing

B.1 Drools License

B.1.1 The License

B.1.2 Summary

Drools is provided under a license similar to that used by The Apache Software Foundation. It is a commercial-friendly license in that it allows you to modify and distribute Drools in either source or binary form. While you are *encouraged* to contribute changes back to the project, you are by no means *required* to do so. The Drools license is not viral or infectious. It does not alter how you license you own product. If you have any questions regarding the licensing of Drools, please contact `info@werken.com`.