The drools Guide

Theory, Usage and Reference

The Werken Company
www.werken.com

July 4, 2003

Contents

List of Figures

Chapter 1

Introduction

1.1 Rules

Many enterprise software systems today already include the concept of rules. Most times, these rules are directly implemented in code and are difficult to adapt to a changing business landscape. The domain model many times includes business logic which may change often. When these business 'rules' are coded using normal systems programming techniques,[1] modification and maintenance of the logic can become difficult.

Examples of typical simple business rules include:

- *When* a customer applies for a loan of more than $80,000 and has less than $5,000 in their savings account and has had the account for less than 3 years, *then* reject the loan application.
- *When* a customer orders a box of goods and a routed delivery vehicle can include the delivery in today's route by lengthening the route by no more than 8 miles, *then* add the customer's delivery to the vehicle.
- *When* a trouble-ticket from a high-priority customer has been unresolved for 60 minutes, *then* escalate the urgency of the ticket and notify the shift manager.
- *When* a customer buys pie, *then* suggest that he might enjoy some ice-cream.

More complex rules that involve multiple participants or chunks of data may also exist in systems:

- *When* someone is selling a product desired by another person for a price less-than-or-equal-to the price the other is is willing to pay, *then* notify both parties.

[1] Many times, a long sequence of `if/else` statements is used to realize business rules.

- *When* an author submits an article and either three junior editors a single senior editor has signed off on it, *then* send the article to the production department work queue.

- *When* email which is not directly addressed to me arrives and the sender is not on my 'approved' list, *then* direct it to my mail folder that holds potential spam.

As companies are often changing the way they do business, responding quickly to changes is important. Changing business logic which is realized in compiled code can become quite an arduous job. Systems that respond to changes in policy as quickly as the enterprise makes decisions reduce maintenance costs and development cycle times.

1.2 Rules Engines

Rules engines were developed to make the creation and maintenance of collections of rules easier and less costly. Through the use of intelligent algorithms (see Chapter 10 on page 35), some dedicated rules engine can also produce efficiencies when working with a large amount of rules, events and data.

A good rules engine allows the business logic of a system to be specified external to the system itself. No longer must these rules be codified by the developers. Many rules engines even provide natural-language or wizard-style GUIs for designing rules, allowing product managers or business analysts to actually specify the logic. Separation of concerns and responsibility is achieved by moving the business rule specification outside of the actual program logic. Developers can concern themselves with systems engineering while the analysts concentrate on the business logic.

There are currently many commercial and open-source implementations of rules engines aside from drools. The commercial vendors have undoubtedly good algorithms and have spent considerable time and effort on the user interfaces and rule specification languages.

- **ILOG JRules**
  `http://www.ilog.com/`

- **Haley Eclipse**
  `http://www.haley.com/`

- **Sandia Jess**
  `http://herzberg.ca.sandia.gov/jess/`

- **CLIPS**
  `http://www.ghg.net/clips/CLIPS.html`

1.3 Standards

1.3.1 RuleML

RuleML is a mark-up language for describing rules. After brief research, we decided that supporting RuleML was not a priority. RuleML tends to forcus on *inference rules* and not the *event-condition-action* or *trigger* rules that are drools's primary focus.

1.3.2 JSR-94

JSR-94 is the specification working its way through the Java Community Process toward defining a common API for rules engines. A draft version for community review has recently been released. The drools project aims to be JSR-94 compliant in the near future.

Chapter 2

Usage

2.1 drools Client API

2.1.1 Introduction

drools is divided into several sets of APIs. The core client API is by far the simplist and most commonly used by developers. The core drools client API consists of locating a rule-base, creating a working memory, and then managing fact assertion, modifcation and retraction.

2.1.2 Locating a Rule-base

`RuleBase` objects are typically loaded from a `RuleBaseRepository`. Different repository implementations provide different mechanisms for the storage of each `RuleBase`. The method by which your project obtains a `RuleBaseRepository` is implementation-specific but may involve a lookup and discovery mechanism such as JNDI.

Once a `RuleBaseRepository` has been obtained, the simple method `lookupRuleBase(..)` method is used to retrieve a `RuleBase` by its URI.[1]

```
RuleBaseRepository repo = myUtilities.getRepository();

String ruleBaseUri = "http://rules.werken.com/family-relationships";

RuleBase ruleBase = repo.lookupRuleBase( ruleBaseUri );
```

[1] URIs that identify rule bases are not necessarily deferenceable. They serve only as unique identifiers for a collection of rules.

2.1.3 Creating a Working Memory

drools provides two different types of working memory implementations: a normal `WorkingMemory` and a `TransactionWorkingMemory`.

- **WorkingMemory**
  The normal `WorkingMemory` implementation propagates fact assertions, modifications and retractions through the Rete-OO network in real-time. Once the fact manipulation methods return control to the client program, all facts have been assimilated and acted upon.

- **TransactionalWorkingMemory**
  The `TransactionalWorkingMemory` does *not* propagate fact manipulation information through the Rete-OO in real-time. Instead, it calculates the net fact changes and performs all manipulations immediately upon usage of the `commit()` method. No actions are performed until `commit()` is called, and all fact information is discarded if `abort()` is used.

The methods on `RuleBase` to construct working memories are:

```
/** Create a WorkingMemory session for this RuleBase.
 *
 *  @see WorkingMemory
 *
 *  @return A newly initialized WorkingMemory.
 */
public WorkingMemory createWorkingMemory()

/** Create a TransactionalWorkingMemory session for this RuleBase.
 *
 *  @see TransactionalWorkingMemory
 *
 *  @return A newly initialized TransactionalWorkingMemory.
 */
public TransactionalWorkingMemory createTransactionalWorkingMemory()
```

2.1.4 Fact Manipulation

Once you have a `WorkingMemory` in hand, you must assert fact objects to make them available to drools for analysis. Additionally, as facts change, the engine must be notified. Likewise, when an object should no longer be considered for analysis, it must be retracted from the engine. Methods for these three actions are defined upon the `WorkingMemory` class.

```
/** Assert a new fact object into this working memory.
*
*  @param object The object to assert.
*
*  @throws AssertionException if an error occurs during assertion.
*/
public void assertObject(Object object) throws AssertionException
```

```
/** Modify a fact object in this working memory.
*
*  With the exception of time-based nodes, modification of
*  a fact object is semantically equivelent to retracting and
*  re-asserting it.
*
*  @param object The object to modify.
*
*  @throws FactException if an error occurs during modification.
*/
public void modifyObject(Object object) throws FactException

/** Retract a fact object from this working memory.
*
*  @param object The object to retract.
*
*  @throws RetractionException if an error occurs during retraction.
*/
public void retractObject(Object object) throws RetractionException
```

2.2 JSR-94 API

The drools project is currently working on a JSR-94 API binding.

Chapter 3

Drools Rule Langauge

3.1 Introduction

drools defines a semantic-module-independant rule language created called the *Drools Rule Language*. DRL is a an XML-based language that uses modern XML features such as XML-Namespaces and XML Schema. The DRL engine within drools is built on top of jakarta-commons-jelly, which is a general XML tag library engine.

3.2 DRL Files

DRL files are XML files using the DRL tags. They are typically files that have the *.drl* suffix. drools only requires that they be accessible through a URL:

- **Local Filesystem**
  DRL files can be stored in the local filesystem and accessed using `file://` URLs.
- **Web/FTP Server**
  DRL files can be stored on the network and accessed using `http://` and `ftp://` URLs.
- **Java Classpath**
  DRL files can be stored in the Java classpath or in a JAR file, and accessed using the `getResource()` method on `java.lang.ClassLoader` and `java.lang.Class` classes.

3.3 Loading DRL Files

A `RuleSetLoader` is provided for loading DRL files into a `RuleBase`. Given a URL, the `RuleSetLoader` will retrieve the DRL file and load all rules and

rule-sets into the specified `RuleBase` which can then immediately be used for knowledge manipulation.

```
import org.drools.RuleBase;
import org.drools.WorkingMemory;
import org.drools.io.RuleSetLoader;
...

    RuleBase ruleBase = new RuleBase();

    RuleSetLoader loader = new RuleSetLoader();

    loader.load( rulesUrl, ruleBase );

    WorkingMemory memory = ruleBase.createWorkingMemory();

    memory.assertObject( account );
```

3.4 Base DRL Syntax

The base syntax for DRL contains a small handful of tags representing the general structure of rules and rule-sets. These tags are defined for the XML namespace URI of `http://drools.org/rules`.

- `<rules>`
  General outter-level wrapper tag.
- `<rule-set>`
  A named collection of rules.
- `<rule>`
  A single rule.
- `<parameter>`
  A root fact-object parameter.
- `<declaration>`
  A local variable declaration.
- `<extraction>`
  A fact extraction.
- `<condition>`
  A filtering condition.
- `<duration>`
  The match duration.
- `<consequence>`
  The rule match consequence.
- `<semantics>`
  Load a semantic module.

3.4.1 drl:rules

The outtermost tag in each DRL file is the `<rules>` tag. It has no attributes and serves only to aggregate `<rule-set>`s and `<rule>`s. The XML namespace declaration for the base DRL should be affixed to this element.

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

3.4.2 drl:rule-set

The `<rule-set>` is a named container for `<rules>`. Its only attribute is `name` to provide for a name.

```
<drl:rule-set name="Gold-Level Member Rules">
  <drl:rule ...>
    ...
  </drl:rule>
</drl:rule-set>
```

3.4.3 drl:rule

The `<rule>` tag is the most complex. It *must* contain at least one `<parameter>` and a `<consequence>` tag. It may optionally contain `<declaration>`, `<extraction>`, `<condition>` and `<duration>` tags. The `name` attribute must be present. A `<rule>` may exist inside either a `<rules>` or `<rule-set>` tag.

```
<drl:rule name="Over Credit Limit">
  <drl:parameter ..>
    ...
  </drl:parameter>
  <drl:declaration ..>
    ...
  </drl:declaration>
  <drl:extraction ..>
    ...
  </drl:extraction>
  <drl:condition ..>
    ...
  </drl:condition>
  <drl:duration ..>
    ...
  </drl:duration>
  <drl:consequence ..>
    ...
  </drl:consequence>
</drl:rule>
```

3.4.4 drl:parameter

The <parameter> tag defines a root fact object that the rule expects to be provided from external resources. The only attribute is identifier which provides the variable identifier to be used to refer to the object elsewhere in the rule. The content of the tag is dependent upon the semantic module used for the rule. For illustration purposes, the Java Semantic Module has been used.

```
<parameter identifier="customer">
  <java:class type="com.werken.Customer"/>
</parameter>
<parameter identifier="account">
  <java:class type="com.werken.Account"/>
</parameter>
```

3.4.5 drl:declaration

A <declaration> tag is similar to a <parameter> in that it defines a typed and named object. It must contain an identifier attribute to specify the name that may be used to refer to the declared object elsewhere in the rule. This tag declares a variable that must be populated internally using an <extraction>. For illustration purposes, the Java Semantic Module has been used.

```
<drl:declaration identifier="custName">
  <java:class type="java.lang.String"/>
</drl:declaration>
<drl:declaration identifier="acctBalance">
  <java:class type="java.math.BigInteger"/>
</drl:declaration>
```

3.4.6 drl:extraction

An <extraction> defines a fact extraction. Its only attribute is target which names the parameter or declaration that the extracted fact should be assigned to. The content of the tag is dependent upon the semantic module used for the rule. For illustration purposes, the Java Semantic Module has been used.

```
<drl:extraction target="customer">
  <java:extractor>account.getCustomer()</java:extractor>
</drl:extraction>
<drl:extraction target="custName">
  <java:extractor>customer.getName()</java:extractor>
</drl:extraction>
<drl:extraction target="acctBalance">
  <java:extractor>account.getBalance()</java:extractor>
</drl:extraction>
```

3.4.7 drl:condition

The `<condition>` defines a condition that must be met in order for the rule to match. It contains the main logic of the rule. The content of the tag is dependent upon the semantic module used for the rule. For illustration purposes, the Java Semantic Module has been used.

```
<drl:condition>
  <java:condition>custName.equals( "McWhirter" )</java:condition>
</drl:condition>
<drl:condition>
  <java:condition>acctBalance.signum() == 0</java:condition>
</drl:condition>>
```

3.4.8 drl:duration

The `<duration>` tag is used to specify a temporal condition. The truth duration of a rule is the amount of time that all other conditions must hold true before a match is determined. The content of the tag is dependent upon the semantic module used for the rule. A simple `<fixed-duration>` tag is supplied as part of the base DRL syntax in order to specify static durations that are not dependent upon rule data.

```
<drl:duration>
  <drl:fixed-duration seconds=".."
                      minutes=".."
                      hours=".."
                      days=".."
                      weeks=".."/>
</drl:duration>
```

3.4.9 drl:consequence

The `<consequence>` tag defines the action to be taken once a rule matches for a set of root fact objects. The content of the tag is dependent upon the semantic module used for the rule. For illustration purposes, the Java Semantic Module has been used.

```
<drl:consequence>
  <java:consequence>
    account.addMoney( new BigInteger( "1000000" ) );
  </java:consequence>
</drl:consequence>
```

3.4.10 drl:semantics

The `<semantics>` tag is used to load a semantic module. It has no content and the `module` attribute is required in order to identify a semantic module to load.

```
<drl:semantics module="org.drools.semantics.java"/>
```

Chapter 4

Java Semantic Module

4.1 Overview

The Java Semantic Module provides implementations of semantic components that adhere to the Java language semantics. The components can be used directly from Java through a DRL file.

- **org.drools.semantics.java.ClassObjectType**
  A `ObjectType` implementation that adheres to Java class types. Usable within `<parameter>` and `<declaration>` DRL tags.
- **org.drools.semantics.java.ExprCondition**
  A `Condition` implementation that uses boolean Java expressions for filtering. Usable within `<condition>` DRL tags.
- **org.drools.semantics.java.ExprExtractor**
  A `Extractor` implementation that uses Java expressions for extracting new facts. Usable within `<extraction>` DRL tags.
- **org.drools.semantics.java.BlockConsequence**
  A `Consequence` impementation that uses A block of Java statements as the action of a matched rule. Usable within `<consequence>` DRL tags.

4.2 Usage with DRL

4.2.1 Loading the Module

The Java Semantic Module's tags exist within the XML namespace URI of `http://drools.org/semantics/java` and within the Java package of `org.drools.semantics.java`. To use the Java Semantic Module within a DRL file, the DRL `<semantics>` tag must be used, as must an XML namespace prefix binding.

```
<drl:rules xmlns:drl="http://drools.org/rules"
       xmlns:java="http://drools.org/semantics/java">

  <drl:semantics module="org.drools.semantics.java"/>

  <drl:rule ...>
    <drl:parameter identifier="account">
      <java:class type="com.werken.Account"/>
    </drl:parameter>
  </drl:rule>

</drl:rules>
```

4.2.2 java:class

The `<java:class>` tag defines an *object type* that adheres to Java class semantics for types. It has a single attribute of `type` which takes a class name as a value. The tag may be used as the content of both `<drl:parameter>` and `<drl:declaration>` DRL tags.

```
<drl:parameter identifier="account">
  <java:class type="com.werken.Account"/>
</drl:parameter>
<drl:declaration identifier="person">
  <java:class type="com.werken.Person"/>
</drl:declaration>
```

4.2.3 java:extractor

The `<java:extractor>` tag defines a *fact extractor* that adheres to Java expression semantics. It has no attributes and the body content is the expression to generate the new fact. The tag may be used as the content of a `<drl:extraction>` tag.

```
<drl:extraction target="accountBalance">
  <java:extractor>person.getAccount().getBalance()</java:extractor>
</drl:extraction>
```

4.2.4 java:condition

The `<java:condition>` tag defines a *condition* that adheres to Java boolean expression semantics. It has no attributes and the body content is the boolean expression to evaluate. The tag may be used as the content of a `<drl:condition>` tag.

```
<drl:condition>
  <java:condition>acctBalance == 0</java:condition>
</drl:condition>
```

4.2.5 java:consequence

The `<java:consequence>` tag defines a *rule consequence* that adheres to Java statement block semantics. It may exist as the content of a `<drl:consequence>` tag. It has no attributes and the body content is the set of statements to execute upon rule match.

```
<drl:consequence>
  <java:consequence>
    System.err.println( "The balance is: " + acctBalance );
    theRepoMan.addAccount( account );
    assertObject( theRepoMan );
  </java:consequence>
</drl:consequence>
```

Chapter 5

Jython Semantic Module

. . . not implemented yet . . .

Chapter 6

XML Semantic Module

. . . not implemented yet . . .

Chapter 7

Rule Assembly

7.1 Overview

Only a handful of classes are required to assemble rules once a *semantic module* has been selected. Each rule is codified as an instance of the `Rule` class which may be a member of a `RuleSet` collection.

1. Instantiate a `Rule`.
2. Add a `Declaration` for each root fact object.
3. Add a `Extraction` for each fact extraction.
4. Add a `Condition` for each restrictive condition.
5. Add a `Consequence` for performing the result of a match.
6. Add the `Rule` to a `RuleBase`.
7. Optionally register the `RuleBase` with a `RuleBaseRepository`

When adding a `Rule` to a `RuleBase`, it is possible that the rule cannot be integrated into the network. This is caused by `Extraction` or `Condition` objects that expect `Declarations` that are otherwise not present in the rule.

7.2 Rule Assembly Example

Rules may be assembled using classes from the `org.drools.rule` package along with one or more semantic modules. Additional tools to allow for assembling rules from a file or database are possible.

```
// -- Create a new Rule

Rule rule = new Rule("example");

// -- Create the semantic Person object type
// -- which maps directly to java Person type.

ObjectType personType = new ObjectType() {
        public boolean matches(Object object) {
            return ( object instanceof Person );
        }
    };

// -- Create the semantic String object type.
// -- which maps directly to java String type.

ObjectType stringType = new ObjectType() {
        public boolean matches(Object object) {
            return ( object instanceof String );
        }
    };

// -- Declare two root fact Person objects
// -- with the identifiers 'sisOne' and 'sisTwo'

final Declaration sisOneDecl = new Declaration( personType,
                                                "sisOne" );

final Declaration sisTwoDecl = new Declaration( personType,
                                                "sisTwo" );

// -- Declare the extracted String object
// -- with the identifier 'petName'

final Declaration petNameDecl = new Declaration( stringType,
                                                 "petName" );

// -- Add the root fact Person declarations
// -- to the rule.

rule.addParameterDeclaration( sisOneDecl );
rule.addParameterDeclaration( sisTwoDecl );
```

```
// -- Create the fact extractor for the dog name

Extractor dogNameExtractor = new Extractor() {
        public Declaration[] getRequiredTupleMembers() {
            return new Declaration[] { sisOneDecl };
        }
        public Object extractFact(Tuple tuple) {
            Person person = (Person) tuple.get( sisOneDecl );
            return person.getDog().getName();
        }
    }
      );

// -- Create the fact extractor for the cat name

Extractor catNameExtractor = new Extractor() {
        public Declaration[] getRequiredTupleMembers() {
            return new Declaration[] { sisTwoDecl };
        }
        public Object extractFact(Tuple tuple) {
            Person person = (Person) tuple.get( sisTwoDecl );
            return person.getCat().getName();
        }
    }
      );

// -- Add the extractions for the dog and cat
// -- name, both to the 'petName' variable.

rule.addExtraction( new Extraction( petNameDecl,
                                    dogNameExtractor ) );

rule.addExtraction( new Extraction( petNameDecl,
                                    catNameExtractor ) );

// -- Add a filter that only allows two
// -- Persons who are sisters to pass.

rule.addCondition( new Condition() {
        public Declaration[] getRequiredTupleMembers() {
            return new Declaration[] { sisOneDecl, sisTwoDecl };
        }
        public boolean isAllowed(Tuple tuple) {
            Person sisOne = (Person) tuple.get( sisOneDecl )
            Person sisTwo = (Person) tuple.get( sisTwoDecl )
            return sisOne.hasSister( sisTwo ) ;
        }
    }
      );
```

```
// -- Attach an action to fire when matched.

rule.setConsequence( new Action() {
        public void invoke(Tuple tuple, WorkingMemory memory) {
            System.err.println( "sisOne: " + tuple.get( sisOneDecl ) );
            System.err.println( "sisTwo: " + tuple.get( sisTwoDecl ) );
            System.err.println( "petName: " + tuple.get( petNameDecl ) );
        }
    }
      );

// -- Create a new rule base

RuleBase ruleBase = new RuleBase();

try
{
    // -- Add the rule to the rule-base.
    //
    // -- May throw a ReteConstructionException
    // -- if the Rule cannot be integrated into
    // -- the Rete-OO network.

    ruleBase.addRule( rule );
}
catch (ReteConstructionException e)
{
    e.printStackTrace();
    return;
}

// -- Create a repository.

SimpleRepository repo = new SimpleRepository();

// -- Register the RuleBase with the repository.

repo.registerRuleBase( "http://rules.werken.com/family-relationships",
                       ruleBase );
```

Chapter 8

SPI

Semantics Provider Interface

8.1 Overview

At its core, drools is merely an algorithmic engine. The *Semantics Provider Interface*, also known as the SPI, provides for wrapping the drools Rete-OO algorithm with arbitrary application semantics. The semantics of several concepts are left open for definition by an implementation.

concept	interface	description
object type	`org.drools.spi.ObjectType`	Differentiates objects by type
condition	`org.drools.spi.Condition`	Tests tuples
fact extraction	`org.drools.spi.Extractor`	Extracts attributes from objects
action	`org.drools.spi.Consequence`	The result of a rule match

Additionally, implementations of some of these interfaces must work with the `Tuple` and `Declaration` objects that flow through the network.

The SPI provides for drools customizations to be tightly linked to the core Rete-OO algorithm engine. By using SPI implementations, no external translation or mapping process is required. Rules can be expressed directly using the semantics provided by the semantics module and manipulated directly by the engine.

8.2 Semantic Objects

8.2.1 org.drools.spi.ObjectType

While the type of any object presented to drools through the fact manipulation methods of a `WorkingMemory` is determined by the Java class of that object, drools provides the `ObjectType` interface for determining the object's semantic type within the algorithm.

```
public interface ObjectType
{
    /** Determine if the passed Object belongs to
     *  the object type defined by this ObjectType.
     *
     *  @param object The Object to test.
     *
     *  @return true if the Object matches this
     *          object type, Otherwise false.
     */
    boolean matches(Object object);
}
```

Figure 8.1: `org.drools.spi.ObjectType` interface

In the *XML Semantics Module*, all presented objects are of the class `Document`. The `ObjectType` implementation for the module inspects the root-level XML element to determine the object's type for the context of drools.

Semantic modules provide implementations of the `ObjectType` interface that are able to determine an object's type within a semantic realm.

The `matches(..)` method can perform any necessary logic to determine if semantic type of the object matches the type described by the `ObjectType` implementation. If the type does match, then `true` is returned. Otherwise, `false` indicates a non-match. A single asserted object may match several `ObjectTypes` as the core engine presents each object to every `ObjectType` for type determination. This allows for a single rule-base to contain a multitude of rules written against many semantic realms.

8.2.2 org.drools.spi.Declaration

A `Declaration` represents a named and typed object. Internally, the drools engine uses strongly-typed semantic objects. Named objects involved in a rule have a `Declaration` that binds an object identifier to an `ObjectType`. If an object is bound to a `Declaration` then the `ObjectType` of the `Declaration` must return `true` from its `matches(..)` method when evaluated against the bound object.

8.2.3 org.drools.spi.Tuple

While the external drools API is object-oriented, the core is still constructed of nodes through which *tuples* flow. A tuple is simply a dictionary of *key-to-value* mappings. A tuple may be considered to be similar to a *row* in a relational database table, where the *key* matches the column type and name, and the *value* matches the column data cell for that row.

The key used to index the associated value is always a `Declaration` object. When an object is initially asserted into a `WorkingMemory` it gets wrapped by

```
public class Declaration
{

    /** Retrieve the ObjectType of this Declaration.
     *
     *  @return The ObjectType of this Declaration.
     */
    public ObjectType getObjectType() { ... }

    /** Retrieve the variable's identifier.
     *
     *  @return The variable's identifier.
     */
    public String getIdentifier() { ... }
}
```

Figure 8.2: `org.drools.spi.Declaration` class

all matching `ParameterNodes` into single-column tuples with the object bound to the `Declaration` of each `ParameterNode`.

An example set of tuples was presented in table 10.2 on page 37. With the exception of the `ObjectType` interface, semantic modules operate on `Tuple` objects.

8.2.4 org.drools.spi.Extractor

Fact extraction is the process of performing an operation upon a `Tuple` to create additional columns or attributes on the `Tuple`. All `Tuples` initially have a single column matching a `Declaration` of the rule. Through fact extraction additional columns can be added based upon knowledge gained from the existing columns.

The *Java Semantic Module* uses normal Java expressions to extract other objects and values reachable from those already in the `Tuple`. The *XML Semantic Module* uses XPath [**?**] expressions to evaluate expressions against documents. The extracted values are associated with a `Declaration` and inserted into the `Tuple`.

Semantics modules provide implementations of the `Extractor` interface to perform fact extraction. A `Extractor` may require more than a single column to perform extraction. To specify which columns are required, the `getRequiredTupleMembers()` method should return an array of `Declarations` which must be present in any tuple presented for extraction.

8.2.5 org.drools.spi.Condition

A `Condition` is a predicate which evaluates against a `Tuple` to determine if the `Tuple` should pass or fail the condition. The `isAllowed(Tuple tuple)` method allows the condition to be evaluated against a `Tuple`. If the `Tuple` passes the

```
public interface Tuple

    /** Retrieve the value bound to a particular Declaration.
     *
     *  @param declaration The Declaration key.
     *
     *  @return The currently bound Object value.
     */
    Object get(Declaration declaration);

    /** Retrieve the Se of all Declarations active in this tuple.
     *
     *  @return The Set of all Declarations in this tuple.
     */
    Set getDeclarations();
```

Figure 8.3: `org.drools.spi.Tuple` interface

```
public interface Extractor
{
    /** Retrieve the array of Declarations required by this
     *  Extractor to perform extraction.
     *
     *  @return The array of Declarations expected
     *          on incoming Tuples.
     */
    Declaration[] getRequiredTupleMembers();

    /** Extract a new fact from the incoming Tuple.
     *
     *  @param tuple The source data tuple.
     *
     *  @return The newly extract fact object.
     *
     *  @throws ExtractionException if an error occurs during
     *          fact extraction activities.
     */
    Object extractFact(Tuple tuple) throws ExtractionException;
}
```

Figure 8.4: `org.drools.spi.Extractor` interface

```
public interface Condition extends Condition
{
    /** Retrieve the array of Declarations required
     *  by this condition to perform its duties.
     *
     *  @return The array of Declarations expected
     *          on incoming Tuples.
     */
    Declaration[] getRequiredTupleMembers();

    /** Determine if the supplied Tuple is allowed
     *  by this filter.
     *
     *  @param tuple The <code>Tuple</code> to test.
     *
     *  @return <code>true</code> if the <code>Tuple</code>
     *          passes this filter, else <code>false</code>.
     *
     *  @throws ConditionException if an error occurs during testing.
     */
    boolean isAllowed(Tuple tuple) throws ConditionException;
}
```

Figure 8.5: `org.drools.spi.Condition` interface

filter, then the method should return `true` otherwise `false` indicates that the `Tuple` does not pass.

Like the `Extractor`, a `Condition` may only be applicable to `Tuple` objects that contain some minimal set of columns. `Condition` implementations must also supply the `getRequiredTupleMembers()` method.

8.2.6 org.drools.spi.Consequence

When a collection of facts, represented by a `Tuple` satisifies all conditions of a rule then an `Consequence` is given an opportunity to fire and perform some activity. Since `Consequence`s may require the ability to manipulate more facts, instaces are provided not only the matching `Tuple`, but also the current `WorkingMemory` instance. `Consequence` also has a subclass named `Action` which allows directly modelling of trigger rules. `Action` is purely a marker interface.

```
public interface Consequence
{
    /** Execute the consequence for the supplied
     *  matching <code>Tuple</code>.
     *
     *  @param tuple The matching tuple.
     *  @param workingMemory The working memory session.
     *
     *  @throws ConsequenceException If an error occurs while
     *          attempting to invoke the consequence.
     */
    void invoke(Tuple tuple,
                WorkingMemory workingMemory) throws ActionInvokationException;
}
```

Figure 8.6: `org.drools.spi.Conseqeunce` interface

Chapter 9

SMF
Semantics Management Framework

. . . not implemented yet . . .

Chapter 10

Algorithms

10.1 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.

10.2 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.

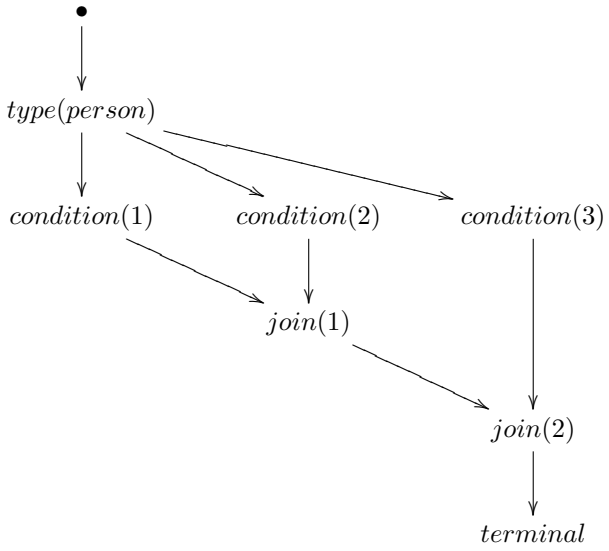

Figure 10.1: Rete network

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 10.2: Example tuple sets

If two sets of tuples (see Figure 10.2) 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.

10.3 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 langauge 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 10.2, 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 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.

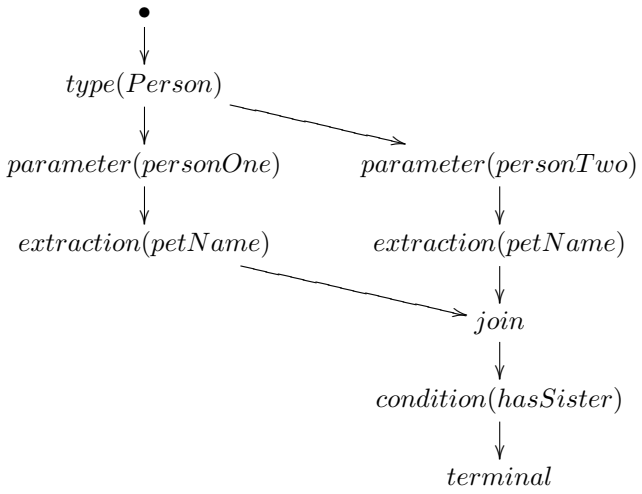

Figure 10.3: Rete-OO network

Appendix A

Project Information

A.1 Web Site

The main focal point of the drools project is the website, where information regarding source & binary distributions, source repository, mailing lists and chat archives can be found:

```
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.werken.com/listinfo/drools-interest
http://lists.werken.com/listinfo/drools-cvs
```

A.3 Source Repository

The drools project maintains a revision control repository using CVS. To check-out 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.werken.com:/cvsroot/drools login
cvs -d:pserver:anonymous@cvs.werken.com:/cvsroot/drools co drools
```

A.4 Internet Relay Chat

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

address	`irc.werken.com`
port	`6667`
channel	`#drools`
url	`irc://irc.werken.com:6667/drools`

Archives of chats in the channel are maintained on-line:

`http://irc.werken.com/channels/drools/`

A.5 Bug & Issue Tracking

The Werken Company provides access to their JIRA issue tracking server to support drools:

`http://jira.werken.com/`

Appendix B

Licensing

B.1 drools License

B.1.1 The License

STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

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`.

B.2 3rd-Party Licenses

drools contains software written by The Werken Company and by other third-party groups. Included are the licenses of included components.

B.2.1 Apache Jakarta

The Apache Software License, Version 1.1

B.2.2 Beanshell

B.2.3 ANTLR

SOFTWARE RIGHTS