Red Hat Decision Manager 7.5

Decision engine in Red Hat Decision Manager

Last Updated: 2020-05-22

Red Hat Decision Manager 7.5 Decision engine in Red Hat Decision Manager

Red Hat Customer Content Services
brms-docs@redhat.com

Legal Notice

Copyright © 2020 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at
http://creativecommons.org/licenses/by-sa/3.0/
. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, the Red Hat logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This document describes basic concepts and functions of the decision engine in Red Hat Decision Manager to consider as you create your business rule system and decision services in Red Hat Decision Manager.

Table of Contents

PREFACE

As a business rules developer, your understanding of the decision engine in Red Hat Decision Manager can help you design more effective business assets and a more scalable decision management architecture. The decision engine is the Red Hat Decision Manager component that stores, processes, and evaluates data to execute business rules and to reach the decisions that you define. This document describes basic concepts and functions of the decision engine to consider as you create your business rule system and decision services in Red Hat Decision Manager.

CHAPTER 1. DECISION ENGINE IN RED HAT DECISION MANAGER

The decision engine is the rules engine in Red Hat Decision Manager. The decision engine stores, processes, and evaluates data to execute the business rules or decision models that you define. The basic function of the decision engine is to match incoming data, or *facts*, to the conditions of rules and determine whether and how to execute the rules.

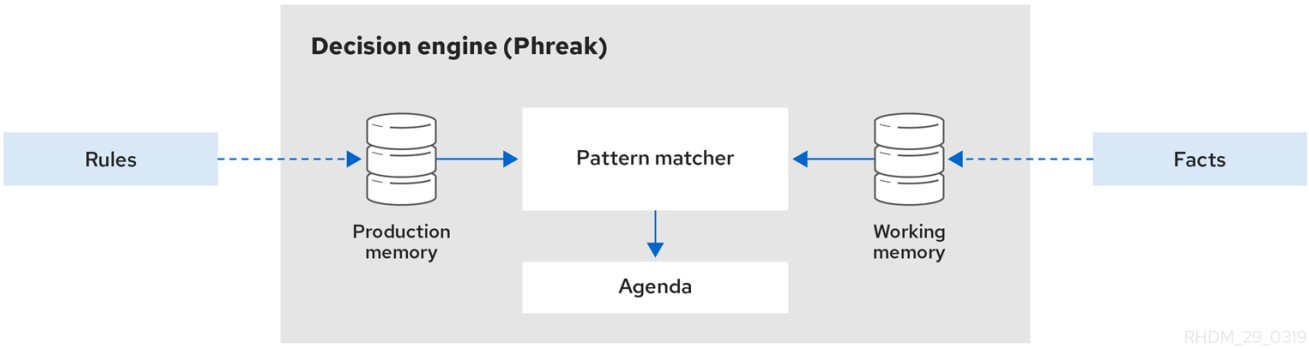
The decision engine operates using the following basic components:

- **Rules:** Business rules or DMN decisions that you define. All rules must contain at a minimum the conditions that trigger the rule and the actions that the rule dictates.
- **Facts:** Data that enters or changes in the decision engine that the decision engine matches to rule conditions to execute applicable rules.
- **Production memory:** Location where rules are stored in the decision engine.
- **Working memory:** Location where facts are stored in the decision engine.
- **Agenda:** Location where activated rules are registered and sorted (if applicable) in preparation for execution.

When a business user or an automated system adds or updates rule-related information in Red Hat Decision Manager, that information is inserted into the working memory of the decision engine in the form of one or more facts. The decision engine matches those facts to the conditions of the rules that are stored in the production memory to determine eligible rule executions. (This process of matching facts to rules is often referred to as *pattern matching*.) When rule conditions are met, the decision engine activates and registers rules in the agenda, where the decision engine then sorts prioritized or conflicting rules in preparation for execution.

The following diagram illustrates these basic components of the decision engine:

Figure 1.1. Overview of basic decision engine components



For more details and examples of rule and fact behavior in the decision engine, see Chapter 3, *Inference and truth maintenance in the decision engine*.

These core concepts can help you to better understand other more advanced components, processes, and sub-processes of the decision engine, and as a result, to design more effective business assets in Red Hat Decision Manager.

CHAPTER 2. KIE SESSIONS

In Red Hat Decision Manager, a KIE session stores and executes runtime data. The KIE session is created from a KIE base or directly from a KIE container if you have defined the KIE session in the KIE module descriptor file (**kmodule.xml**) for your project.

Example KIE session configuration in a kmodule.xml file

```
<kmodule>
  ...
  <kbase>
    ...
    <ksession name="KSession2_1" type="stateless" default="true" clockType="realtime">
    ...
  </kbase>
  ...
</kmodule>
```

A KIE base is a repository that you define in the KIE module descriptor file (**kmodule.xml**) for your project and contains all rules and other business assets in Red Hat Decision Manager, but does not contain any runtime data.

Example KIE base configuration in a kmodule.xml file

```
<kmodule>
  ...
  <kbase name="KBase2" default="false" eventProcessingMode="stream" equalsBehavior="equality"
declarativeAgenda="enabled" packages="org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

A KIE session can be stateless or stateful. In a stateless KIE session, data from a previous invocation of the KIE session (the previous session state) is discarded between session invocations. In a stateful KIE session, that data is retained. The type of KIE session you use depends on your project requirements and how you want data from different asset invocations to be persisted.

2.1. STATELESS KIE SESSIONS

A stateless KIE session is a session that does not use inference to make iterative changes to facts over time. In a stateless KIE session, data from a previous invocation of the KIE session (the previous session state) is discarded between session invocations, whereas in a stateful KIE session, that data is retained. A stateless KIE session behaves similarly to a function in that the results that it produces are determined by the contents of the KIE base and by the data that is passed into the KIE session for execution at a specific point in time. The KIE session has no memory of any data that was passed into the KIE session previously.

Stateless KIE sessions are commonly used for the following use cases:

- **Validation**, such as validating that a person is eligible for a mortgage
- **Calculation**, such as computing a mortgage premium

- **Routing and filtering**, such as sorting incoming emails into folders or sending incoming emails to a destination

For example, consider the following driver's license data model and sample DRL rule:

Data model for driver's license application

```
public class Applicant {
  private String name;
  private int age;
  private boolean valid;
  // Getter and setter methods
}
```

Sample DRL rule for driver's license application

```
package com.company.license

rule "Is of valid age"
when
  $a : Applicant(age < 18)
then
  $a.setValid(false);
end
```

The **Is of valid age** rule disqualifies any applicant younger than 18 years old. When the **Applicant** object is inserted into the decision engine, the decision engine evaluates the constraints for each rule and searches for a match. The **"objectType"** constraint is always implied, after which any number of explicit field constraints are evaluated. The variable **$a** is a binding variable that references the matched object in the rule consequence.

NOTE

The dollar sign (**$**) is optional and helps to differentiate between variable names and field names.

In this example, the sample rule and all other files in the ~/**resources** folder of the Red Hat Decision Manager project are built with the following code:

Create the KIE container

```
KieServices kieServices = KieServices.Factory.get();

KieContainer kContainer = kieServices.getKieClasspathContainer();
```

This code compiles all the rule files found on the class path and adds the result of this compilation, a **KieModule** object, in the **KieContainer**.

Finally, the **StatelessKieSession** object is instantiated from the **KieContainer** and is executed against specified data:

Instantiate the stateless KIE session and enter data

```
StatelessKieSession kSession = kContainer.newStatelessKieSession();

Applicant applicant = new Applicant("Mr John Smith", 16);

assertTrue(applicant.isValid());

ksession.execute(applicant);

assertFalse(applicant.isValid());
```

In a stateless KIE session configuration, the **execute()** call acts as a combination method that instantiates the **KieSession** object, adds all the user data and executes user commands, calls **fireAllRules()**, and then calls **dispose()**. Therefore, with a stateless KIE session, you do not need to call **fireAllRules()** or call **dispose()** after session invocation as you do with a stateful KIE session.

In this case, the specified applicant is under the age of 18, so the application is declined.

For a more complex use case, see the following example. This example uses a stateless KIE session and executes rules against an iterable list of objects, such as a collection.

Expanded data model for driver's license application

```
public class Applicant {
  private String name;
  private int age;
  // Getter and setter methods
}

public class Application {
  private Date dateApplied;
  private boolean valid;
  // Getter and setter methods
}
```

Expanded DRL rule set for driver's license application

```
package com.company.license

rule "Is of valid age"
when
  Applicant(age < 18)
  $a : Application()
then
  $a.setValid(false);
end

rule "Application was made this year"
when
  $a : Application(dateApplied > "01-jan-2009")
then
  $a.setValid(false);
end
```

Expanded Java source with iterable execution in a stateless KIE session

```
StatelessKieSession ksession = kbase.newStatelessKnowledgeSession();
Applicant applicant = new Applicant("Mr John Smith", 16);
Application application = new Application();

assertTrue(application.isValid());
ksession.execute(Arrays.asList(new Object[] { application, applicant })); (1)
assertFalse(application.isValid());

ksession.execute
  (CommandFactory.newInsertIterable(new Object[] { application, applicant })); (2)

List<Command> cmds = new ArrayList<Command>(); (3)
cmds.add(CommandFactory.newInsert(new Person("Mr John Smith"), "mrSmith"));
cmds.add(CommandFactory.newInsert(new Person("Mr John Doe"), "mrDoe"));

BatchExecutionResults results = ksession.execute(CommandFactory.newBatchExecution(cmds));
assertEquals(new Person("Mr John Smith"), results.getValue("mrSmith"));
```

1. Method for executing rules against an iterable collection of objects produced by the **Arrays.asList()** method. Every collection element is inserted before any matched rules are executed. The **execute(Object object)** and **execute(Iterable objects)** methods are wrappers around the **execute(Command command)** method that comes from the **BatchExecutor** interface.
2. Execution of the iterable collection of objects using the **CommandFactory** interface.
3. **BatchExecutor** and **CommandFactory** configurations for working with many different commands or result output identifiers. The **CommandFactory** interface supports other commands that you can use in the **BatchExecutor**, such as **StartProcess**, **Query**, and **SetGlobal**.

2.1.1. Global variables in stateless KIE sessions

The **StatelessKieSession** object supports global variables (globals) that you can configure to be resolved as session-scoped globals, delegate globals, or execution-scoped globals.

- **Session-scoped globals:** For session-scoped globals, you can use the method **getGlobals()** to return a **Globals** instance that provides access to the KIE session globals. These globals are used for all execution calls. Use caution with mutable globals because execution calls can be executing simultaneously in different threads.

  Session-scoped global

  ```
  import org.kie.api.runtime.StatelessKieSession;

  StatelessKieSession ksession = kbase.newStatelessKieSession();

  // Set a global `myGlobal` that can be used in the rules.
  ksession.setGlobal("myGlobal", "I am a global");

  // Execute while resolving the `myGlobal` identifier.
  ksession.execute(collection);
  ```

- **Delegate globals:** For delegate globals, you can assign a value to a global (with

setGlobal(String, Object)) so that the value is stored in an internal collection that maps identifiers to values. Identifiers in this internal collection have priority over any supplied delegate. If an identifier cannot be found in this internal collection, the delegate global (if any) is used.

- **Execution-scoped globals:** For execution-scoped globals, you can use the **Command** object to set a global that is passed to the **CommandExecutor** interface for execution-specific global resolution.

The **CommandExecutor** interface also enables you to export data using out identifiers for globals, inserted facts, and query results:

Out identifiers for globals, inserted facts, and query results

```
import org.kie.api.runtime.ExecutionResults;

// 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");
```

2.2. STATEFUL KIE SESSIONS

A stateful KIE session is a session that uses inference to make iterative changes to facts over time. In a stateful KIE session, data from a previous invocation of the KIE session (the previous session state) is retained between session invocations, whereas in a stateless KIE session, that data is discarded.

WARNING

Ensure that you call the **dispose()** method after running a stateful KIE session so that no memory leaks occur between session invocations.

Stateful KIE sessions are commonly used for the following use cases:

- **Monitoring**, such as monitoring a stock market and automating the buying process
- **Diagnostics**, such as running fault-finding processes or medical diagnostic processes
- **Logistics**, such as parcel tracking and delivery provisioning

- **Ensuring compliance**, such as verifying the legality of market trades

For example, consider the following fire alarm data model and sample DRL rules:

Data model for sprinklers and fire alarm

```
public class Room {
  private String name;
  // Getter and setter methods
}

public class Sprinkler {
  private Room room;
  private boolean on;
  // Getter and setter methods
}

public class Fire {
  private Room room;
  // Getter and setter methods
}

public class Alarm { }
```

Sample DRL rule set for activating sprinklers and alarm

```
rule "When there is a fire turn on the sprinkler"
when
  Fire($room : room)
  $sprinkler : Sprinkler(room == $room, on == false)
then
  modify($sprinkler) { setOn(true) };
  System.out.println("Turn on the sprinkler for room "+$room.getName());
end

rule "Raise the alarm when we have one or more fires"
when
    exists Fire()
then
    insert( new Alarm() );
    System.out.println( "Raise the alarm" );
end

rule "Cancel the alarm when all the fires have gone"
when
    not Fire()
    $alarm : Alarm()
then
    delete( $alarm );
    System.out.println( "Cancel the alarm" );
end


rule "Status output when things are ok"
when
```

```
    not Alarm()
    not Sprinkler( on == true )
then
    System.out.println( "Everything is ok" );
end
```

For the **When there is a fire turn on the sprinkler** rule, when a fire occurs, the instances of the **Fire** class are created for that room and inserted into the KIE session. The rule adds a constraint for the specific **room** matched in the **Fire** instance so that only the sprinkler for that room is checked. When this rule is executed, the sprinkler activates. The other sample rules determine when the alarm is activated or deactivated accordingly.

Whereas a stateless KIE session relies on standard Java syntax to modify a field, a stateful KIE session relies on the **modify** statement in rules to notify the decision engine of changes. The decision engine then reasons over the changes and assesses impact on subsequent rule executions. This process is part of the decision engine ability to use *inference* and *truth maintenance* and is essential in stateful KIE sessions.

In this example, the sample rules and all other files in the **~/resources** folder of the Red Hat Decision Manager project are built with the following code:

Create the KIE container

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

This code compiles all the rule files found on the class path and adds the result of this compilation, a **KieModule** object, in the **KieContainer**.

Finally, the **KieSession** object is instantiated from the **KieContainer** and is executed against specified data:

Instantiate the stateful KIE session and enter data

```
KieSession ksession = kContainer.newKieSession();

String[] names = new String[]{"kitchen", "bedroom", "office", "livingroom"};
Map<String,Room> name2room = new HashMap<String,Room>();
for( String name: names ){
    Room room = new Room( name );
    name2room.put( name, room );
    ksession.insert( room );
    Sprinkler sprinkler = new Sprinkler( room );
    ksession.insert( sprinkler );
}

ksession.fireAllRules();
```

Console output

```
> Everything is ok
```

With the data added, the decision engine completes all pattern matching but no rules have been executed, so the configured verification message appears. As new data triggers rule conditions, the

decision engine executes rules to activate the alarm and later to cancel the alarm that has been activated:

Enter new data to trigger rules

```
Fire kitchenFire = new Fire( name2room.get( "kitchen" ) );
Fire officeFire = new Fire( name2room.get( "office" ) );

FactHandle kitchenFireHandle = ksession.insert( kitchenFire );
FactHandle officeFireHandle = ksession.insert( officeFire );

ksession.fireAllRules();
```

Console output

```
> Raise the alarm
> Turn on the sprinkler for room kitchen
> Turn on the sprinkler for room office
```

```
ksession.delete( kitchenFireHandle );
ksession.delete( officeFireHandle );

ksession.fireAllRules();
```

Console output

```
> Cancel the alarm
> Turn off the sprinkler for room office
> Turn off the sprinkler for room kitchen
> Everything is ok
```

In this case, a reference is kept for the returned **FactHandle** object. A fact handle is an internal engine reference to the inserted instance and enables instances to be retracted or modified later.

As this example illustrates, the data and results from previous stateful KIE sessions (the activated alarm) affect the invocation of subsequent sessions (alarm cancellation).

2.3. KIE SESSION POOLS

In use cases with large amounts of KIE runtime data and high system activity, KIE sessions might be created and disposed very frequently. A high turnover of KIE sessions is not always time consuming, but when the turnover is repeated millions of times, the process can become a bottleneck and require substantial clean-up effort.

For these high-volume cases, you can use KIE session pools instead of many individual KIE sessions. To use a KIE session pool, you obtain a KIE session pool from a KIE container, define the initial number of KIE sessions in the pool, and create the KIE sessions from that pool as usual:

Example KIE session pool

```
// Obtain a KIE session pool from the KIE container
KieContainerSessionsPool pool = kContainer.newKieSessionsPool(10);
```

```
// Create KIE sessions from the KIE session pool
KieSession kSession = pool.newKieSession();
```

In this example, the KIE session pool starts with 10 KIE sessions in it, but you can specify the number of KIE sessions that you need. This integer value is the number of KIE sessions that are only initially created in the pool. If required by the running application, the number of KIE sessions in the pool can dynamically grow beyond that value.

After you define a KIE session pool, the next time you use the KIE session as usual and call **dispose()** on it, the KIE session is reset and pushed back into the pool instead of being destroyed.

KIE session pools typically apply to stateful KIE sessions, but KIE session pools can also affect stateless KIE sessions that you reuse with multiple **execute()** calls. When you create a stateless KIE session directly from a KIE container, the KIE session continues to internally create a new KIE session for each **execute()** invocation. Conversely, when you create a stateless KIE session from a KIE session pool, the KIE session internally uses only the specific KIE sessions provided by the pool.

When you finish using a KIE session pool, you can call the **shutdown()** method on it to avoid memory leaks. Alternatively, you can call **dispose()** on the KIE container to shut down all the pools created from the KIE container.

CHAPTER 3. INFERENCE AND TRUTH MAINTENANCE IN THE DECISION ENGINE

The basic function of the decision engine is to match data to business rules and determine whether and how to execute rules. To ensure that relevant data is applied to the appropriate rules, the decision engine makes *inferences* based on existing knowledge and performs the actions based on the inferred information.

For example, the following DRL rule determines the age requirements for adults, such as in a bus pass policy:

Rule to define age requirement

```
rule "Infer Adult"
when
  $p : Person(age >= 18)
then
  insert(new IsAdult($p))
end
```

Based on this rule, the decision engine infers whether a person is an adult or a child and performs the specified action (the **then** consequence). Every person who is 18 years old or older has an instance of **IsAdult** inserted for them in the working memory. This inferred relation of age and bus pass can then be invoked in any rule, such as in the following rule segment:

```
$p : Person()
IsAdult(person == $p)
```

In many cases, new data in a rule system is the result of other rule executions, and this new data can affect the execution of other rules. If the decision engine asserts data as a result of executing a rule, the decision engine uses truth maintenance to justify the assertion and enforce truthfulness when applying inferred information to other rules. Truth maintenance also helps to identify inconsistencies and to handle contradictions. For example, if two rules are executed and result in a contradictory action, the decision engine chooses the action based on assumptions from previously calculated conclusions.

The decision engine inserts facts using either stated or logical insertions:

- **Stated insertions:** Defined with **insert()**. After stated insertions, facts are generally retracted explicitly. (The term *insertion*, when used generically, refers to *stated insertion*.)
- **Logical insertions:** Defined with **insertLogical()**. After logical insertions, the facts that were inserted are automatically retracted when the conditions in the rules that inserted the facts are no longer true. The facts are retracted when no condition supports the logical insertion. A fact that is logically inserted is considered to be *justified* by the decision engine.

For example, the following sample DRL rules use stated fact insertion to determine the age requirements for issuing a child bus pass or an adult bus pass:

Rules to issue bus pass, stated insertion

```
rule "Issue Child Bus Pass"
when
  $p : Person(age < 18)
then
```

```
  insert(new ChildBusPass($p));
end

rule "Issue Adult Bus Pass"
when
  $p : Person(age >= 18)
then
  insert(new AdultBusPass($p));
end
```

These rules are not easily maintained in the decision engine as bus riders increase in age and move from child to adult bus pass. As an alternative, these rules can be separated into rules for bus rider age and rules for bus pass type using logical fact insertion. The logical insertion of the fact makes the fact dependent on the truth of the **when** clause.

The following DRL rules use logical insertion to determine the age requirements for children and adults:

Children and adult age requirements, logical insertion

```
rule "Infer Child"
when
  $p : Person(age < 18)
then
  insertLogical(new IsChild($p))
end

rule "Infer Adult"
when
  $p : Person(age >= 18)
then
  insertLogical(new IsAdult($p))
end
```

IMPORTANT

For logical insertions, your fact objects must override the **equals** and **hashCode** methods from the **java.lang.Object** object according to the Java standard. Two objects are equal if their **equals** methods return **true** for each other and if their **hashCode** methods return the same values. For more information, see the Java API documentation for your Java version.

When the condition in the rule is false, the fact is automatically retracted. This behavior is helpful in this example because the two rules are mutually exclusive. In this example, if the person is younger than 18 years old, the rule logically inserts an **IsChild** fact. After the person is 18 years old or older, the **IsChild** fact is automatically retracted and the **IsAdult** fact is inserted.

The following DRL rules then determine whether to issue a child bus pass or an adult bus pass and logically insert the **ChildBusPass** and **AdultBusPass** facts. This rule configuration is possible because the truth maintenance system in the decision engine supports chaining of logical insertions for a cascading set of retracts.

Rules to issue bus pass, logical insertion

```
rule "Issue Child Bus Pass"
```

```
when
  $p : Person()
    IsChild(person == $p)
then
  insertLogical(new ChildBusPass($p));
end

rule "Issue Adult Bus Pass"
when
  $p : Person()
    IsAdult(person =$p)
then
  insertLogical(new AdultBusPass($p));
end
```

When a person turns 18 years old, the **IsChild** fact and the person's **ChildBusPass** fact is retracted. To these set of conditions, you can relate another rule that states that a person must return the child pass after turning 18 years old. When the decision engine automatically retracts the **ChildBusPass** object, the following rule is executed to send a request to the person:

Rule to notify bus pass holder of new pass

```
rule "Return ChildBusPass Request"
when
  $p : Person()
    not(ChildBusPass(person == $p))
then
  requestChildBusPass($p);
end
```

The following flowcharts illustrate the life cycle of stated and logical insertions:

Figure 3.1. Stated insertion

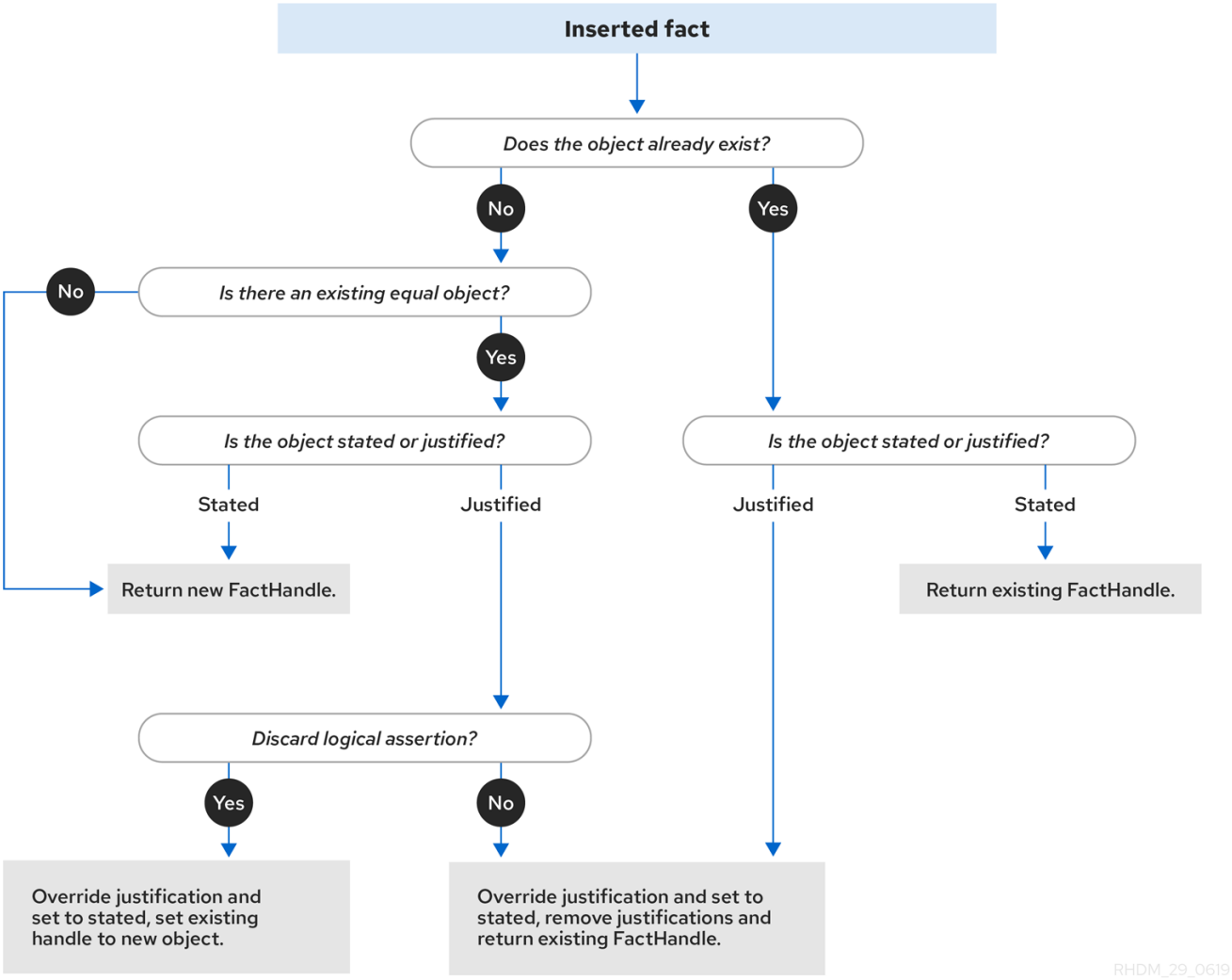
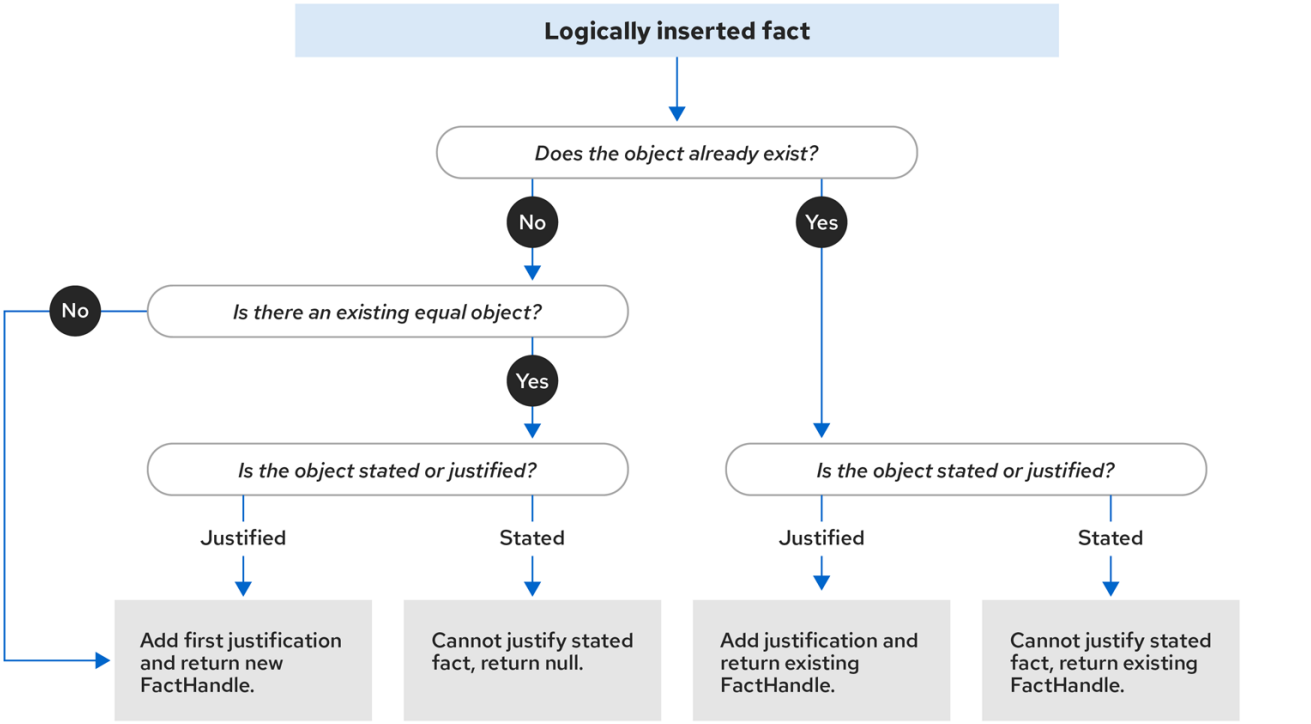


Figure 3.2. Logical insertion



When the decision engine logically inserts an object during a rule execution, the decision engine *justifies* the object by executing the rule. For each logical insertion, only one equal object can exist, and each subsequent equal logical insertion increases the justification counter for that logical insertion. A justification is removed when the conditions of the rule become untrue. When no more justifications exist, the logical object is automatically retracted.

3.1. FACT EQUALITY MODES IN THE DECISION ENGINE

The decision engine supports the following fact equality modes that determine how the decision engine stores and compares inserted facts:

- **identity**: (Default) The decision engine uses an **IdentityHashMap** to store all inserted facts. For every new fact insertion, the decision engine returns a new **FactHandle** object. If a fact is inserted again, the decision engine returns the original **FactHandle** object, ignoring repeated insertions for the same fact. In this mode, two facts are the same for the decision engine only if they are the very same object with the same identity.

- **equality**: The decision engine uses a **HashMap** to store all inserted facts. The decision engine returns a new **FactHandle** object only if the inserted fact is not equal to an existing fact, according to the **equals()** method of the inserted fact. In this mode, two facts are the same for the decision engine if they are composed the same way, regardless of identity. Use this mode when you want objects to be assessed based on feature equality instead of explicit identity.

As an illustration of fact equality modes, consider the following example facts:

Example facts

```
Person p1 = new Person("John", 45);
Person p2 = new Person("John", 45);
```

In **identity** mode, facts **p1** and **p2** are different instances of a **Person** class and are treated as separate objects because they have separate identities. In **equality** mode, facts **p1** and **p2** are treated as the same object because they are composed the same way. This difference in behavior affects how you can interact with fact handles.

For example, assume that you insert facts **p1** and **p2** into the decision engine and later you want to retrieve the fact handle for **p1**. In **identity** mode, you must specify **p1** to return the fact handle for that exact object, whereas in **equality** mode, you can specify **p1**, **p2**, or **new Person("John", 45)** to return the fact handle.

Example code to insert a fact and return the fact handle in identity mode

```
ksession.insert(p1);

ksession.getFactHandle(p1);
```

Example code to insert a fact and return the fact handle in equality mode

```
ksession.insert(p1);

ksession.getFactHandle(p1);

// Alternate option:
ksession.getFactHandle(new Person("John", 45));
```

To set the fact equality mode, use one of the following options:

- Set the system property **drools.equalityBehavior** to **identity** (default) or **equality**.
- Set the equality mode while creating the KIE base programatically:

```
KieServices ks = KieServices.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
kieBaseConf.setOption(EqualityBehaviorOption.EQUALITY);
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

- Set the equality mode in the KIE module descriptor file (**kmodule.xml**) for a specific Red Hat Decision Manager project:

```
<kmodule>
  ...
  <kbase name="KBase2" default="false" equalsBehavior="equality"
packages="org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

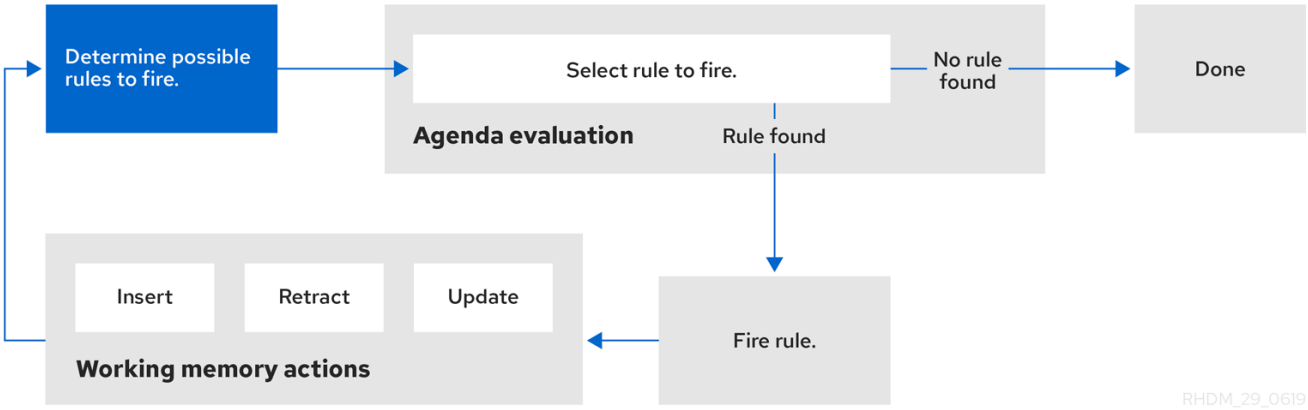
CHAPTER 4. EXECUTION CONTROL IN THE DECISION ENGINE

When new rule data enters the working memory of the decision engine, rules may become fully matched and eligible for execution. A single working memory action can result in multiple eligible rule executions. When a rule is fully matched, the decision engine creates an activation instance, referencing the rule and the matched facts, and adds the activation onto the decision engine agenda. The agenda controls the execution order of these rule activations using a conflict resolution strategy.

After the first call of **fireAllRules()** in the Java application, the decision engine cycles repeatedly through two phases:

- **Agenda evaluation.** In this phase, the decision engine selects all rules that can be executed. If no executable rules exist, the execution cycle ends. If an executable rule is found, the decision engine registers the activation in the agenda and then moves on to the working memory actions phase to perform rule consequence actions.
- **Working memory actions.** In this phase, the decision engine performs the rule consequence actions (the **then** portion of each rule) for all activated rules previously registered in the agenda. After all the consequence actions are complete or the main Java application process calls **fireAllRules()** again, the decision engine returns to the agenda evaluation phase to reassess rules.

Figure 4.1. Two-phase execution process in the decision engine



When multiple rules exist on the agenda, the execution of one rule may cause another rule to be removed from the agenda. To avoid this, you can define how and when rules are executed in the decision engine. Some common methods for defining rule execution order are by using rule salience, agenda groups, activation groups, or rule units for DRL rule sets.

4.1. SALIENCE FOR RULES

Each rule has an integer **salience** attribute that determines the order of execution. Rules with a higher salience value are given higher priority when ordered in the activation queue. The default salience value for rules is zero, but the salience can be negative or positive.

For example, the following sample DRL rules are listed in the decision engine stack in the order shown:

```
rule "RuleA"
salience 95
when
```

```
    $fact : MyFact( field1 == true )
then
    System.out.println("Rule2 : " + $fact);
    update($fact);
end

rule "RuleB"
salience 100
when
   $fact : MyFact( field1 == false )
then
   System.out.println("Rule1 : " + $fact);
   $fact.setField1(true);
   update($fact);
end
```

The **RuleB** rule is listed second, but it has a higher salience value than the **RuleA** rule and is therefore executed first.

4.2. AGENDA GROUPS FOR RULES

An agenda group is a set of rules bound together by the same **agenda-group** rule attribute. Agenda groups partition rules on the decision engine agenda. At any one time, only one group has a *focus* that gives that group of rules priority for execution before rules in other agenda groups. You determine the focus with a **setFocus()** call for the agenda group. You can also define rules with an **auto-focus** attribute so that the next time the rule is activated, the focus is automatically given to the entire agenda group to which the rule is assigned.

Each time the **setFocus()** call is made in a Java application, the decision engine adds the specified agenda group to the top of the rule stack. The default agenda group **"MAIN"** contains all rules that do not belong to a specified agenda group and is executed first in the stack unless another group has the focus.

For example, the following sample DRL rules belong to specified agenda groups and are listed in the decision engine stack in the order shown:

Sample DRL rules for banking application

```
rule "Increase balance for credits"
  agenda-group "calculation"
when
  ap : AccountPeriod()
  acc : Account( $accountNo : accountNo )
  CashFlow( type == CREDIT,
            accountNo == $accountNo,
            date >= ap.start && <= ap.end,
            $amount : amount )
then
  acc.balance  += $amount;
end
```

```
rule "Print balance for AccountPeriod"
  agenda-group "report"
when
  ap : AccountPeriod()
```

```
  acc : Account()
then
  System.out.println( acc.accountNo +
                      " : " + acc.balance );
end
```

For this example, the rules in the **"report"** agenda group must always be executed first and the rules in the **"calculation"** agenda group must always be executed second. Any remaining rules in other agenda groups can then be executed. Therefore, the **"report"** and **"calculation"** groups must receive the focus to be executed in that order, before other rules can be executed:

Set the focus for the order of agenda group execution

```
Agenda agenda = ksession.getAgenda();
agenda.getAgendaGroup( "report" ).setFocus();
agenda.getAgendaGroup( "calculation" ).setFocus();
ksession.fireAllRules();
```

You can also use the **clear()** method to cancel all the activations generated by the rules belonging to a given agenda group before each has had a chance to be executed:

Cancel all other rule activations

```
ksession.getAgenda().getAgendaGroup( "Group A" ).clear();
```

4.3. ACTIVATION GROUPS FOR RULES

An activation group is a set of rules bound together by the same **activation-group** rule attribute. In this group, only one rule can be executed. After conditions are met for a rule in that group to be executed, all other pending rule executions from that activation group are removed from the agenda.

For example, the following sample DRL rules belong to the specified activation group and are listed in the decision engine stack in the order shown:

Sample DRL rules for banking

```
rule "Print balance for AccountPeriod1"
  activation-group "report"
when
  ap : AccountPeriod1()
  acc : Account()
then
  System.out.println( acc.accountNo +
                      " : " + acc.balance );
end
```

```
rule "Print balance for AccountPeriod2"
  activation-group "report"
when
  ap : AccountPeriod2()
  acc : Account()
then
```

```
  System.out.println( acc.accountNo +
                      " : " + acc.balance );
end
```

For this example, if the first rule in the **"report"** activation group is executed, the second rule in the group and all other executable rules on the agenda are removed from the agenda.

4.4. RULE EXECUTION MODES AND THREAD SAFETY IN THE DECISION ENGINE

The decision engine supports the following rule execution modes that determine how and when the decision engine executes rules:

- **Passive mode**: (Default) The decision engine evaluates rules when a user or an application explicitly calls **fireAllRules()**. Passive mode in the decision engine is best for applications that require direct control over rule evaluation and execution, or for complex event processing (CEP) applications that use the pseudo clock implementation in the decision engine.

  Example CEP application code with the decision engine in passive mode

  ```
  KieSessionConfiguration config = KieServices.Factory.get().newKieSessionConfiguration();
  config.setOption( ClockTypeOption.get("pseudo") );
  KieSession session = kbase.newKieSession( conf, null );
  SessionPseudoClock clock = session.getSessionClock();

  session.insert( tick1 );
  session.fireAllRules();

  clock.advanceTime(1, TimeUnit.SECONDS);
  session.insert( tick2 );
  session.fireAllRules();

  clock.advanceTime(1, TimeUnit.SECONDS);
  session.insert( tick3 );
  session.fireAllRules();

  session.dispose();
  ```

- **Active mode**: If a user or application calls **fireUntilHalt()**, the decision engine starts in active mode and evaluates rules continually until the user or application explicitly calls **halt()**. Active mode in the decision engine is best for applications that delegate control of rule evaluation and execution to the decision engine, or for complex event processing (CEP) applications that use the real-time clock implementation in the decision engine. Active mode is also optimal for CEP applications that use active queries.

  Example CEP application code with the decision engine in active mode

  ```
  KieSessionConfiguration config = KieServices.Factory.get().newKieSessionConfiguration();
  config.setOption( ClockTypeOption.get("realtime") );
  KieSession session = kbase.newKieSession( conf, null );

  new Thread( new Runnable() {
    @Override
    public void run() {
  ```

```
      session.fireUntilHalt();
  }
} ).start();

session.insert( tick1 );

... Thread.sleep( 1000L ); ...

session.insert( tick2 );

... Thread.sleep( 1000L ); ...

session.insert( tick3 );

session.halt();
session.dispose();
```

This example calls **fireUntilHalt()** from a dedicated execution thread to prevent the current thread from being blocked indefinitely while the decision engine continues evaluating rules. The dedicated thread also enables you to call **halt()** at a later stage in the application code.

Although you should avoid using both **fireAllRules()** and **fireUntilHalt()** calls, especially from different threads, the decision engine can handle such situations safely using thread-safety logic and an internal state machine. If a **fireAllRules()** call is in progress and you call **fireUntilHalt()**, the decision engine continues to run in passive mode until the **fireAllRules()** operation is complete and then starts in active mode in response to the **fireUntilHalt()** call. However, if the decision engine is running in active mode following a **fireUntilHalt()** call and you call **fireAllRules()**, the **fireAllRules()** call is ignored and the decision engine continues to run in active mode until you call **halt()**.

For added thread safety in active mode, the decision engine supports a **submit()** method that you can use to group and perform operations on a KIE session in a thread-safe, atomic action:

Example application code with submit() method to perform atomic operations in active mode

```
KieSession session = ...;

new Thread( new Runnable() {
  @Override
  public void run() {
      session.fireUntilHalt();
  }
} ).start();

final FactHandle fh = session.insert( fact_a );

... Thread.sleep( 1000L ); ...

session.submit( new KieSession.AtomicAction() {
  @Override
  public void execute( KieSession kieSession ) {
    fact_a.setField("value");
    kieSession.update( fh, fact_a );
    kieSession.insert( fact_1 );
    kieSession.insert( fact_2 );
    kieSession.insert( fact_3 );
```

```
  }
} );

... Thread.sleep( 1000L ); ...

session.insert( fact_z );

session.halt();
session.dispose();
```

Thread safety and atomic operations are also helpful from a client-side perspective. For example, you might need to insert more than one fact at a given time, but require the decision engine to consider the insertions as an atomic operation and to wait until all the insertions are complete before evaluating the rules again.

4.5. FACT PROPAGATION MODES IN THE DECISION ENGINE

The decision engine supports the following fact propagation modes that determine how the decision engine progresses inserted facts through the engine network in preparation for rule execution:

- **Lazy**: (Default) Facts are propagated in batch collections at rule execution, not in real time as the facts are individually inserted by a user or application. As a result, the order in which the facts are ultimately propagated through the decision engine may be different from the order in which the facts were individually inserted.
- **Immediate**: Facts are propagated immediately in the order that they are inserted by a user or application.
- **Eager**: Facts are propagated lazily (in batch collections), but before rule execution. The decision engine uses this propagation behavior for rules that have the **no-loop** or **lock-on-active** attribute.

By default, the Phreak rule algorithm in the decision engine uses lazy fact propagation for improved rule evaluation overall. However, in few cases, this lazy propagation behavior can alter the expected result of certain rule executions that may require immediate or eager propagation.

For example, the following rule uses a specified query with a **?** prefix to invoke the query in pull-only or passive fashion:

Example rule with a passive query

```
query Q (Integer i)
   String( this == i.toString() )
end

rule "Rule"
  when
    $i : Integer()
    ?Q( $i; )
  then
    System.out.println( $i );
end
```

For this example, the rule should be executed only when a **String** that satisfies the query is inserted before the **Integer**, such as in the following example commands:

Example commands that should trigger the rule execution

```
KieSession ksession = ...
ksession.insert("1");
ksession.insert(1);
ksession.fireAllRules();
```

However, due to the default lazy propagation behavior in Phreak, the decision engine does not detect the insertion sequence of the two facts in this case, so this rule is executed regardless of **String** and **Integer** insertion order. For this example, immediate propagation is required for the expected rule evaluation.

To alter the decision engine propagation mode to achieve the expected rule evaluation in this case, you can add the **@Propagation(<type>)** tag to your rule and set **<type>** to **LAZY**, **IMMEDIATE**, or **EAGER**.

In the same example rule, the immediate propagation annotation enables the rule to be evaluated only when a **String** that satisfies the query is inserted before the **Integer**, as expected:

Example rule with a passive query and specified propagation mode

```
query Q (Integer i)
    String( this == i.toString() )
end

rule "Rule" @Propagation(IMMEDIATE)
  when
    $i : Integer()
    ?Q( $i; )
  then
    System.out.println( $i );
end
```

4.6. AGENDA EVALUATION FILTERS

The decision engine supports an **AgendaFilter** object in the filter interface that you can use to allow or deny the evaluation of specified rules during agenda evaluation. You can specify an agenda filter as part of a **fireAllRules()** call.

The following example code permits only rules ending with the string **"Test"** to be evaluated and executed. All other rules are filtered out of the decision engine agenda.

Example agenda filter definition

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

4.7. RULE UNITS IN DRL RULE SETS

Rule units are groups of data sources, global variables, and DRL rules that function together for a specific purpose. You can use rule units to partition a rule set into smaller units, bind different data sources to those units, and then execute the individual unit. Rule units are an enhanced alternative to rule-grouping DRL attributes such as rule agenda groups or activation groups for execution control.

Rule units are helpful when you want to coordinate rule execution so that the complete execution of one

rule unit triggers the start of another rule unit and so on. For example, assume that you have a set of rules for data enrichment, another set of rules that processes that data, and another set of rules that extract the output from the processed data. If you add these rule sets into three distinct rule units, you can coordinate those rule units so that complete execution of the first unit triggers the start of the second unit and the complete execution of the second unit triggers the start of third unit.

To define a rule unit, implement the **RuleUnit** interface as shown in the following example:

Example rule unit class

```
package org.mypackage.myunit;

public static class AdultUnit implements RuleUnit {
    private int adultAge;
    private DataSource<Person> persons;

    public AdultUnit( ) { }

    public AdultUnit( DataSource<Person> persons, int age ) {
        this.persons = persons;
        this.age = age;
    }

    // A data source of `Persons` in this rule unit:
    public DataSource<Person> getPersons() {
        return persons;
    }

    // A global variable in this rule unit:
    public int getAdultAge() {
        return adultAge;
    }

    // Life-cycle methods:
    @Override
    public void onStart() {
        System.out.println("AdultUnit started.");
    }

    @Override
    public void onEnd() {
        System.out.println("AdultUnit ended.");
    }
}
```

In this example, **persons** is a source of facts of type **Person**. A rule unit data source is a source of the data processed by a given rule unit and represents the entry point that the decision engine uses to evaluate the rule unit. The **adultAge** global variable is accessible from all the rules belonging to this rule unit. The last two methods are part of the rule unit life cycle and are invoked by the decision engine.

The decision engine supports the following optional life-cycle methods for rule units:

Table 4.1. Rule unit life-cycle methods

Method	Invoked when
onStart()	Rule unit execution starts
onEnd()	Rule unit execution ends
onSuspend()	Rule unit execution is suspended (used only with **runUntilHalt()**)
onResume()	Rule unit execution is resumed (used only with **runUntilHalt()**)
onYield(RuleUnit other)	The consequence of a rule in the rule unit triggers the execution of a different rule unit

You can add one or more rules to a rule unit. By default, all the rules in a DRL file are automatically associated with a rule unit that follows the naming convention of the DRL file name. If the DRL file is in the same package and has the same name as a class that implements the **RuleUnit** interface, then all of the rules in that DRL file implicitly belong to that rule unit. For example, all the rules in the **AdultUnit.drl** file in the **org.mypackage.myunit** package are automatically part of the rule unit **org.mypackage.myunit.AdultUnit**.

To override this naming convention and explicitly declare the rule unit that the rules in a DRL file belong to, use the **unit** keyword in the DRL file. The **unit** declaration must immediately follow the package declaration and contain the name of the class in that package that the rules in the DRL file are part of.

Example rule unit declaration in a DRL file

```
package org.mypackage.myunit
unit AdultUnit

rule Adult
  when
    $p : Person(age >= adultAge) from persons
  then
    System.out.println($p.getName() + " is adult and greater than " + adultAge);
end
```

WARNING

Do not mix rules with and without a rule unit in the same KIE base. Mixing two rule paradigms in a KIE base results in a compilation error.

You can also rewrite the same pattern in a more convenient way using OOPath notation, as shown in the following example:

Example rule unit declaration in a DRL file that uses OOPath notation

```
package org.mypackage.myunit
unit AdultUnit

rule Adult
  when
    $p : /persons[age >= adultAge]
  then
    System.out.println($p.getName() + " is adult and greater than " + adultAge);
end
```

NOTE

OOPath is an object-oriented syntax extension of XPath that is designed for browsing graphs of objects in DRL rule condition constraints. OOPath uses the compact notation from XPath for navigating through related elements while handling collections and filtering constraints, and is specifically useful for graphs of objects.

In this example, any matching facts in the rule conditions are retrieved from the **persons** data source defined in the **DataSource** definition in the rule unit class. The rule condition and action use the **adultAge** variable in the same way that a global variable is defined at the DRL file level.

To execute one or more rule units defined in a KIE base, create a new **RuleUnitExecutor** class bound to the KIE base, create the rule unit from the relevant data source, and run the rule unit executer:

Example rule unit execution

```
// Create a `RuleUnitExecutor` class and bind it to the KIE base:
KieBase kbase = kieContainer.getKieBase();
RuleUnitExecutor executor = RuleUnitExecutor.create().bind( kbase );

// Create the `AdultUnit` rule unit using the `persons` data source and run the executor:
RuleUnit adultUnit = new AdultUnit(persons, 18);
executor.run( adultUnit );
```

Rules are executed by the **RuleUnitExecutor** class. The **RuleUnitExecutor** class creates KIE sessions and adds the required **DataSource** objects to those sessions, and then executes the rules based on the **RuleUnit** that is passed as a parameter to the **run()** method.

The example execution code produces the following output when the relevant **Person** facts are inserted in the **persons** data source:

Example rule unit execution output

```
org.mypackage.myunit.AdultUnit started.
Jane is adult and greater than 18
John is adult and greater than 18
org.mypackage.myunit.AdultUnit ended.
```

Instead of explicitly creating the rule unit instance, you can register the rule unit variables in the executor and pass to the executor the rule unit class that you want to run, and then the executor creates an instance of the rule unit. You can then set the **DataSource** definition and other variables as needed before running the rule unit.

Alternate rule unit execution option with registered variables

```
executor.bindVariable( "persons", persons );
        .bindVariable( "adultAge", 18 );
executor.run( AdultUnit.class );
```

The name that you pass to the **RuleUnitExecutor.bindVariable()** method is used at run time to bind the variable to the field of the rule unit class with the same name. In the previous example, the **RuleUnitExecutor** inserts into the new rule unit the data source bound to the **"persons"** name and inserts the value **18** bound to the String **"adultAge"** into the fields with the corresponding names inside the **AdultUnit** class.

To override this default variable-binding behavior, use the **@UnitVar** annotation to explicitly define a logical binding name for each field of the rule unit class. For example, the field bindings in the following class are redefined with alternative names:

Example code to modify variable binding names with @UnitVar

```
package org.mypackage.myunit;

public static class AdultUnit implements RuleUnit {
    @UnitVar("minAge")
    private int adultAge = 18;

    @UnitVar("data")
    private DataSource<Person> persons;
}
```

You can then bind the variables to the executor using those alternative names and run the rule unit:

Example rule unit execution with modified variable names

```
executor.bindVariable( "data", persons );
        .bindVariable( "minAge", 18 );
executor.run( AdultUnit.class );
```

You can execute a rule unit in *passive mode* by using the **run()** method (equivalent to invoking **fireAllRules()** on a KIE session) or in *active mode* using the **runUntilHalt()** method (equivalent to invoking **fireUntilHalt()** on a KIE session). By default, the decision engine runs in *passive mode* and evaluates rule units only when a user or an application explicitly calls **run()** (or **fireAllRules()** for standard rules). If a user or application calls **runUntilHalt()** for rule units (or **fireUntilHalt()** for standard rules), the decision engine starts in *active mode* and evaluates rule units continually until the user or application explicitly calls **halt()**.

If you use the **runUntilHalt()** method, invoke the method on a separate execution thread to avoid blocking the main thread:

Example rule unit execution with runUntilHalt() on a separate thread

```
new Thread( () -> executor.runUntilHalt( adultUnit ) ).start();
```

4.7.1. Data sources for rule units

A rule unit data source is a source of the data processed by a given rule unit and represents the entry point that the decision engine uses to evaluate the rule unit. A rule unit can have zero or more data

sources and each **DataSource** definition declared inside a rule unit can correspond to a different entry point into the rule unit executor. Multiple rule units can share a single data source, but each rule unit must use different entry points through which the same objects are inserted.

You can create a **DataSource** definition with a fixed set of data in a rule unit class, as shown in the following example:

Example data source definition

```
DataSource<Person> persons = DataSource.create( new Person( "John", 42 ),
                                                new Person( "Jane", 44 ),
                                                new Person( "Sally", 4 ) );
```

Because a data source represents the entry point of the rule unit, you can insert, update, or delete facts in a rule unit:

Example code to insert, modify, and delete a fact in a rule unit

```
// Insert a fact:
Person john = new Person( "John", 42 );
FactHandle johnFh = persons.insert( john );

// Modify the fact and optionally specify modified properties (for property reactivity):
john.setAge( 43 );
persons.update( johnFh, john, "age" );

// Delete the fact:
persons.delete( johnFh );
```

4.7.2. Rule unit execution control

Rule units are helpful when you want to coordinate rule execution so that the execution of one rule unit triggers the start of another rule unit and so on.

To facilitate rule unit execution control, the decision engine supports the following rule unit methods that you can use in DRL rule actions to coordinate the execution of rule units:

- **drools.run()**: Triggers the execution of a specified rule unit class. This method imperatively interrupts the execution of the rule unit and activates the other specified rule unit.
- **drools.guard()**: Prevents (guards) a specified rule unit class from being executed until the associated rule condition is met. This method declaratively schedules the execution of the other specified rule unit. When the decision engine produces at least one match for the condition in the guarding rule, the guarded rule unit is considered active. A rule unit can contain multiple guarding rules.

As an example of the **drools.run()** method, consider the following DRL rules that each belong to a specified rule unit. The **NotAdult** rule uses the **drools.run(AdultUnit.class)** method to trigger the execution of the **AdultUnit** rule unit:

Example DRL rules with controlled execution using drools.run()

```
package org.mypackage.myunit
unit AdultUnit
```

```
rule Adult
  when
    Person(age >= 18, $name : name) from persons
  then
    System.out.println($name + " is adult");
end
```

```
package org.mypackage.myunit
unit NotAdultUnit

rule NotAdult
  when
    $p : Person(age < 18, $name : name) from persons
  then
    System.out.println($name + " is NOT adult");
    modify($p) { setAge(18); }
    drools.run( AdultUnit.class );
end
```

The example also uses a **RuleUnitExecutor** class created from the KIE base that was built from these rules and a **DataSource** definition of **persons** bound to it:

Example rule executor and data source definitions

```
RuleUnitExecutor executor = RuleUnitExecutor.create().bind( kbase );
DataSource<Person> persons = executor.newDataSource( "persons",
                                        new Person( "John", 42 ),
                                        new Person( "Jane", 44 ),
                                        new Person( "Sally", 4 ) );
```

In this case, the example creates the **DataSource** definition directly from the **RuleUnitExecutor** class and binds it to the **"persons"** variable in a single statement.

The example execution code produces the following output when the relevant **Person** facts are inserted in the **persons** data source:

Example rule unit execution output

```
Sally is NOT adult
John is adult
Jane is adult
Sally is adult
```

The **NotAdult** rule detects a match when evaluating the person **"Sally"**, who is under 18 years old. The rule then modifies her age to **18** and uses the **drools.run(AdultUnit.class)** method to trigger the execution of the **AdultUnit** rule unit. The **AdultUnit** rule unit contains a rule that can now be executed for all of the 3 **persons** in the **DataSource** definition.

As an example of the **drools.guard()** method, consider the following **BoxOffice** class and **BoxOfficeUnit** rule unit class:

Example BoxOffice class

```
public class BoxOffice {
```

```
    private boolean open;

    public BoxOffice( boolean open ) {
        this.open = open;
    }

    public boolean isOpen() {
        return open;
    }

    public void setOpen( boolean open ) {
        this.open = open;
    }
}
```

Example BoxOfficeUnit rule unit class

```
public class BoxOfficeUnit implements RuleUnit {
    private DataSource<BoxOffice> boxOffices;

    public DataSource<BoxOffice> getBoxOffices() {
        return boxOffices;
    }
}
```

The example also uses the following **TicketIssuerUnit** rule unit class to keep selling box office tickets for the event as long as at least one box office is open. This rule unit uses **DataSource** definitions of **persons** and **tickets**:

Example TicketIssuerUnit rule unit class

```
public class TicketIssuerUnit implements RuleUnit {
    private DataSource<Person> persons;
    private DataSource<AdultTicket> tickets;

    private List<String> results;

    public TicketIssuerUnit() { }

    public TicketIssuerUnit( DataSource<Person> persons, DataSource<AdultTicket> tickets ) {
        this.persons = persons;
        this.tickets = tickets;
    }

    public DataSource<Person> getPersons() {
        return persons;
    }

    public DataSource<AdultTicket> getTickets() {
        return tickets;
    }

    public List<String> getResults() {
```

```
        return results;
    }
}
```

The **BoxOfficeUnit** rule unit contains a **BoxOfficeIsOpen** DRL rule that uses the **drools.guard(TicketIssuerUnit.class)** method to guard the execution of the **TicketIssuerUnit** rule unit that distributes the event tickets, as shown in the following DRL rule examples:

Example DRL rules with controlled execution using drools.guard()

```
package org.mypackage.myunit;
unit TicketIssuerUnit;

rule IssueAdultTicket when
    $p: /persons[ age >= 18 ]
then
    tickets.insert(new AdultTicket($p));
end
rule RegisterAdultTicket when
    $t: /tickets
then
    results.add( $t.getPerson().getName() );
end
```

```
package org.mypackage.myunit;
unit BoxOfficeUnit;

rule BoxOfficeIsOpen
  when
    $box: /boxOffices[ open ]
  then
    drools.guard( TicketIssuerUnit.class );
end
```

In this example, so long as at least one box office is **open**, the guarded **TicketIssuerUnit** rule unit is active and distributes event tickets. When no more box offices are in **open** state, the guarded **TicketIssuerUnit** rule unit is prevented from being executed.

The following example class illustrates a more complete box office scenario:

Example class for the box office scenario

```
DataSource<Person> persons = executor.newDataSource( "persons" );
DataSource<BoxOffice> boxOffices = executor.newDataSource( "boxOffices" );
DataSource<AdultTicket> tickets = executor.newDataSource( "tickets" );

List<String> list = new ArrayList<>();
executor.bindVariable( "results", list );

// Two box offices are open:
BoxOffice office1 = new BoxOffice(true);
FactHandle officeFH1 = boxOffices.insert( office1 );
BoxOffice office2 = new BoxOffice(true);
FactHandle officeFH2 = boxOffices.insert( office2 );
```

```
persons.insert(new Person("John", 40));

// Execute `BoxOfficeIsOpen` rule, run `TicketIssuerUnit` rule unit, and execute `RegisterAdultTicket`
rule:
executor.run(BoxOfficeUnit.class);

assertEquals( 1, list.size() );
assertEquals( "John", list.get(0) );
list.clear();

persons.insert(new Person("Matteo", 30));

// Execute `RegisterAdultTicket` rule:
executor.run(BoxOfficeUnit.class);

assertEquals( 1, list.size() );
assertEquals( "Matteo", list.get(0) );
list.clear();

// One box office is closed, the other is open:
office1.setOpen(false);
boxOffices.update(officeFH1, office1);
persons.insert(new Person("Mark", 35));
executor.run(BoxOfficeUnit.class);

assertEquals( 1, list.size() );
assertEquals( "Mark", list.get(0) );
list.clear();

// All box offices are closed:
office2.setOpen(false);
boxOffices.update(officeFH2, office2); // Guarding rule is no longer true.
persons.insert(new Person("Edson", 35));
executor.run(BoxOfficeUnit.class); // No execution

assertEquals( 0, list.size() );
```

4.7.3. Rule unit identity conflicts

In rule unit execution scenarios with guarded rule units, a rule can guard multiple rule units and at the same time a rule unit can be guarded and then activated by multiple rules. For these two-way guarding scenarios, rule units must have a clearly defined identity to avoid identity conflicts.

By default, the identity of a rule unit is the rule unit class name and is treated as a singleton class by the **RuleUnitExecutor**. This identification behavior is encoded in the **getUnitIdentity()** default method of the **RuleUnit** interface:

Default identity method in the RuleUnit interface

```
default Identity getUnitIdentity() {
    return new Identity( getClass() );
}
```

In some cases, you may need to override this default identification behavior to avoid conflicting identities between rule units.

For example, the following **RuleUnit** class contains a **DataSource** definition that accepts any kind of object:

Example Unit0 rule unit class

```
public class Unit0 implements RuleUnit {
    private DataSource<Object> input;

    public DataSource<Object> getInput() {
        return input;
    }
}
```

This rule unit contains the following DRL rule that guards another rule unit based on two conditions (in OOPath notation):

Example GuardAgeCheck DRL rule in the rule unit

```
package org.mypackage.myunit
unit Unit0

rule GuardAgeCheck
  when
    $i: /input#Integer
    $s: /input#String
  then
    drools.guard( new AgeCheckUnit($i) );
    drools.guard( new AgeCheckUnit($s.length()) );
end
```

The guarded **AgeCheckUnit** rule unit verifies the age of a set of **persons**. The **AgeCheckUnit** contains a **DataSource** definition of the **persons** to check, a **minAge** variable that it verifies against, and a **List** for gathering the results:

Example AgeCheckUnit rule unit

```
public class AgeCheckUnit implements RuleUnit {
    private final int minAge;
    private DataSource<Person> persons;
    private List<String> results;

    public AgeCheckUnit( int minAge ) {
        this.minAge = minAge;
    }

    public DataSource<Person> getPersons() {
        return persons;
    }

    public int getMinAge() {
        return minAge;
    }

    public List<String> getResults() {
```

```
        return results;
    }
}
```

The **AgeCheckUnit** rule unit contains the following DRL rule that performs the verification of the **persons** in the data source:

Example CheckAge DRL rule in the rule unit

```
package org.mypackage.myunit
unit AgeCheckUnit

rule CheckAge
  when
    $p : /persons{ age > minAge }
  then
    results.add($p.getName() + ">" + minAge);
end
```

This example creates a **RuleUnitExecutor** class, binds the class to the KIE base that contains these two rule units, and creates the two **DataSource** definitions for the same rule units:

Example executor and data source definitions

```
RuleUnitExecutor executor = RuleUnitExecutor.create().bind( kbase );

DataSource<Object> input = executor.newDataSource( "input" );
DataSource<Person> persons = executor.newDataSource( "persons",
                                        new Person( "John", 42 ),
                                        new Person( "Sally", 4 ) );

List<String> results = new ArrayList<>();
executor.bindVariable( "results", results );
```

You can now insert some objects into the input data source and execute the **Unit0** rule unit:

Example rule unit execution with inserted objects

```
ds.insert("test");
ds.insert(3);
ds.insert(4);
executor.run(Unit0.class);
```

Example results list from the execution

```
[Sally>3, John>3]
```

In this example, the rule unit named **AgeCheckUnit** is considered a singleton class and then executed only once, with the **minAge** variable set to **3**. Both the String **"test"** and the Integer **4** inserted into the input data source can also trigger a second execution with the **minAge** variable set to **4**. However, the second execution does not occur because another rule unit with the same identity has already been evaluated.

To resolve this rule unit identity conflict, override the **getUnitIdentity()** method in the **AgeCheckUnit** class to include also the **minAge** variable in the rule unit identity:

Modified AgeCheckUnit rule unit to override the getUnitIdentity() method

```
public class AgeCheckUnit implements RuleUnit {

    ...

    @Override
    public Identity getUnitIdentity() {
        return new Identity(getClass(), minAge);
    }
}
```

With this override in place, the previous example rule unit execution produces the following output:

Example results list from executing the modified rule unit

```
[John>4, Sally>3, John>3]
```

The rule units with **minAge** set to **3** and **4** are now considered two different rule units and both are executed.

CHAPTER 5. PHREAK RULE ALGORITHM IN THE DECISION ENGINE

The decision engine in Red Hat Decision Manager uses the Phreak algorithm for rule evaluation. Phreak evolved from the Rete algorithm, including the enhanced Rete algorithm ReteOO that was introduced in previous versions of Red Hat Decision Manager for object-oriented systems. Overall, Phreak is more scalable than Rete and ReteOO, and is faster in large systems.

While Rete is considered eager (immediate rule evaluation) and data oriented, Phreak is considered lazy (delayed rule evaluation) and goal oriented. The Rete algorithm performs many actions during the insert, update, and delete actions in order to find partial matches for all rules. This eagerness of the Rete algorithm during rule matching requires a lot of time before eventually executing rules, especially in large systems. With Phreak, this partial matching of rules is delayed deliberately to handle large amounts of data more efficiently.

The Phreak algorithm adds the following set of enhancements to previous Rete algorithms:

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

5.1. RULE EVALUATION IN PHREAK

When the decision engine starts, all rules are considered to be *unlinked* from pattern-matching data that can trigger the rules. At this stage, the Phreak algorithm in the decision engine does not evaluate the rules. The **insert**, **update**, and **delete** actions are queued, and Phreak uses a heuristic, based on the rule most likely to result in execution, to calculate and select the next rule for evaluation. When all the required input values are populated for a rule, the rule is considered to be *linked* to the relevant pattern-matching data. Phreak then creates a goal that represents this rule and places the goal into a priority queue that is ordered by rule salience. Only the rule for which the goal was created is evaluated, and other potential rule evaluations are delayed. While individual rules are evaluated, node sharing is still achieved through the process of segmentation.

Unlike the tuple-oriented Rete, the Phreak propagation is collection oriented. For the rule that is being evaluated, the decision engine accesses the first node and processes 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 delete actions are processed, adding the results to the same set. The set is then propagated to the next child node and the same process repeats until it reaches the terminal node. This cycle creates a batch process effect that can provide performance advantages for certain rule constructs.

The linking and unlinking of rules happens through a layered bit-mask system, based on network segmentation. When the rule network is built, segments are created for rule network nodes that are shared by the same set of rules. A rule is composed of a path of segments. In case a rule does not share any node with any other rule, it becomes a single segment.

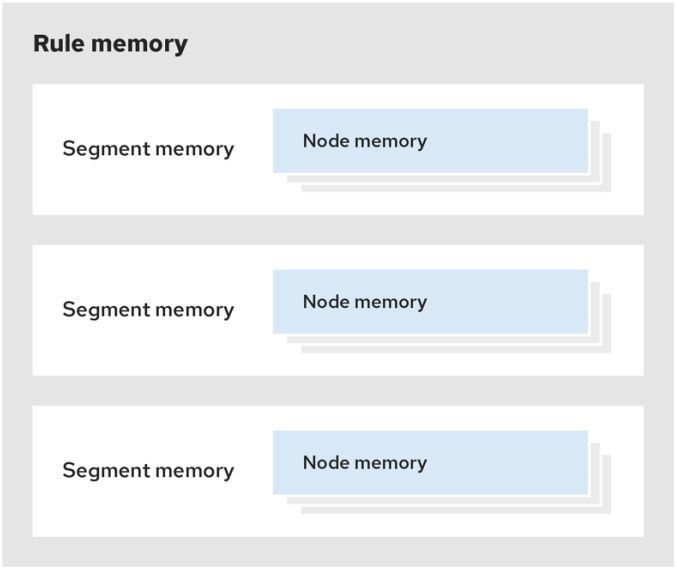
A bit-mask offset is assigned to each node in the segment. Another bit mask is assigned to each segment in the path of the rule according to these requirements:

- If at least one input for a node exists, the node bit is set to the **on** state.
- If each node in a segment has the bit set to the **on** state, the segment bit is also set to the **on** state.
- If any node bit is set to the **off** state, the segment is also set to the **off** state.
- If each segment in the path of the rule is set to the **on** state, the rule is considered linked, and a goal is created to schedule the rule for evaluation.

The same bit-mask technique is used to track modified nodes, segments, and rules. This tracking ability enables an already linked rule to be unscheduled from evaluation if it has been modified since the evaluation goal for it was created. As a result, no rules can ever evaluate partial matches.

This process of rule evaluation is possible in Phreak because, as opposed to a single unit of memory in Rete, Phreak has three layers of contextual memory with node, segment, and rule memory types. This layering enables much more contextual understanding during the evaluation of a rule.

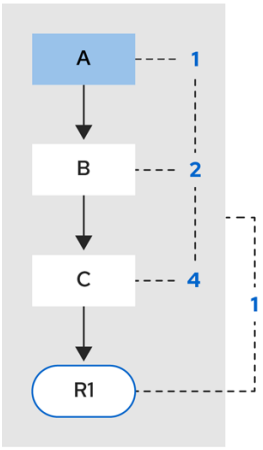
Figure 5.1. Phreak three-layered memory system



The following examples illustrate how rules are organized and evaluated in this three-layered memory system in Phreak.

Example 1: A single rule (R1) with three patterns: A, B and C. The rule forms a single segment, with bits 1, 2, and 4 for the nodes. The single segment has a bit offset of 1.

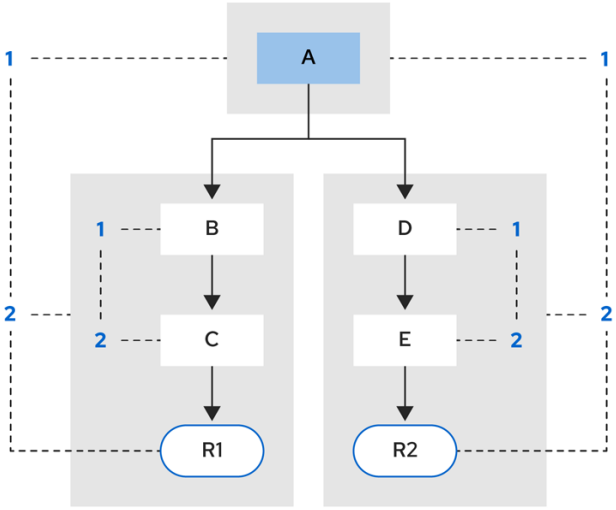
Figure 5.2. Example 1: Single rule



RHDM_29_0319

Example 2: Rule R2 is added and shares pattern A.

Figure 5.3. Example 2: Two rules with pattern sharing



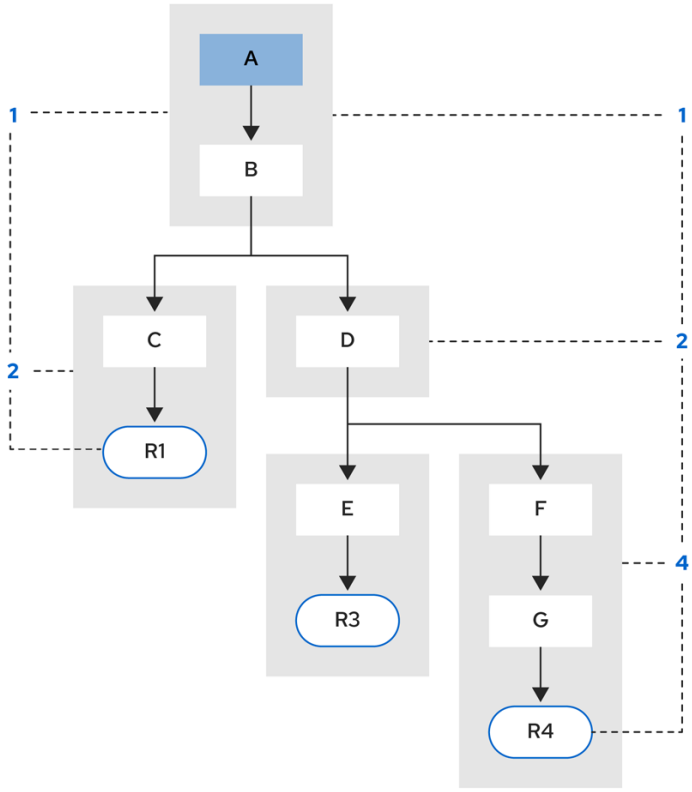
RHDM_29_0319

Pattern A is placed in its own segment, resulting in two segments for each rule. Those two segments form a path for their respective rules. The first segment is shared by both paths. When pattern A is linked, the segment becomes linked. The segment then iterates over each path that the segment is shared by, setting the bit 1 to **on**. If patterns B and C are later turned on, the second segment for path R1 is linked, and this causes bit 2 to be turned on for R1. With bit 1 and bit 2 turned on for R1, the rule is now linked and a goal is created to schedule the rule for later evaluation and execution.

When a rule is evaluated, the segments enable the results of the matching to be shared. Each segment has a staging memory to queue all inserts, updates, and deletes for that segment. When R1 is evaluated, the rule processes pattern A, and this results in a set of tuples. The algorithm detects a segmentation split, creates peered tuples for each insert, update, and delete in the set, and adds them to the R2 staging memory. Those tuples are then merged with any existing staged tuples and are executed when R2 is eventually evaluated.

Example 3: Rules R3 and R4 are added and share patterns A and B.

Figure 5.4. Example 3: Three rules with pattern sharing

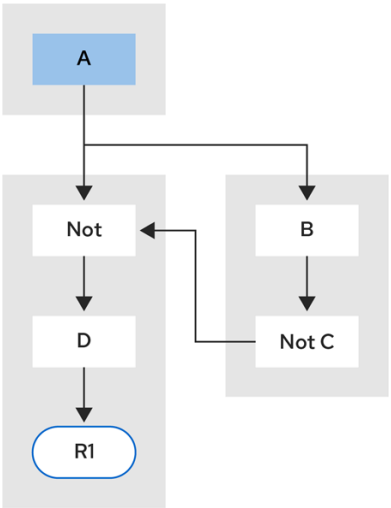


RHDM_29_0319

Rules R3 and R4 have three segments and R1 has two segments. Patterns A and B are shared by R1, R3, and R4, while pattern D is shared by R3 and R4.

Example 4: A single rule (R1) with a subnetwork and no pattern sharing.

Figure 5.5. Example 4: Single rule with a subnetwork and no pattern sharing

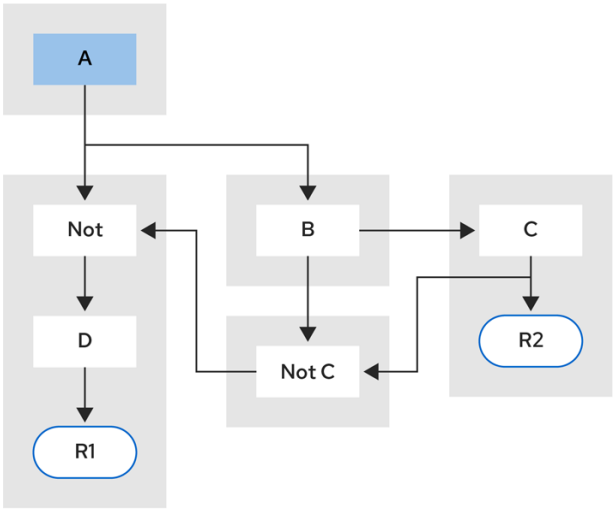


RHDM_29_0319

Subnetworks are formed when a **Not**, **Exists**, or **Accumulate** node contains more than one element. In this example, the element **B not(C)** forms the subnetwork. The element **not(C)** is a single element that does not require a subnetwork and is therefore merged inside of the **Not** node. The subnetwork uses a dedicated segment. Rule R1 still has a path of two segments and the subnetwork forms another inner path. When the subnetwork is linked, it is also linked in the outer segment.

Example 5: Rule R1 with a subnetwork that is shared by rule R2.

Figure 5.6. Example 5: Two rules, one with a subnetwork and pattern sharing



The subnetwork nodes in a rule can be shared by another rule that does not have a subnetwork. This sharing causes the subnetwork segment to be split into two segments.

Constrained **Not** nodes and **Accumulate** nodes can never unlink a segment, and are always considered to have their bits turned on.

The Phreak evaluation algorithm is stack based instead of method-recursion based. Rule evaluation can be paused and resumed at any time when a **StackEntry** is used to represent the node currently being evaluated.

When a rule evaluation reaches a subnetwork, a **StackEntry** object is created for the outer path segment and the subnetwork segment. The subnetwork segment is evaluated first, and when the set reaches the end of the subnetwork path, the segment is merged into a staging list for the outer node that the segment feeds into. The previous **StackEntry** object is then resumed and can now process the results of the subnetwork. This process has the added benefit, especially for **Accumulate** nodes, that all work is completed in a batch, before propagating to the child node.

The same stack system is used for efficient backward chaining. When a rule evaluation reaches a query node, the evaluation is paused and the query is added to the stack. The query is then evaluated to produce a result set, which is saved in a memory location for the resumed **StackEntry** object to pick up and propagate to the child node. If the query itself called other queries, the process repeats, while the current query is paused and a new evaluation is set up for the current query node.

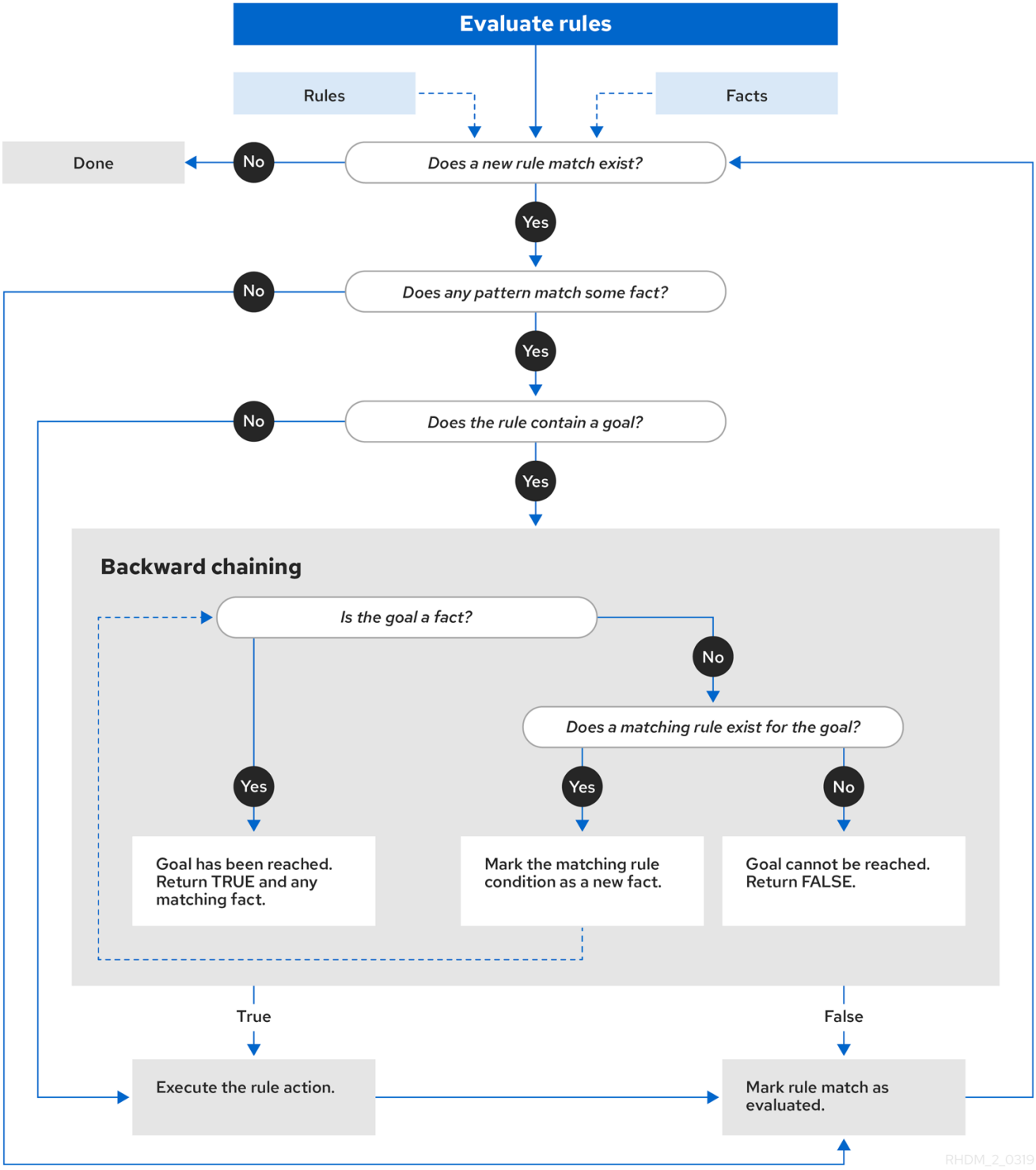
5.1.1. Rule evaluation with forward and backward chaining

The decision engine in Red Hat Decision Manager is a hybrid reasoning system that uses both forward chaining and backward chaining to evaluate rules. A forward-chaining rule system is a data-driven system that starts with a fact in the working memory of the decision engine and reacts to changes to that fact. When objects are inserted into working memory, any rule conditions that become true as a result of the change are scheduled for execution by the agenda.

In contrast, a backward-chaining rule system is a goal-driven system that starts with a conclusion that the decision engine attempts to satisfy, often using recursion. If the system cannot reach the conclusion or goal, it searches for subgoals, which are conclusions that complete part of the current goal. The system continues this process until either the initial conclusion is satisfied or all subgoals are satisfied.

The following diagram illustrates how the decision engine evaluates rules using forward chaining overall with a backward-chaining segment in the logic flow:

Figure 5.7. Rule evaluation logic using forward and backward chaining



5.2. RULE BASE CONFIGURATION

Red Hat Decision Manager contains a **RuleBaseConfiguration.java** object that you can use to configure exception handler settings, multithreaded execution, and sequential mode in the decision engine.

For the rule base configuration options, download the **Red Hat Decision Manager 7.5.1 Source Distribution** ZIP file from the Red Hat Customer Portal and navigate to **~/rhdm-7.5.1-sources/src/drools-$VERSION/drools-core/src/main/java/org/drools/core/RuleBaseConfiguration.java**.

The following rule base configuration options are available for the decision engine:

drools.consequenceExceptionHandler

When configured, this system property defines the class that manages the exceptions thrown by rule consequences. You can use this property to specify a custom exception handler for rule evaluation in the decision engine.
Default value: **org.drools.core.runtime.rule.impl.DefaultConsequenceExceptionHandler**

You can specify the custom exception handler using one of the following options:

- Specify the exception handler in a system property:

```
drools.consequenceExceptionHandler=org.drools.core.runtime.rule.impl.MyCustomConsequenceExceptionHandler
```

- Specify the exception handler while creating the KIE base programatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
kieBaseConf.setOption(ConsequenceExceptionHandlerOption.get(MyCustomConsequenceExceptionHandler.class));
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

drools.multithreadEvaluation

When enabled, this system property enables the decision engine to evaluate rules in parallel by dividing the Phreak rule network into independent partitions. You can use this property to increase the speed of rule evaluation for specific rule bases.
Default value: **false**

You can enable multithreaded evaluation using one of the following options:

- Enable the multithreaded evaluation system property:

```
drools.multithreadEvaluation=true
```

- Enable multithreaded evaluation while creating the KIE base programatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
kieBaseConf.setOption(MultithreadEvaluationOption.YES);
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

WARNING

Rules that use queries, salience, or agenda groups are currently not supported by the parallel decision engine. If these rule elements are present in the KIE base, the compiler emits a warning and automatically switches back to single-threaded evaluation. However, in some cases, the decision engine might not detect the unsupported rule elements and rules might be evaluated incorrectly. For example, the decision engine might not detect when rules rely on implicit salience given by rule ordering inside the DRL file, resulting in incorrect evaluation due to the unsupported salience attribute.

drools.sequential

When enabled, this system property enables sequential mode in the decision engine. In sequential mode, the decision engine evaluates rules one time in the order that they are listed in the decision engine agenda without regard to changes in the working memory. This means that the decision engine ignores any **insert**, **modify**, or **update** statements in rules and executes rules in a single sequence. As a result, rule execution may be faster in sequential mode, but important updates may not be applied to your rules. You can use this property if you use stateless KIE sessions and you do not want the execution of rules to influence subsequent rules in the agenda. Sequential mode applies to stateless KIE sessions only.
Default value: **false**

You can enable sequential mode using one of the following options:

- Enable the sequential mode system property:

```
drools.sequential=true
```

- Enable sequential mode while creating the KIE base programatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
kieBaseConf.setOption(SequentialOption.YES);
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

- Enable sequential mode in the KIE module descriptor file (**kmodule.xml**) for a specific Red Hat Decision Manager project:

```
<kmodule>
  ...
  <kbase name="KBase2" default="false" sequential="true" packages="org.domain.pkg2,
org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

5.3. SEQUENTIAL MODE IN PHREAK

Sequential mode is an advanced rule base configuration in the decision engine, supported by Phreak, that enables the decision engine to evaluate rules one time in the order that they are listed in the decision engine agenda without regard to changes in the working memory. In sequential mode, the decision engine ignores any **insert**, **modify**, or **update** statements in rules and executes rules in a single sequence. As a result, rule execution may be faster in sequential mode, but important updates may not be applied to your rules.

Sequential mode applies to only stateless KIE sessions because stateful KIE sessions inherently use data from previously invoked KIE sessions. If you use a stateless KIE session and you want the execution of rules to influence subsequent rules in the agenda, then do not enable sequential mode. Sequential mode is disabled by default in the decision engine.

To enable sequential mode, use one of the following options:

- Set the system property **drools.sequential** to **true**.
- Enable sequential mode while creating the KIE base programatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
kieBaseConf.setOption(SequentialOption.YES);
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

- Enable sequential mode in the KIE module descriptor file (**kmodule.xml**) for a specific Red Hat Decision Manager project:

```
<kmodule>
  ...
  <kbase name="KBase2" default="false" sequential="true" packages="org.domain.pkg2,
org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

To configure sequential mode to use a dynamic agenda, use one of the following options:

- Set the system property **drools.sequential.agenda** to **dynamic**.
- Set the sequential agenda option while creating the KIE base programatically:

```
KieServices ks = KieServices.Factory.get();
KieBaseConfiguration kieBaseConf = ks.newKieBaseConfiguration();
kieBaseConf.setOption(SequentialAgendaOption.DYNAMIC);
KieBase kieBase = kieContainer.newKieBase(kieBaseConf);
```

When you enable sequential mode, the decision engine evaluates rules in the following way:

1. Rules are ordered by salience and position in the rule set.
2. An element for each possible rule match is created. The element position indicates the execution order.
3. Node memory is disabled, with the exception of the right-input object memory.

4. The left-input adapter node propagation is disconnected and the object with the node is referenced in a **Command** object. The **Command** object is added to a list in the working memory for later execution.

5. All objects are asserted, and then the list of **Command** objects is checked and executed.

6. All matches that result from executing the list are added to elements based on the sequence number of the rule.

7. The elements that contain matches are executed in a sequence. If you set a maximum number of rule executions, the decision engine activates no more than that number of rules in the agenda for execution.

In sequential mode, the **LeftInputAdapterNode** node creates a **Command** object and adds it to a list in the working memory of the decision engine. This **Command** object contains references to the **LeftInputAdapterNode** node and the propagated object. These references stop any left-input propagations at insertion time so that the right-input propagation never needs to attempt to join the left inputs. The references also avoid the need for the left-input memory.

All nodes have their memory turned off, including the left-input tuple memory, but excluding the right-input object memory. After all the assertions are finished and the right-input memory of all the objects is populated, the decision engine iterates over the list of **LeftInputAdatperNode Command** objects. The objects propagate down the network, attempting to join the right-input objects, but they are not retained in the left input.

The agenda with a priority queue to schedule the tuples is replaced by an element for each rule. The sequence number of the **RuleTerminalNode** node indicates the element where to place the match. After all **Command** objects have finished, the elements are checked and existing matches are executed. To improve performance, the first and the last populated cell in the elements are retained.

When the network is constructed, each **RuleTerminalNode** node receives a sequence number based on its salience number and the order in which it was added to the network.

The right-input node memories are typically hash maps for fast object deletion. Because object deletions are not supported, Phreak uses an object list when the values of the object are not indexed. For a large number of objects, indexed hash maps provide a performance increase. If an object has only a few instances, Phreak uses an object list instead of an index.

CHAPTER 6. COMPLEX EVENT PROCESSING (CEP)

In Red Hat Decision Manager, an event is a record of a significant change of state in the application domain at a point in time. Depending on how the domain is modeled, the change of state may be represented by a single event, multiple atomic events, or hierarchies of correlated events. From a complex event processing (CEP) perspective, an event is a type of fact or object that occurs at a specific point in time, and a business rule is a definition of how to react to the data from that fact or object. For example, in a stock broker application, 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 because a change has occurred in the state of the application domain at a given time.

The decision engine in Red Hat Decision Manager uses complex event processing (CEP) to detect and process multiple events within a collection of events, to uncover relationships that exist between events, and to infer new data from the events and their relationships.

CEP use cases share several requirements and goals with business rule use cases.

From a business perspective, business rule definitions are often defined based on the occurrence of scenarios triggered by events. In the following examples, events form the basis of business rules:

- In an algorithmic trading application, a rule performs an action if the security price increases by X percent above the day opening price. The price increases are denoted by events on a stock trading application.
- In a monitoring application, a rule performs an action if the temperature in the server room increases X degrees in Y minutes. The sensor readings are denoted by events.

From a technical perspective, business rule evaluation and CEP have the following key similarities:

- Both business rule evaluation and CEP require seamless integration with the enterprise infrastructure and applications. This is particularly important with life-cycle management, auditing, and security.
- Both business rule evaluation and CEP have functional requirements such as pattern matching, and non-functional requirements such as response time limits and query-rule explanations.

CEP scenarios have the following key characteristics:

- Scenarios usually process large numbers of events, but only a small percentage of the events are relevant.
- Events are usually immutable and represent a record of change in state.
- Rules and queries run against events and must react to detected event patterns.
- Related events usually have a strong temporal relationship.
- Individual events are not prioritized. The CEP system prioritizes patterns of related events and the relationships between them.
- Events usually need to be composed and aggregated.

Given these common CEP scenario characteristics, the CEP system in Red Hat Decision Manager supports the following features and functions to optimize event processing:

- Event processing with proper semantics

- Event detection, correlation, aggregation, and composition
- Event stream processing
- Temporal constraints to model the temporal relationships between events
- Sliding windows of significant events
- Session-scoped unified clock
- Required volumes of events for CEP use cases
- Reactive rules
- Adapters for event input into the decision engine (pipeline)

6.1. EVENTS IN COMPLEX EVENT PROCESSING

In Red Hat Decision Manager, an event is a record of a significant change of state in the application domain at a point in time. Depending on how the domain is modeled, the change of state may be represented by a single event, multiple atomic events, or hierarchies of correlated events. From a complex event processing (CEP) perspective, an event is a type of fact or object that occurs at a specific point in time, and a business rule is a definition of how to react to the data from that fact or object. For example, in a stock broker application, 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 because a change has occurred in the state of the application domain at a given time.

Events have the following key characteristics:

- **Are immutable:** An event is a record of change that has occurred at some time in the past and cannot be changed.

NOTE

The decision engine does not enforce immutability on the Java objects that represent events. This behavior makes event data enrichment possible. Your application should be able to populate unpopulated event attributes, and these attributes are used by the decision engine to enrich the event with inferred data. However, you should not change event attributes that have already been populated.

- **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.
- **Have managed life cycles:** Because events are immutable and have temporal constraints, they are usually only relevant for a specified period of time. This means that the decision engine can automatically manage the life cycle of events.
- **Can use sliding windows:** You can define sliding windows of time or length with events. A sliding time window is a specified period of time during which events can be processed. A sliding length window is a specified number of events that can be processed.

6.2. DECLARING FACTS AS EVENTS

You can declare facts as events in your Java class or DRL rule file so that the decision engine handles

the facts as events during complex event processing. You can declare the facts as interval-based events or point-in-time events. Interval-based events have a duration time and persist in the working memory of the decision engine until their duration time has lapsed. Point-in-time events have no duration and are essentially interval-based events with a duration of zero.

Procedure

For the relevant fact type in your Java class or DRL rule file, enter the **@role(event)** metadata tag and parameter. The **@role** metadata tag accepts the following two values:

- **fact**: (Default) Declares the type as a regular fact
- **event**: Declares the type as an event

For example, the following snippet declares that the **StockPoint** fact type in a stock broker application must be handled as an event:

Declare fact type as an event

```
import some.package.StockPoint

declare StockPoint
  @role( event )
end
```

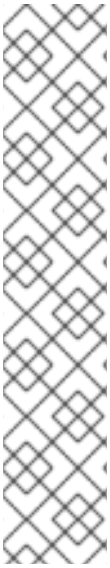
If **StockPoint** is a fact type declared in the DRL rule file instead of in a pre-existing class, you can declare the event in-line in your application code:

Declare fact type in-line and assign it to event role

```
declare StockPoint
  @role( event )

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

6.3. METADATA TAGS FOR EVENTS

The decision engine uses the following metadata tags for events that are inserted into the working memory of the decision engine. You can change the default metadata tag values in your Java class or DRL rule file as needed.



NOTE

The examples in this section that refer to the **VoiceCall** class assume that the sample application domain model includes the following class details:

VoiceCall fact class in an example 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

This tag determines whether a given fact type is handled as a regular fact or an event in the decision engine during complex event processing.
Default parameter: **fact**

Supported parameters: **fact**, **event**

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

Example: Declare VoiceCall as event type

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

@timestamp

This tag is automatically assigned to every event in the decision engine. By default, the time is provided by the session clock and assigned to the event when it is inserted into the working memory of the decision engine. You can specify a custom time stamp attribute instead of the default time stamp added by the session clock.
Default parameter: The time added by the decision engine session clock

Supported parameters: Session clock time or custom time stamp attribute

```
@timestamp( <attributeName> )
```

Example: Declare VoiceCall timestamp attribute

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

@duration

This tag determines the duration time for events in the decision engine. Events can be interval-

based events or point-in-time events. Interval-based events have a duration time and persist in the working memory of the decision engine until their duration time has lapsed. Point-in-time events have no duration and are essentially interval-based events with a duration of zero. By default, every event in the decision engine has a duration of zero. You can specify a custom duration attribute instead of the default.
Default parameter: Null (zero)

Supported parameters: Custom duration attribute

```
@duration( <attributeName> )
```

Example: Declare VoiceCall duration attribute

```
declare VoiceCall
  @role( event )
  @timestamp( callDateTime )
  @duration( callDuration )
end
```

@expires

This tag determines the time duration before an event expires in the working memory of the decision engine. By default, an event expires when the event can no longer match and activate any of the current rules. You can define an amount of time after which an event should expire. This tag definition also overrides the implicit expiration offset calculated from temporal constraints and sliding windows in the KIE base. This tag is available only when the decision engine is running in stream mode.
Default parameter: Null (event expires after event can no longer match and activate rules)

Supported parameters: Custom **timeOffset** attribute in the format **[#d][#h][#m][#s][[ms]]**

```
@expires( <timeOffset> )
```

Example: Declare expiration offset for VoiceCall events

```
declare VoiceCall
  @role( event )
  @timestamp( callDateTime )
  @duration( callDuration )
  @expires( 1h35m )
end
```

6.4. EVENT PROCESSING MODES IN THE DECISION ENGINE

The decision engine runs in either cloud mode or stream mode. In cloud mode, the decision engine processes facts as facts with no temporal constraints, independent of time, and in no particular order. In stream mode, the decision engine processes facts as events with strong temporal constraints, in real time or near real time. Stream mode uses synchronization to make event processing possible in Red Hat Decision Manager.

Cloud mode

Cloud mode is the default operating mode of the decision engine. In cloud mode, the decision engine

treats events as an unordered cloud. Events still have time stamps, but the decision engine running in cloud mode cannot draw relevance from the time stamp because cloud mode ignores the present time. This mode uses the rule constraints to find the matching tuples to activate and execute rules. Cloud mode does not impose any kind of additional requirements on facts. However, because the decision engine in this mode has no concept of time, it cannot use temporal features such as sliding windows or automatic life-cycle management. In cloud mode, events must be explicitly retracted when they are no longer needed.

The following requirements are not imposed in cloud mode:

- No clock synchronization because the decision engine has no notion of time
- No ordering of events because the decision engine processes events as an unordered cloud, against which the decision engine match rules

You can specify cloud mode either by setting the system property in the relevant configuration files or by using the Java client API:

Set cloud mode using system property

```
drools.eventProcessingMode=cloud
```

Set cloud mode using Java client API

```
import org.kie.api.conf.EventProcessingOption;
import org.kie.api.KieBaseConfiguration;
import org.kie.api.KieServices.Factory;

KieBaseConfiguration config = KieServices.Factory.get().newKieBaseConfiguration();

config.setOption(EventProcessingOption.CLOUD);
```

You can also specify cloud mode using the **eventProcessingMode="<mode>"** KIE base attribute in the KIE module descriptor file (**kmodule.xml**) for a specific Red Hat Decision Manager project:

Set cloud mode using project kmodule.xml file

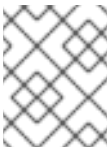
```
<kmodule>
  ...
  <kbase name="KBase2" default="false" eventProcessingMode="cloud"
packages="org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

Stream mode

Stream mode enables the decision engine to process events chronologically and in real time as they are inserted into the decision engine. In stream mode, the decision engine synchronizes streams of events (so that events in different streams can be processed in chronological order), implements sliding windows of time or length, and enables automatic life-cycle management.
The following requirements apply to stream mode:

- Events in each stream must be ordered chronologically.

- A session clock must be present to synchronize event streams.



NOTE

Your application does not need to enforce ordering events between streams, but using event streams that have not been synchronized may cause unexpected results.

You can specify stream mode either by setting the system property in the relevant configuration files or by using the Java client API:

Set stream mode using system property

```
drools.eventProcessingMode=stream
```

Set stream mode using Java client API

```
import org.kie.api.conf.EventProcessingOption;
import org.kie.api.KieBaseConfiguration;
import org.kie.api.KieServices.Factory;

KieBaseConfiguration config = KieServices.Factory.get().newKieBaseConfiguration();

config.setOption(EventProcessingOption.STREAM);
```

You can also specify stream mode using the **eventProcessingMode="<mode>"** KIE base attribute in the KIE module descriptor file (**kmodule.xml**) for a specific Red Hat Decision Manager project:

Set stream mode using project kmodule.xml file

```
<kmodule>
  ...
  <kbase name="KBase2" default="false" eventProcessingMode="stream"
packages="org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

6.4.1. Negative patterns in decision engine stream mode

A negative pattern is a pattern for conditions that are not met. For example, the following DRL rule activates a fire alarm if a fire is detected and the sprinkler is not activated:

Fire alarm rule with a negative pattern

```
rule "Sound the alarm"
when
  $f : FireDetected()
  not(SprinklerActivated())
then
  // Sound the alarm.
end
```

In cloud mode, the decision engine assumes all facts (regular facts and events) are known in advance and evaluates negative patterns immediately. In stream mode, the decision engine can support temporal constraints on facts to wait for a set time before activating a rule.

The same example rule in stream mode activates the fire alarm as usual, but applies a 10-second delay.

Fire alarm rule with a negative pattern and time delay (stream mode only)

```
rule "Sound the alarm"
when
  $f : FireDetected()
  not(SprinklerActivated(this after[0s,10s] $f))
then
  // Sound the alarm.
end
```

The following modified fire alarm rule expects one **Heartbeat** event to occur every 10 seconds. If the expected event does not occur, the rule is executed. This rule uses the same type of object in both the first pattern and in the negative pattern. The negative pattern has the temporal constraint to wait 0 to 10 seconds before executing and excludes the **Heartbeat** event bound to **$h** so that the rule can be executed. The bound event **$h** must be explicitly excluded in order for the rule to be executed because the temporal constraint **[0s, ...]** does not inherently exclude that event from being matched again.

Fire alarm rule excluding a bound event in a negative pattern (stream mode only)

```
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
```

6.5. PROPERTY-CHANGE SETTINGS AND LISTENERS FOR FACT TYPES

By default, the decision engine does not re-evaluate all fact patterns for fact types each time a rule is triggered, but instead reacts only to modified properties that are constrained or bound inside a given pattern. For example, if a rule calls **modify()** as part of the rule actions but the action does not generate new data in the KIE base, the decision engine does not automatically re-evaluate all fact patterns because no data was modified. This property reactivity behavior prevents unwanted recursions in the KIE base and results in more efficient rule evaluation. This behavior also means that you do not always need to use the **no-loop** rule attribute to avoid infinite recursion.

You can modify or disable this property reactivity behavior with the following **KnowledgeBuilderConfiguration** options, and then use a property-change setting in your Java class or DRL files to fine-tune property reactivity as needed:

- **ALWAYS**: (Default) All types are property reactive, but you can disable property reactivity for a specific type by using the **@classReactive** property-change setting.
- **ALLOWED**: No types are property reactive, but you can enable property reactivity for a specific type by using the **@propertyReactive** property-change setting.

- **DISABLED**: No types are property reactive. All property-change listeners are ignored.

Example property reactivity setting in KnowledgeBuilderConfiguration

```
KnowledgeBuilderConfiguration config =
KnowledgeBuilderFactory.newKnowledgeBuilderConfiguration();
config.setOption(PropertySpecificOption.ALLOWED);
KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder(config);
```

Alternatively, you can update the **drools.propertySpecific** system property in the **standalone.xml** file of your Red Hat Decision Manager distribution:

Example property reactivity setting in system properties

```
<system-properties>
  ...
  <property name="drools.propertySpecific" value="ALLOWED"/>
  ...
</system-properties>
```

The decision engine supports the following property-change settings and listeners for fact classes or declared DRL fact types:

@classReactive

If property reactivity is set to **ALWAYS** in the decision engine (all types are property reactive), this tag disables the default property reactivity behavior for a specific Java class or a declared DRL fact type. You can use this tag if you want the decision engine to re-evaluate all fact patterns for the specified fact type each time the rule is triggered, instead of reacting only to modified properties that are constrained or bound inside a given pattern.

Example: Disable default property reactivity in a DRL type declaration

```
declare Person
  @classReactive
    firstName : String
    lastName : String
end
```

Example: Disable default property reactivity in a Java class

```
@classReactive
public static class Person {
    private String firstName;
    private String lastName;
}
```

@propertyReactive

If property reactivity is set to **ALLOWED** in the decision engine (no types are property reactive unless specified), this tag enables property reactivity for a specific Java class or a declared DRL fact type. You can use this tag if you want the decision engine to react only to modified properties that are constrained or bound inside a given pattern for the specified fact type, instead of re-evaluating all fact patterns for the fact each time the rule is triggered.

Example: Enable property reactivity in a DRL type declaration (when reactivity is disabled globally)

```
declare Person
  @propertyReactive
    firstName : String
    lastName : String
end
```

Example: Enable property reactivity in a Java class (when reactivity is disabled globally)

```
@propertyReactive
public static class Person {
    private String firstName;
    private String lastName;
}
```

@watch

This tag enables property reactivity for additional properties that you specify in-line in fact patterns in DRL rules. This tag is supported only if property reactivity is set to **ALWAYS** in the decision engine, or if property reactivity is set to **ALLOWED** and the relevant fact type uses the **@propertyReactive** tag. You can use this tag in DRL rules to add or exclude specific properties in fact property reactivity logic.
Default parameter: None

Supported parameters: Property name, ***** (all), **!** (not), **!*** (no properties)

```
<factPattern> @watch ( <property> )
```

Example: Enable or disable property reactivity in fact patterns

```
// Listens for changes in both `firstName` (inferred) and `lastName`:
Person(firstName == $expectedFirstName) @watch( lastName )

// Listens for changes in all properties of the `Person` fact:
Person(firstName == $expectedFirstName) @watch( * )

// Listens for changes in `lastName` and explicitly excludes changes in `firstName`:
Person(firstName == $expectedFirstName) @watch( lastName, !firstName )

// Listens for changes in all properties of the `Person` fact except `age`:
Person(firstName == $expectedFirstName) @watch( *, !age )

// Excludes changes in all properties of the `Person` fact (equivalent to using `@classReactivity`
tag):
Person(firstName == $expectedFirstName) @watch( !* )
```

The decision engine generates a compilation error if you use the **@watch** tag for properties in a fact type that uses the **@classReactive** tag (disables property reactivity) or when property reactivity is set to **ALLOWED** in the decision engine and the relevant fact type does not use the **@propertyReactive** tag. Compilation errors also arise if you duplicate properties in listener annotations, such as **@watch(firstName, ! firstName)**.

@propertyChangeSupport

For facts that implement support for property changes as defined in the JavaBeans Specification, this tag enables the decision engine to monitor changes in the fact properties.

Example: Declare property change support in JavaBeans object

```
declare Person
   @propertyChangeSupport
end
```

6.6. TEMPORAL OPERATORS FOR EVENTS

In stream mode, the decision engine supports the following temporal operators for events that are inserted into the working memory of the decision engine. You can use these operators to define the temporal reasoning behavior of the events that you declare in your Java class or DRL rule file. Temporal operators are not supported when the decision engine is running in cloud mode.

- **after**
- **before**
- **coincides**
- **during**
- **includes**
- **finishes**
- **finished by**
- **meets**
- **met by**
- **overlaps**
- **overlapped by**
- **starts**
- **started by**

after

This operator specifies if the current event occurs after the correlated event. This operator can also define an amount of time after which the current event can follow the correlated event, or a delimiting time range during which the current event can follow the correlated event.
For example, the following pattern matches if **$eventA** starts between 3 minutes and 30 seconds and 4 minutes after **$eventB** finishes. If **$eventA** starts earlier than 3 minutes and 30 seconds after **$eventB** finishes, or later than 4 minutes after **$eventB** finishes, then the pattern is not matched.

```
$eventA : EventA(this after[3m30s, 4m] $eventB)
```

You can also express this operator in the following way:

```
3m30s <= $eventA.startTimestamp - $eventB.endTimeStamp <= 4m
```

The **after** operator supports up to two parameter values:

- 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 1 millisecond and runs indefinitely with no end time.

The **after** operator also supports negative time ranges:

```
$eventA : EventA(this after[-3m30s, -2m] $eventB)
```

If the first value is greater than the second value, the decision engine automatically reverses them. For example, the following two patterns are interpreted by the decision engine in the same way:

```
$eventA : EventA(this after[-3m30s, -2m] $eventB)
$eventA : EventA(this after[-2m, -3m30s] $eventB)
```

before

This operator specifies if the current event occurs before the correlated event. This operator can also define an amount of time before which the current event can precede the correlated event, or a delimiting time range during which the current event can precede the correlated event.
For example, the following pattern matches if **$eventA** finishes between 3 minutes and 30 seconds and 4 minutes before **$eventB** starts. If **$eventA** finishes earlier than 3 minutes and 30 seconds before **$eventB** starts, or later than 4 minutes before **$eventB** starts, then the pattern is not matched.

```
$eventA : EventA(this before[3m30s, 4m] $eventB)
```

You can also express this operator in the following way:

```
3m30s <= $eventB.startTimestamp - $eventA.endTimeStamp <= 4m
```

The **before** operator supports up to two parameter values:

- 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 1 millisecond and runs indefinitely with no end time.

The **before** operator also supports negative time ranges:

```
$eventA : EventA(this before[-3m30s, -2m] $eventB)
```

If the first value is greater than the second value, the decision engine automatically reverses them. For example, the following two patterns are interpreted by the decision engine in the same way:

```
$eventA : EventA(this before[-3m30s, -2m] $eventB)
$eventA : EventA(this before[-2m, -3m30s] $eventB)
```

coincides

This operator specifies if the two events occur at the same time, with the same start and end times.
For example, the following pattern matches if both the start and end time stamps of **$eventA** and **$eventB** are identical:

```
$eventA : EventA(this coincides $eventB)
```

The **coincides** operator supports up to two parameter values for the distance between the event start and end times, if they are not identical:

- If only one parameter is given, the parameter 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 is used as a threshold for the end time.

The following pattern uses start and end time thresholds:

```
$eventA : EventA(this coincides[15s, 10s] $eventB)
```

The pattern matches if the following conditions are met:

```
abs($eventA.startTimestamp - $eventB.startTimestamp) <= 15s
&&
abs($eventA.endTimestamp - $eventB.endTimestamp) <= 10s
```

WARNING

The decision engine does not support negative intervals for the **coincides** operator. If you use negative intervals, the decision engine generates an error.

during

This operator specifies if the current event occurs within the time frame of when the correlated event starts and ends. The current event must start after the correlated event starts and must end before the correlated event ends. (With the **coincides** operator, the start and end times are the same or nearly the same.)

For example, the following pattern matches if **$eventA** starts after **$eventB** starts and ends before **$eventB** ends:

```
$eventA : EventA(this during $eventB)
```

You can also express this operator in the following way:

```
$eventB.startTimestamp < $eventA.startTimestamp <= $eventA.endTimestamp <
$eventB.endTimestamp
```

The **during** operator supports one, two, or four optional parameters:

- If one value is defined, this value is 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 are a threshold between which the current event start time and end time must occur in relation to the correlated event start and end times.
  For example, if the values are **5s** and **10s**, the current event must start between 5 and 10 seconds after the correlated event starts and must end between 5 and 10 seconds before the correlated event ends.
- If four values are defined, the first and second values are the minimum and maximum distances between the start times of the events, and the third and fourth values are the minimum and maximum distances between the end times of the two events.

includes

This operator specifies if the correlated event occurs within the time frame of when the current event occurs. The correlated event must start after the current event starts and must end before the current event ends. (The behavior of this operator is the reverse of the **during** operator behavior.)
For example, the following pattern matches if **$eventB** starts after **$eventA** starts and ends before **$eventA** ends:

```
$eventA : EventA(this includes $eventB)
```

You can also express this operator in the following way:

```
$eventA.startTimestamp < $eventB.startTimestamp <= $eventB.endTimestamp <
$eventA.endTimestamp
```

The **includes** operator supports one, two, or four optional parameters:

- If one value is defined, this value is 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 are a threshold between which the correlated event start time and end time must occur in relation to the current event start and end times.
  For example, if the values are **5s** and **10s**, the correlated event must start between 5 and 10 seconds after the current event starts and must end between 5 and 10 seconds before the current event ends.

- If four values are defined, the first and second values are the minimum and maximum distances between the start times of the events, and the third and fourth values are the minimum and maximum distances between the end times of the two events.

finishes

This operator specifies if the current event starts after the correlated event but both events end at the same time.
For example, the following pattern matches if **$eventA** starts after **$eventB** starts and ends at the same time when **$eventB** ends:

```
$eventA : EventA(this finishes $eventB)
```

You can also express this operator in the following way:

```
$eventB.startTimestamp < $eventA.startTimestamp
&&
$eventA.endTimestamp == $eventB.endTimestamp
```

The **finishes** operator supports one optional parameter that sets the maximum time allowed between the end times of the two events:

```
$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 decision engine does not support negative intervals for the **finishes** operator. If you use negative intervals, the decision engine generates an error.

finished by

This operator specifies if the correlated event starts after the current event but both events end at the same time. (The behavior of this operator is the reverse of the **finishes** operator behavior.)
For example, the following pattern matches if **$eventB** starts after **$eventA** starts and ends at the same time when **$eventA** ends:

```
$eventA : EventA(this finishedby $eventB)
```

You can also express this operator in the following way:

```
$eventA.startTimestamp < $eventB.startTimestamp
&&
$eventA.endTimestamp == $eventB.endTimestamp
```

The **finished by** operator supports one optional parameter that 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 decision engine does not support negative intervals for the **finished by** operator. If you use negative intervals, the decision engine generates an error.

meets

This operator specifies if the current event ends at the same time when the correlated event starts.
For example, the following pattern matches if **$eventA** ends at the same time when **$eventB** starts:

```
$eventA : EventA(this meets $eventB)
```

You can also express this operator in the following way:

```
abs($eventB.startTimestamp - $eventA.endTimestamp) == 0
```

The **meets** operator supports one optional parameter that sets the maximum time allowed between the end time of the current event and the start time of the correlated event:

```
$eventA : EventA(this meets[5s] $eventB)
```

This pattern matches if these conditions are met:

```
abs($eventB.startTimestamp - $eventA.endTimestamp) <= 5s
```

WARNING

The decision engine does not support negative intervals for the **meets** operator. If you use negative intervals, the decision engine generates an error.

met by

This operator specifies if the correlated event ends at the same time when the current event starts. (The behavior of this operator is the reverse of the **meets** operator behavior.)
For example, the following pattern matches if **$eventB** ends at the same time when **$eventA** starts:

```
$eventA : EventA(this metby $eventB)
```

You can also express this operator in the following way:

```
abs($eventA.startTimestamp - $eventB.endTimestamp) == 0
```

The **met by** operator supports one optional parameter that sets the maximum distance between the end time of the correlated event and the start time of the current event:

```
$eventA : EventA(this metby[5s] $eventB)
```

This pattern matches if these conditions are met:

```
abs($eventA.startTimestamp - $eventB.endTimestamp) <= 5s
```

WARNING

The decision engine does not support negative intervals for the **met by** operator. If you use negative intervals, the decision engine generates an error.

overlaps

This operator specifies if the current event starts before the correlated event starts and it ends during the time frame that the correlated event occurs. The current event must end between the start and end times of the correlated event.
For example, the following pattern matches if **$eventA** starts before **$eventB** starts and then ends while **$eventB** occurs, before **$eventB** ends:

```
$eventA : EventA(this overlaps $eventB)
```

The **overlaps** operator supports up to two parameters:

- If one parameter is defined, the value is the maximum distance between the start time of the correlated event and the end time of the current event.
- If two parameters are defined, the values are the minimum distance (first value) and the maximum distance (second value) between the start time of the correlated event and the end time of the current event.

overlapped by

This operator specifies if the correlated event starts before the current event starts and it ends during the time frame that the current event occurs. The correlated event must end between the start and end times of the current event. (The behavior of this operator is the reverse of the **overlaps** operator behavior.)
For example, the following pattern matches if **$eventB** starts before **$eventA** starts and then ends while **$eventA** occurs, before **$eventA** ends:

```
$eventA : EventA(this overlappedby $eventB)
```

The **overlapped by** operator supports up to two parameters:

- If one parameter is defined, the value is the maximum distance between the start time of the current event and the end time of the correlated event.
- If two parameters are defined, the values are the minimum distance (first value) and the maximum distance (second value) between the start time of the current event and the end time of the correlated event.

starts

This operator specifies if the two events start at the same time but the current event ends before the correlated event ends.
For example, the following pattern matches if **$eventA** and **$eventB** start at the same time, and **$eventA** ends before **$eventB** ends:

```
$eventA : EventA(this starts $eventB)
```

You can also express this operator in the following way:

```
$eventA.startTimestamp == $eventB.startTimestamp
&&
$eventA.endTimestamp < $eventB.endTimestamp
```

The **starts** operator supports one optional parameter that sets the maximum distance between the start times of the two events:

```
$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 decision engine does not support negative intervals for the **starts** operator. If you use negative intervals, the decision engine generates an error.

started by

This operator specifies if the two events start at the same time but the correlated event ends before the current event ends. (The behavior of this operator is the reverse of the **starts** operator behavior.)
For example, the following pattern matches if **$eventA** and **$eventB** start at the same time, and **$eventB** ends before **$eventA** ends:

```
$eventA : EventA(this startedby $eventB)
```

You can also express this operator in the following way:

```
$eventA.startTimestamp == $eventB.startTimestamp
&&
$eventA.endTimestamp > $eventB.endTimestamp
```

The **started by** operator supports one optional parameter that sets the maximum distance between the start times of the two events:

```
$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 decision engine does not support negative intervals for the **started by** operator. If you use negative intervals, the decision engine generates an error.

6.7. SESSION CLOCK IMPLEMENTATIONS IN THE DECISION ENGINE

During complex event processing, events in the decision engine may have temporal constraints and therefore require a session clock that provides the current time. For example, if a rule needs to

determine the average price of a given stock over the last 60 minutes, the decision engine must be able to compare the stock price event time stamp with the current time in the session clock.

The decision engine supports a real-time clock and a pseudo clock. You can use one or both clock types depending on the scenario:

- **Rules testing:** Testing requires a controlled environment, and when the tests include rules with temporal constraints, you must be able to control the input rules and facts and the flow of time.
- **Regular execution:** The decision engine reacts to events in real time and therefore requires a real-time clock.
- **Special environments:** Specific environments may have specific time control requirements. For example, clustered environments may require clock synchronization or Java Enterprise Edition (JEE) environments may require a clock provided by the application server.
- **Rules replay or simulation:** In order to replay or simulate scenarios, the application must be able to control the flow of time.

Consider your environment requirements as you decide whether to use a real-time clock or pseudo clock in the decision engine.

Real-time clock

The real-time clock is the default clock implementation in the decision engine and uses the system clock to determine the current time for time stamps. To configure the decision engine to use the real-time clock, set the KIE session configuration parameter to **realtime**:

Configure real-time clock in KIE session

```
import org.kie.api.KieServices.Factory;
import org.kie.api.runtime.conf.ClockTypeOption;
import org.kie.api.runtime.KieSessionConfiguration;

KieSessionConfiguration config = KieServices.Factory.get().newKieSessionConfiguration();

config.setOption(ClockTypeOption.get("realtime"));
```

Pseudo clock

The pseudo clock implementation in the decision engine is helpful for testing temporal rules and it can be controlled by the application. To configure the decision engine to use the pseudo clock, set the KIE session configuration parameter to **pseudo**:

Configure pseduo clock in KIE session

```
import org.kie.api.runtime.conf.ClockTypeOption;
import org.kie.api.runtime.KieSessionConfiguration;
import org.kie.api.KieServices.Factory;

KieSessionConfiguration config = KieServices.Factory.get().newKieSessionConfiguration();

config.setOption(ClockTypeOption.get("pseudo"));
```

You can also use additional configurations and fact handlers to control the pseudo clock:

Control pseudo clock behavior in KIE session

```
import java.util.concurrent.TimeUnit;

import org.kie.api.runtime.KieSessionConfiguration;
import org.kie.api.KieServices.Factory;
import org.kie.api.runtime.KieSession;
import org.drools.core.time.SessionPseudoClock;
import org.kie.api.runtime.rule.FactHandle;
import org.kie.api.runtime.conf.ClockTypeOption;

KieSessionConfiguration conf = KieServices.Factory.get().newKieSessionConfiguration();

conf.setOption( ClockTypeOption.get("pseudo"));
KieSession session = kbase.newKieSession(conf, null);

SessionPseudoClock clock = session.getSessionClock();

// 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);
```

6.8. EVENT STREAMS AND ENTRY POINTS

The decision engine can process high volumes of events in the form of event streams. In DRL rule declarations, a stream is also known as an *entry point*. When you declare an entry point in a DRL rule or Java application, the decision engine, at compile time, identifies and creates the proper internal structures to use data from only that entry point to evaluate that rule.

Facts from one entry point, or stream, can join facts from any other entry point in addition to facts already in the working memory of the decision engine. Facts always remain associated with the entry point through which they entered the decision engine. Facts of the same type can enter the decision engine through several entry points, but facts that enter the decision engine through entry point A can never match a pattern from entry point B.

Event streams have the following characteristics:

- Events in the stream are ordered by time stamp. The time stamps may have different semantics for different streams, but they are always ordered internally.
- Event streams usually have a high volume of events.
- Atomic events in streams are usually not useful individually, only collectively in a stream.
- Event streams can be homogeneous and contain a single type of event, or heterogeneous and contain events of different types.

6.8.1. Declaring entry points for rule data

You can declare an entry point (event stream) for events so that the decision engine uses data from only that entry point to evaluate the rules. You can declare an entry point either implicitly by referencing it in DRL rules or explicitly in your Java application.

Procedure

Use one of the following methods to declare the entry point:

- In the DRL rule file, specify **from entry-point "<name>"** for the inserted fact:

  Authorize withdrawal rule with "ATM Stream" entry point

  ```
  rule "Authorize withdrawal"
  when
    WithdrawRequest($ai : accountId, $am : amount) from entry-point "ATM Stream"
    CheckingAccount(accountId == $ai, balance > $am)
  then
    // Authorize withdrawal.
  end
  ```

  Apply fee rule with "Branch Stream" entry point

  ```
  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
  ```

  Both example DRL rules from a banking application insert the event **WithdrawalRequest** with the fact **CheckingAccount**, but from different entry points. At run time, the decision engine evaluates the **Authorize withdrawal** rule using data from only the **"ATM Stream"** entry point, and evaluates the **Apply fee** rule using data from only the **"Branch Stream"** entry point. Any events inserted into the **"ATM Stream"** can never match patterns for the **"Apply fee"** rule, and any events inserted into the **"Branch Stream"** can never match patterns for the **"Authorize withdrawal rule"**.

- In the Java application code, use the **getEntryPoint()** method to specify and obtain an **EntryPoint** object and insert facts into that entry point accordingly:

  Java application code with EntryPoint object and inserted facts

  ```
  import org.kie.api.runtime.KieSession;
  import org.kie.api.runtime.rule.EntryPoint;

  // Create your KIE base and KIE session as usual.
  KieSession session = ...

  // Create a reference to the entry point.
  EntryPoint atmStream = session.getEntryPoint("ATM Stream");

  // Start inserting your facts into the entry point.
  atmStream.insert(aWithdrawRequest);
  ```

Any DRL rules that specify **from entry-point "ATM Stream"** are then evaluated based on the data in this entry point only.

6.9. SLIDING WINDOWS OF TIME OR LENGTH

In stream mode, the decision engine can process events from a specified sliding window of time or length. A sliding time window is a specified period of time during which events can be processed. A sliding length window is a specified number of events that can be processed. When you declare a sliding window in a DRL rule or Java application, the decision engine, at compile time, identifies and creates the proper internal structures to use data from only that sliding window to evaluate that rule.

For example, the following DRL rule snippets instruct the decision engine to process only the stock points from the last 2 minutes (sliding time window) or to process only the last 10 stock points (sliding length window):

Process stock points from the last 2 minutes (sliding time window)

```
StockPoint() over window:time(2m)
```

Process the last 10 stock points (sliding length window)

```
StockPoint() over window:length(10)
```

6.9.1. Declaring sliding windows for rule data

You can declare a sliding window of time (flow of time) or length (number of occurrences) for events so that the decision engine uses data from only that window to evaluate the rules.

Procedure

In the DRL rule file, specify **over window:<time_or_length>(<value>)** for the inserted fact.

For example, the following two DRL rules activate a fire alarm based on an average temperature. However, the first rule uses a sliding time window to calculate the average over the last 10 minutes while the second rule uses a sliding length window to calculate the average over the last one hundred temperature readings.

Average temperature over sliding time window

```
rule "Sound the alarm if 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
```

Average temperature over sliding length window

```
rule "Sound the alarm if 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
```

The decision engine discards any **SensorReading** events that are more than 10 minutes old or that are not part of the last one hundred readings, and continues recalculating the average as the minutes or readings "slide" forward in real time.

The decision engine does not automatically remove outdated events from the KIE session because other rules without sliding window declarations might depend on those events. The decision engine stores events in the KIE session until the events expire either by explicit rule declarations or by implicit reasoning within the decision engine based on inferred data in the KIE base.

6.10. MEMORY MANAGEMENT FOR EVENTS

In stream mode, the decision engine uses automatic memory management to maintain events that are stored in KIE sessions. The decision engine can retract from a KIE session any events that no longer match any rule due to their temporal constraints and release any resources held by the retracted events.

The decision engine uses either explicit or inferred expiration to retract outdated events:

- **Explicit expiration:** The decision engine removes events that are explicitly set to expire in rules that declare the **@expires** tag:

  DRL rule snippet with explicit expiration

  ```
  declare StockPoint
    @expires( 30m )
  end
  ```

  This example rule sets any **StockPoint** events to expire after 30 minutes and to be removed from the KIE session if no other rules use the events.

- **Inferred expiration:** The decision engine can calculate the expiration offset for a given event implicitly by analyzing the temporal constraints in the rules:

  DRL rule with temporal constraints

  ```
  rule "Correlate orders"
  when
    $bo : BuyOrder($id : id)
    $ae : AckOrder(id == $id, this after[0,10s] $bo)
  then
    // Perform an action.
  end
  ```

  For this example rule, the decision engine automatically calculates that whenever a **BuyOrder** event occurs, the decision engine needs to store the event for up to 10 seconds and wait for the matching **AckOrder** event. After 10 seconds, the decision engine infers the expiration and

removes the event from the KIE session. An **AckOrder** event can only match an existing **BuyOrder** event, so the decision engine infers the expiration if no match occurs and removes the event immediately.

The decision engine analyzes the entire KIE base to find the offset for every event type and to ensure that no other rules use the events that are pending removal. Whenever an implicit expiration clashes with an explicit expiration value, the decision engine uses the greater time frame of the two to store the event longer.

CHAPTER 7. DECISION ENGINE QUERIES AND LIVE QUERIES

You can use queries with the decision engine to retrieve fact sets based on fact patterns as they are used in rules. The patterns might also use optional parameters.

To use queries with the decision engine, you add the query definitions in DRL files and then obtain the matching results in your application code. While a query iterates over a result collection, you can use any identifier that is bound to the query to access the corresponding fact or fact field by calling the **get()** method with the binding variable name as the argument. If the binding refers to a fact object, you can retrieve the fact handle by calling **getFactHandle()** with the variable name as the parameter.

Example query definition in a DRL file

```
query "people under the age of 21"
    $person : Person( age < 21 )
end
```

Example application code to obtain and iterate over query results

```
QueryResults results = ksession.getQueryResults( "people under the age of 21" );
System.out.println( "we have " + results.size() + " people under the age of 21" );

System.out.println( "These people are under the age of 21:" );

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

Invoking queries and processing the results by iterating over the returned set can be difficult when you are monitoring changes over time. To alleviate this difficulty with ongoing queries, Red Hat Decision Manager provides *live queries*, which use an attached listener for change events instead of returning an iterable result set. Live queries remain open by creating a view and publishing change events for the contents of this view.

To activate a live query, start your query with parameters and monitor changes in the resulting view. You can use the **dispose()** method to terminate the query and discontinue this reactive scenario.

Example query definition in a DRL file

```
query colors(String $color1, String $color2)
    TShirt(mainColor = $color1, secondColor = $color2, $price: manufactureCost)
end
```

Example application code with an event listener and a live query

```
final List updated = new ArrayList();
final List removed = new ArrayList();
final List added = new ArrayList();

ViewChangedEventListener listener = new ViewChangedEventListener() {
 public void rowUpdated(Row row) {
  updated.add( row.get( "$price" ) );
 }
```

```
 public void rowRemoved(Row row) {
  removed.add( row.get( "$price" ) );
 }

 public void rowAdded(Row row) {
  added.add( row.get( "$price" ) );
 }
};

// Open the live query:
LiveQuery query = ksession.openLiveQuery( "colors",
                                  new Object[] { "red", "blue" },
                                  listener );
...
...

// Terminate the live query:
query.dispose()
```

CHAPTER 8. DECISION ENGINE EVENT LISTENERS AND DEBUG LOGGING

In Red Hat Decision Manager, you can add or remove listeners for decision engine events, such as fact insertions and rule executions. With decision engine event listeners, you can be notified of decision engine activity and separate your logging and auditing work from the core of your application.

The decision engine supports the following default event listeners for the agenda and working memory:

- **AgendaEventListener**
- **WorkingMemoryEventListener**

For each event listener, the decision engine also supports the following specific events that you can specify to be monitored:

- **MatchCreatedEvent**
- **MatchCancelledEvent**
- **BeforeMatchFiredEvent**
- **AfterMatchFiredEvent**
- **AgendaGroupPushedEvent**
- **AgendaGroupPoppedEvent**
- **ObjectInsertEvent**
- **ObjectDeletedEvent**
- **ObjectUpdatedEvent**
- **ProcessCompletedEvent**
- **ProcessNodeLeftEvent**
- **ProcessNodeTriggeredEvent**
- **ProcessStartEvent**

For example, the following code uses a **DefaultAgendaEventListener** listener attached to a KIE session and specifies the **AfterMatchFiredEvent** event to be monitored. The code prints pattern matches after the rules are executed (fired):

Example code to monitor and print AfterMatchFiredEvent events in the agenda

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

The decision engine also supports the following agenda and working memory event listeners for debug logging:

- **DebugAgendaEventListener**
- **DebugRuleRuntimeEventListener**

These event listeners implement the same supported event-listener methods and include a debug print statement by default. You can add a specific supported event to be monitored and documented, or monitor all agenda or working memory activity.

For example, the following code uses the **DebugRuleRuntimeEventListener** event listener to monitor and print all working memory events:

Example code to monitor and print all working memory events

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

8.1. CONFIGURING A LOGGING UTILITY IN THE DECISION ENGINE

The decision engine uses the Java logging API SLF4J for system logging. You can use one of the following logging utilities with the decision engine to investigate decision engine activity, such as for troubleshooting or data gathering:

- Logback
- Apache Commons Logging
- Apache Log4j
- **java.util.logging** package

Procedure

For the logging utility that you want to use, add the relevant dependency to your Maven project or save the relevant XML configuration file in the **org.drools** package of your Red Hat Decision Manager distribution:

Example Maven dependency for Logback

```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>${logback.version}</version>
</dependency>
```

Example logback.xml configuration file in org.drools package

```
<configuration>
  <logger name="org.drools" level="debug"/>
  ...
<configuration>
```

Example log4j.xml configuration file in org.drools package

```
<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/">
  <category name="org.drools">
    <priority value="debug" />
  </category>
  ...
</log4j:configuration>
```

NOTE

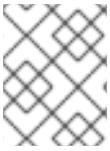
If you are developing for an ultra light environment, use the **slf4j-nop** or **slf4j-simple** logger.

CHAPTER 9. EXAMPLE DECISIONS IN RED HAT DECISION MANAGER FOR AN IDE

Red Hat Decision Manager provides example decisions distributed as Java classes that you can import into your integrated development environment (IDE). You can use these examples to better understand decision engine capabilities or use them as a reference for the decisions that you define in your own Red Hat Decision Manager projects.

The following example decision sets are some of the examples available in Red Hat Decision Manager:

- **Hello World example**: Demonstrates basic rule execution and use of debug output
- **State example**: Demonstrates forward chaining and conflict resolution through rule salience and agenda groups
- **Fibonacci example**: Demonstrates recursion and conflict resolution through rule salience
- **Banking example**: Demonstrates pattern matching, basic sorting, and calculation
- **Pet Store example**: Demonstrates rule agenda groups, global variables, callbacks, and GUI integration
- **Sudoku example**: Demonstrates complex pattern matching, problem solving, callbacks, and GUI integration
- **House of Doom example**: Demonstrates backward chaining and recursion



NOTE

For optimization examples provided with Red Hat Business Optimizer, see *Getting started with Red Hat Business Optimizer*.

9.1. IMPORTING AND EXECUTING RED HAT DECISION MANAGER EXAMPLE DECISIONS IN AN IDE

You can import Red Hat Decision Manager example decisions into your integrated development environment (IDE) and execute them to explore how the rules and code function. You can use these examples to better understand decision engine capabilities or use them as a reference for the decisions that you define in your own Red Hat Decision Manager projects.

Prerequisites

- Java 8 or later is installed.
- Maven 3.5.x or later is installed.
- An IDE is installed, such as Red Hat CodeReady Studio.

Procedure

1. Download and unzip the **Red Hat Decision Manager 7.5.1 Source Distribution** from the Red Hat Customer Portal to a temporary directory, such as **/rhdm-7.5.1-sources**.

2. Open your IDE and select **File → Import → Maven → Existing Maven Projects**, or the equivalent option for importing a Maven project.

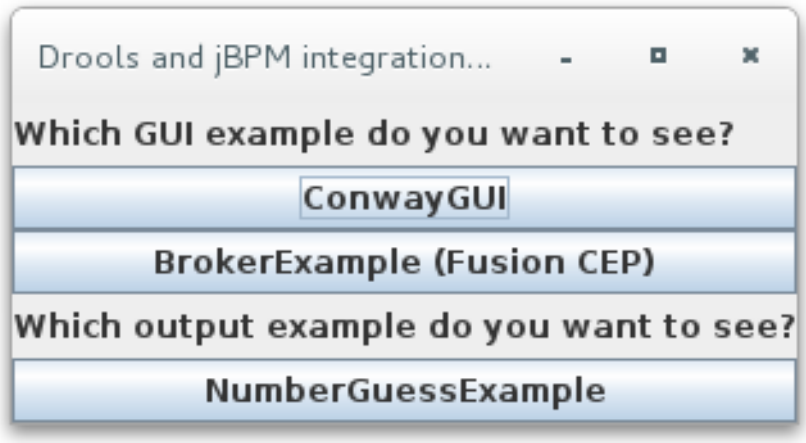
3. Click **Browse**, navigate to **~/rhdm-7.5.1-sources/src/drools-$VERSION/drools-examples** (or, for the Conway's Game of Life example, **~/rhdm-7.5.1-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples**), and import the project.

4. Navigate to the example package that you want to run and find the Java class with the **main** method.

5. Right-click the Java class and select **Run As → Java Application** to run the example.
   To run all examples through a basic user interface, run the **DroolsExamplesApp.java** class (or, for Conway's Game of Life, the **DroolsJbpmIntegrationExamplesApp.java** class) in the **org.drools.examples** main class.

Figure 9.1. Interface for all examples in drools-examples (DroolsExamplesApp.java)

Figure 9.2. Interface for all examples in droolsjbpm-integration-examples (DroolsJbpmIntegrationExamplesApp.java)



9.2. HELLO WORLD EXAMPLE DECISIONS (BASIC RULES AND DEBUGGING)

The Hello World example decision set demonstrates how to insert objects into the decision engine working memory, how to match the objects using rules, and how to configure logging to trace the internal activity of the decision engine.

The following is an overview of the Hello World example:

- Name: **helloworld**
- Main class: **org.drools.examples.helloworld.HelloWorldExample** (in **src/main/java**)
- Module: **drools-examples**
- **Type**: Java application
- Rule file: **org.drools.examples.helloworld.HelloWorld.drl** (in **src/main/resources**)
- **Objective**: Demonstrates basic rule execution and use of debug output

In the Hello World example, a KIE session is generated to enable rule execution. All rules require a KIE session for execution.

KIE session for rule execution

```
KieServices ks = KieServices.Factory.get(); (1)
KieContainer kc = ks.getKieClasspathContainer(); (2)
KieSession ksession = kc.newKieSession("HelloWorldKS"); (3)
```

(1) Obtains the **KieServices** factory. This is the main interface that applications use to interact with the decision engine.

(2) Creates a **KieContainer** from the project class path. This detects a **/META-INF/kmodule.xml** file from which it configures and instantiates a **KieContainer** with a **KieModule**.

(3) Creates a **KieSession** based on the **"HelloWorldKS"** KIE session configuration defined in the **/META-INF/kmodule.xml** file.

NOTE

For more information about Red Hat Decision Manager project packaging, see *Packaging and deploying a Red Hat Decision Manager project*.

Red Hat Decision Manager has an event model that exposes internal engine activity. Two default debug listeners, **DebugAgendaEventListener** and **DebugRuleRuntimeEventListener**, print debug event information to the **System.err** output. The **KieRuntimeLogger** provides execution auditing, the result of which you can view in a graphical viewer.

Debug listeners and audit loggers

```
// Set up listeners.
ksession.addEventListener( new DebugAgendaEventListener() );
ksession.addEventListener( new DebugRuleRuntimeEventListener() );

// Set up a file-based audit logger.
KieRuntimeLogger logger = KieServices.get().getLoggers().newFileLogger( ksession,
"./target/helloworld" );

// Set up a ThreadedFileLogger so that the audit view reflects events while debugging.
KieRuntimeLogger logger = ks.getLoggers().newThreadedFileLogger( ksession, "./target/helloworld",
1000 );
```

The logger is a specialized implementation built on the **Agenda** and **RuleRuntime** listeners. When the decision engine has finished executing, **logger.close()** is called.

The example creates a single **Message** object with the message **"Hello World"**, inserts the status **HELLO** into the **KieSession**, executes rules with **fireAllRules()**.

Data insertion and execution

```
// Insert facts into the KIE session.
final Message message = new Message();
message.setMessage( "Hello World" );
message.setStatus( Message.HELLO );
ksession.insert( message );

// Fire the rules.
ksession.fireAllRules();
```

Rule execution uses a data model to pass data as inputs and outputs to the **KieSession**. The data model in this example has two fields: the **message**, which is a **String**, and the **status**, which can be **HELLO** or **GOODBYE**.

Data model class

```
public static class Message {
    public static final int HELLO   = 0;
    public static final int GOODBYE = 1;

    private String          message;
```

```
    private int           status;
    ...
}
```

The two rules are located in the file **src/main/resources/org/drools/examples/helloworld/HelloWorld.drl**.

The **when** condition of the **"Hello World"** rule states that the rule is activated for each **Message** object inserted into the KIE session that has the status **Message.HELLO**. Additionally, two variable bindings are created: the variable **message** is bound to the **message** attribute and the variable **m** is bound to the matched **Message** object itself.

The **then** action of the rule specifies to print the content of the bound variable **message** to **System.out**, and then changes the values of the **message** and **status** attributes of the **Message** object bound to **m**. The rule uses the **modify** statement to apply a block of assignments in one statement and to notify the decision engine of the changes at the end of the block.

"Hello World" rule

```
rule "Hello World"
  when
    m : Message( status == Message.HELLO, message : message )
  then
    System.out.println( message );
    modify ( m ) { message = "Goodbye cruel world",
                   status = Message.GOODBYE };
end
```

The **"Good Bye"** rule is similar to the **"Hello World"** rule except that it matches **Message** objects that have the status **Message.GOODBYE**.

"Good Bye" rule

```
rule "Good Bye"
  when
    Message( status == Message.GOODBYE, message : message )
  then
    System.out.println( message );
end
```

To execute the example, run the **org.drools.examples.helloworld.HelloWorldExample** class as a Java application in your IDE. The rule writes to **System.out**, the debug listener writes to **System.err**, and the audit logger creates a log file in **target/helloworld.log**.

System.out output in the IDE console

```
Hello World
Goodbye cruel world
```

System.err output in the IDE console

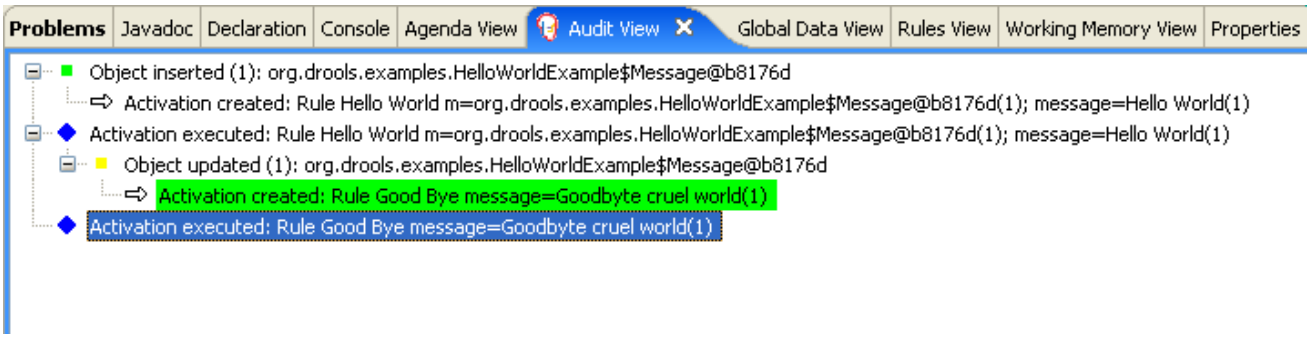
```
==>[ActivationCreated(0): rule=Hello World;
                  tuple=[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
[ObjectInserted: handle=
```

```
[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96];
            object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
[BeforeActivationFired: rule=Hello World;
             tuple=[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
==>[ActivationCreated(4): rule=Good Bye;
             tuple=[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
[ObjectUpdated: handle=
[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96];
          old_object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96;
          new_object=org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]
[AfterActivationFired(0): rule=Hello World]
[BeforeActivationFired: rule=Good Bye;
             tuple=[fid:1:2:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
[AfterActivationFired(4): rule=Good Bye]
```

To better understand the execution flow of this example, you can load the audit log file from **target/helloworld.log** into your IDE debug view or **Audit View**, if available (for example, in **Window → Show View** in some IDEs).

In this example, the **Audit view** shows that the object is inserted, which creates an activation for the **"Hello World"** rule. The activation is then executed, which updates the **Message** object and causes the **"Good Bye"** rule to activate. Finally, the **"Good Bye"** rule is executed. When you select an event in the **Audit View**, the origin event, which is the **"Activation created"** event in this example, is highlighted in green.

Figure 9.3. Hello World example Audit View



9.3. STATE EXAMPLE DECISIONS (FORWARD CHAINING AND CONFLICT RESOLUTION)

The State example decision set demonstrates how the decision engine uses forward chaining and any changes to facts in the working memory to resolve execution conflicts for rules in a sequence. The example focuses on resolving conflicts through salience values or through agenda groups that you can define in rules.

The following is an overview of the State example:

- Name: **state**
- Main classes: **org.drools.examples.state.StateExampleUsingSalience**, **org.drools.examples.state.StateExampleUsingAgendaGroup** (in **src/main/java**)
- Module: **drools-examples**
- **Type**: Java application

- Rule files: **org.drools.examples.state.*.drl** (in **src/main/resources**)
- **Objective**: Demonstrates forward chaining and conflict resolution through rule salience and agenda groups

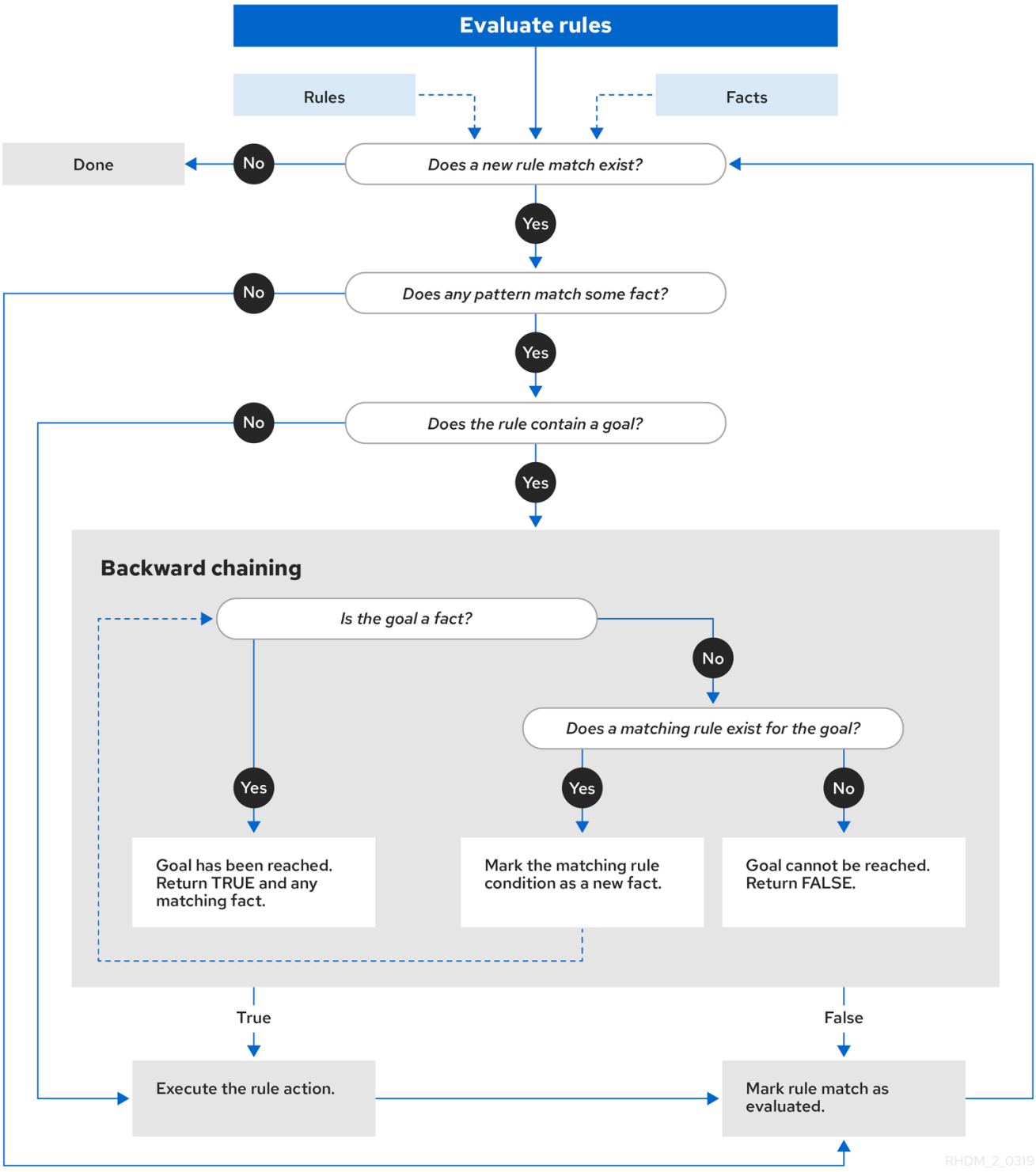
A forward-chaining rule system is a data-driven system that starts with a fact in the working memory of the decision engine and reacts to changes to that fact. When objects are inserted into working memory, any rule conditions that become true as a result of the change are scheduled for execution by the agenda.

In contrast, a backward-chaining rule system is a goal-driven system that starts with a conclusion that the decision engine attempts to satisfy, often using recursion. If the system cannot reach the conclusion or goal, it searches for subgoals, which are conclusions that complete part of the current goal. The system continues this process until either the initial conclusion is satisfied or all subgoals are satisfied.

The decision engine in Red Hat Decision Manager uses both forward and backward chaining to evaluate rules.

The following diagram illustrates how the decision engine evaluates rules using forward chaining overall with a backward-chaining segment in the logic flow:

Figure 9.4. Rule evaluation logic using forward and backward chaining



In the State example, each **State** class has fields for its name and its current state (see the class **org.drools.examples.state.State**). The following states are the two possible states for each object:

- **NOTRUN**
- **FINISHED**

State class

```
public class State {
    public static final int NOTRUN   = 0;
```

```
    public static final int FINISHED = 1;

    private final PropertyChangeSupport changes =
        new PropertyChangeSupport( this );

    private String name;
    private int    state;

    ... setters and getters go here...
}
```

The State example contains two versions of the same example to resolve rule execution conflicts:

- A **StateExampleUsingSalience** version that resolves conflicts by using rule salience
- A **StateExampleUsingAgendaGroups** version that resolves conflicts by using rule agenda groups

Both versions of the state example involve four **State** objects: **A**, **B**, **C**, and **D**. Initially, their states are set to **NOTRUN**, which is the default value for the constructor that the example uses.

State example using salience

The **StateExampleUsingSalience** version of the State example uses salience values in rules to resolve rule execution conflicts. Rules with a higher salience value are given higher priority when ordered in the activation queue.

The example inserts each **State** instance into the KIE session and then calls **fireAllRules()**.

Salience State example execution

```
final State a = new State( "A" );
final State b = new State( "B" );
final State c = new State( "C" );
final State d = new State( "D" );

ksession.insert( a );
ksession.insert( b );
ksession.insert( c );
ksession.insert( d );

ksession.fireAllRules();

// Dispose KIE session if stateful (not required if stateless).
ksession.dispose();
```

To execute the example, run the **org.drools.examples.state.StateExampleUsingSalience** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

Salience State example output in the IDE console

```
A finished
B finished
C finished
```

```
D finished
```

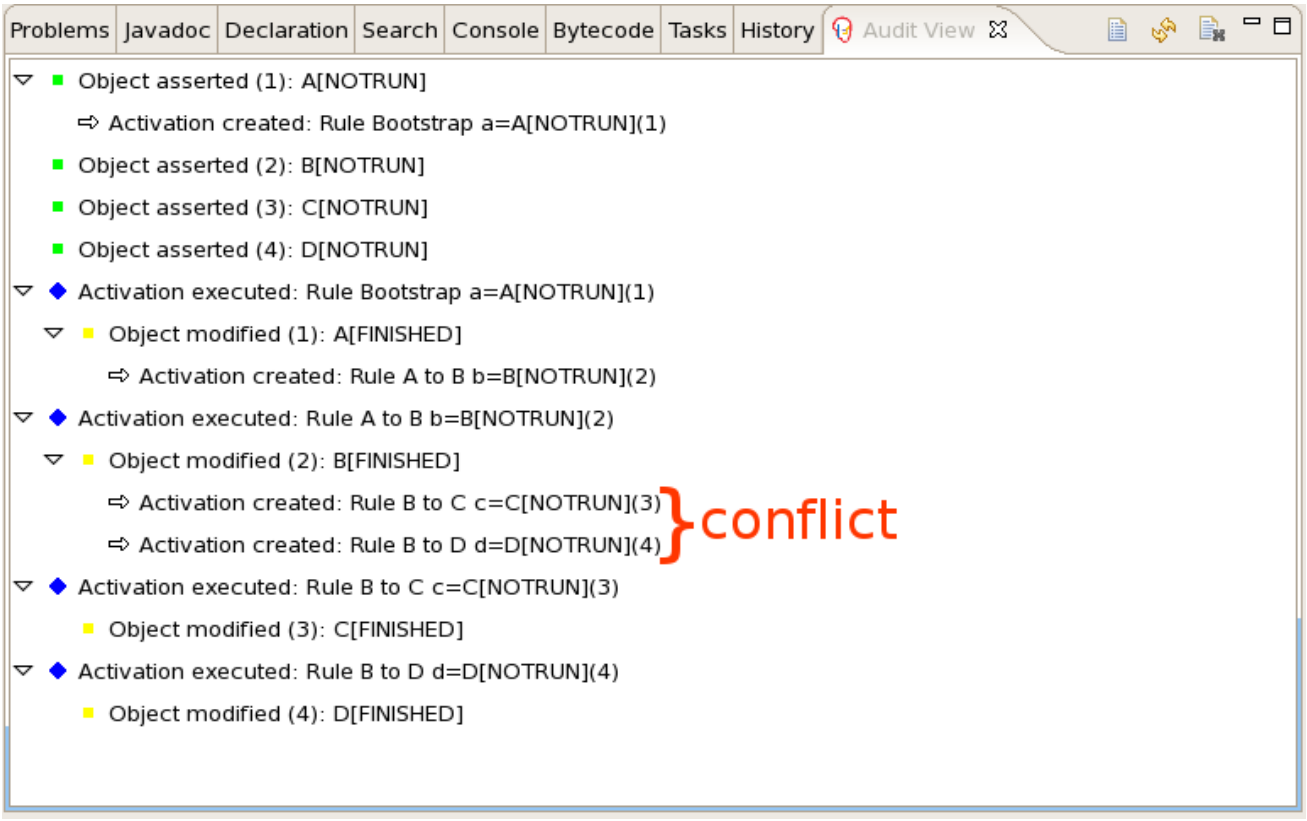
Four rules are present.

First, the **"Bootstrap"** rule fires, setting **A** to state **FINISHED**, which then causes **B** to change its state to **FINISHED**. Objects **C** and **D** are both dependent on **B**, causing a conflict that is resolved by the salience values.

To better understand the execution flow of this example, you can load the audit log file from **target/state.log** into your IDE debug view or **Audit View**, if available (for example, in **Window → Show View** in some IDEs).

In this example, the **Audit View** shows that the assertion of the object **A** in the state **NOTRUN** activates the **"Bootstrap"** rule, while the assertions of the other objects have no immediate effect.

Figure 9.5. Salience State example Audit View



Rule "Bootstrap" in salience State example

```
rule "Bootstrap"
  when
    a : State(name == "A", state == State.NOTRUN )
  then
    System.out.println(a.getName() + " finished" );
    a.setState( State.FINISHED );
end
```

The execution of the **"Bootstrap"** rule changes the state of **A** to **FINISHED**, which activates rule **"A to B"**.

Rule "A to B" in salience State example

```
rule "A to B"
  when
    State(name == "A", state == State.FINISHED )
    b : State(name == "B", state == State.NOTRUN )
  then
    System.out.println(b.getName() + " finished" );
    b.setState( State.FINISHED );
end
```

The execution of rule **"A to B"** changes the state of **B** to **FINISHED**, which activates both rules **"B to C"** and **"B to D"**, placing their activations onto the decision engine agenda.

Rules "B to C" and "B to D" in salience State example

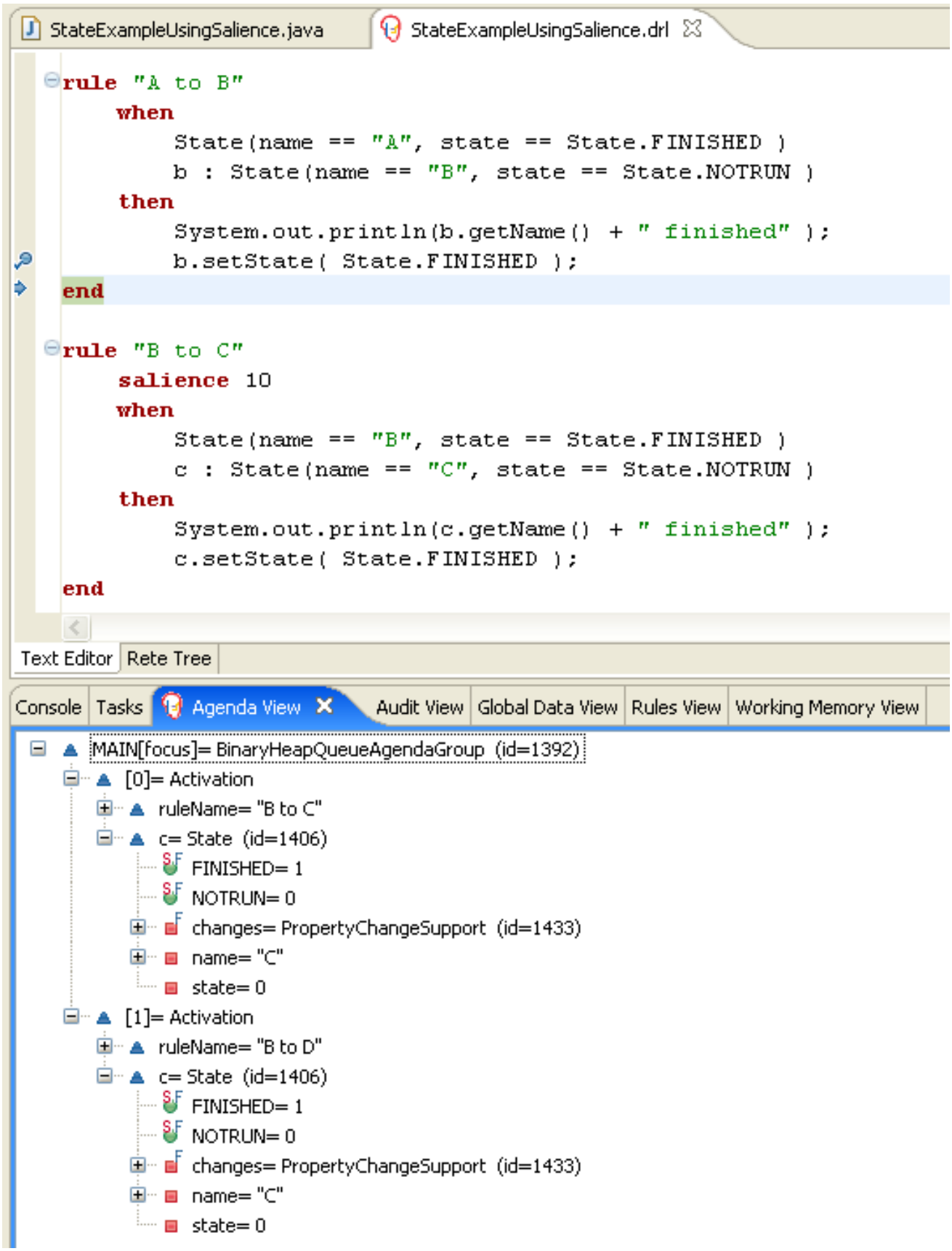
```
rule "B to C"
    salience 10
  when
    State(name == "B", state == State.FINISHED )
    c : State(name == "C", state == State.NOTRUN )
  then
    System.out.println(c.getName() + " finished" );
    c.setState( State.FINISHED );
end

rule "B to D"
  when
    State(name == "B", state == State.FINISHED )
    d : State(name == "D", state == State.NOTRUN )
  then
    System.out.println(d.getName() + " finished" );
    d.setState( State.FINISHED );
end
```

From this point on, both rules may fire and, therefore, the rules are in conflict. The conflict resolution strategy enables the decision engine agenda to decide which rule to fire. Rule **"B to C"** has the higher salience value (**10** versus the default salience value of **0**), so it fires first, modifying object **C** to state **FINISHED**.

The **Audit View** in your IDE shows the modification of the **State** object in the rule **"A to B"**, which results in two activations being in conflict.

You can also use the **Agenda View** in your IDE to investigate the state of the decision engine agenda. In this example, the **Agenda View** shows the breakpoint in the rule **"A to B"** and the state of the agenda with the two conflicting rules. Rule **"B to D"** fires last, modifying object **D** to state **FINISHED**.

Figure 9.6. Salience State example Agenda View



State example using agenda groups

The **StateExampleUsingAgendaGroups** version of the State example uses agenda groups in rules to resolve rule execution conflicts. Agenda groups enable you to partition the decision engine agenda to provide more execution control over groups of rules. By default, all rules are in the agenda group **MAIN**. You can use the **agenda-group** attribute to specify a different agenda group for the rule.

Initially, a working memory has its focus on the agenda group **MAIN**. Rules in an agenda group only fire when the group receives the focus. You can set the focus either by using the method **setFocus()** or the rule attribute **auto-focus**. The **auto-focus** attribute enables the rule to be given a focus automatically for its agenda group when the rule is matched and activated.

In this example, the **auto-focus** attribute enables rule **"B to C"** to fire before **"B to D"**.

Rule "B to C" in agenda group State example

```
rule "B to C"
    agenda-group "B to C"
    auto-focus true
  when
    State(name == "B", state == State.FINISHED )
    c : State(name == "C", state == State.NOTRUN )
  then
    System.out.println(c.getName() + " finished" );
    c.setState( State.FINISHED );
    kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "B to D" ).setFocus();
end
```

The rule **"B to C"** calls **setFocus()** on the agenda group **"B to D"**, enabling its active rules to fire, which then enables the rule **"B to D"** to fire.

Rule "B to D" in agenda group State example

```
rule "B to D"
    agenda-group "B to D"
  when
    State(name == "B", state == State.FINISHED )
    d : State(name == "D", state == State.NOTRUN )
  then
    System.out.println(d.getName() + " finished" );
    d.setState( State.FINISHED );
end
```

To execute the example, run the **org.drools.examples.state.StateExampleUsingAgendaGroups** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window (same as the salience version of the State example):

Agenda group State example output in the IDE console

```
A finished
B finished
C finished
D finished
```

Dynamic facts in the State example

Another notable concept in this State example is the use of *dynamic facts*, based on objects that implement a **PropertyChangeListener** object. In order for the decision engine to see and react to changes of fact properties, the application must notify the decision engine that changes occurred. You

can configure this communication explicitly in the rules by using the **modify** statement, or implicitly by specifying that the facts implement the **PropertyChangeSupport** interface as defined by the JavaBeans specification.

This example demonstrates how to use the **PropertyChangeSupport** interface to avoid the need for explicit **modify** statements in the rules. To make use of this interface, ensure that your facts implement **PropertyChangeSupport** in the same way that the class **org.drools.example.State** implements it, and then use the following code in the DRL rule file to configure the decision engine to listen for property changes on those facts:

Declaring a dynamic fact

```
declare type State
  @propertyChangeSupport
end
```

When you use **PropertyChangeListener** objects, each setter must implement additional code for the notification. For example, the following setter for **state** is in the class **org.drools.examples**:

Setter example with PropertyChangeSupport

```
public void setState(final int newState) {
    int oldState = this.state;
    this.state = newState;
    this.changes.firePropertyChange( "state",
                                     oldState,
                                     newState );
}
```

9.4. FIBONACCI EXAMPLE DECISIONS (RECURSION AND CONFLICT RESOLUTION)

The Fibonacci example decision set demonstrates how the decision engine uses recursion to resolve execution conflicts for rules in a sequence. The example focuses on resolving conflicts through salience values that you can define in rules.

The following is an overview of the Fibonacci example:

- **Name**: **fibonacci**
- **Main class**: **org.drools.examples.fibonacci.FibonacciExample** (in **src/main/java**)
- **Module**: **drools-examples**
- **Type**: Java application
- **Rule file**: **org.drools.examples.fibonacci.Fibonacci.drl** (in **src/main/resources**)
- **Objective**: Demonstrates recursion and conflict resolution through rule salience

The Fibonacci Numbers form a sequence starting with 0 and 1. The next Fibonacci number is obtained by adding the two preceding Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, and so on.

The Fibonacci example uses the single fact class **Fibonacci** with the following two fields:

- **sequence**
- **value**

The **sequence** field indicates the position of the object in the Fibonacci number sequence. The **value** field shows the value of that Fibonacci object for that sequence position, where **-1** indicates a value that still needs to be computed.

Fibonacci class

```
public static class Fibonacci {
    private int  sequence;
    private long value;

    public Fibonacci( final int sequence ) {
        this.sequence = sequence;
        this.value = -1;
    }

    ... setters and getters go here...
}
```

To execute the example, run the **org.drools.examples.fibonacci.FibonacciExample** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

Fibonacci example output in the IDE console

```
recurse for 50
recurse for 49
recurse for 48
recurse for 47
...
recurse for 5
recurse for 4
recurse for 3
recurse for 2
1 == 1
2 == 1
3 == 2
4 == 3
5 == 5
6 == 8
...
47 == 2971215073
48 == 4807526976
49 == 7778742049
50 == 12586269025
```

To achieve this behavior in Java, the example inserts a single **Fibonacci** object with a sequence field of **50**. The example then uses a recursive rule to insert the other 49 **Fibonacci** objects.

Instead of implementing the **PropertyChangeSupport** interface to use dynamic facts, this example uses the MVEL dialect **modify** keyword to enable a block setter action and notify the decision engine of changes.

Fibonacci example execution

```
ksession.insert( new Fibonacci( 50 ) );
ksession.fireAllRules();
```

This example uses the following three rules:

- **"Recurse"**
- **"Bootstrap"**
- **"Calculate"**

The rule **"Recurse"** matches each asserted **Fibonacci** object with a value of **-1**, creating and asserting a new **Fibonacci** object with a sequence of one less than the currently matched object. Each time a Fibonacci object is added while the one with a sequence field equal to **1** does not exist, the rule re-matches and fires again. The **not** conditional element is used to stop the rule matching once you have all 50 Fibonacci objects in memory. The rule also has a **salience** value because you need to have all 50 **Fibonacci** objects asserted before you execute the **"Bootstrap"** rule.

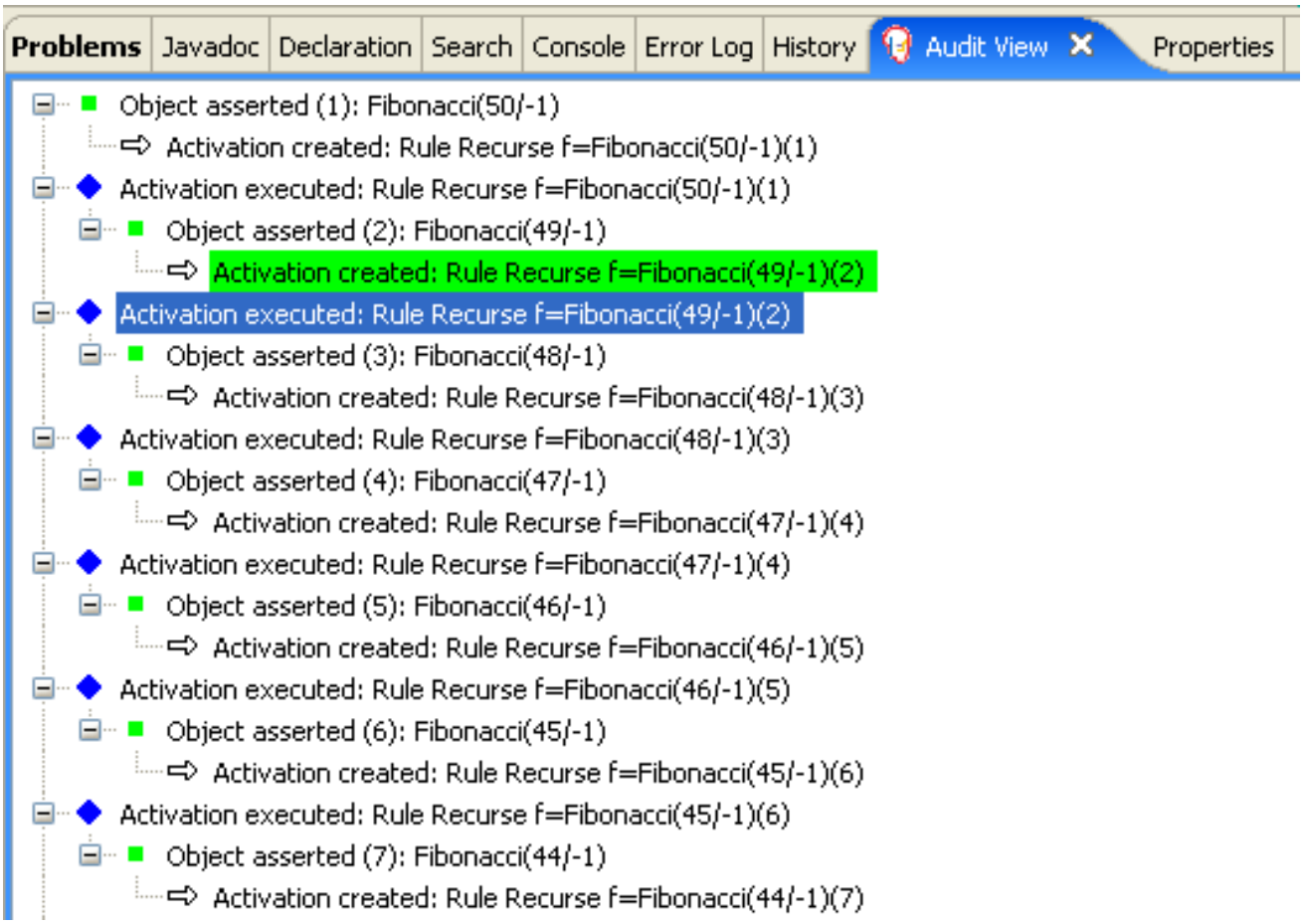
Rule "Recurse"

```
rule "Recurse"
    salience 10
  when
    f : Fibonacci ( value == -1 )
    not ( Fibonacci ( sequence == 1 ) )
  then
    insert( new Fibonacci( f.sequence - 1 ) );
    System.out.println( "recurse for " + f.sequence );
end
```

To better understand the execution flow of this example, you can load the audit log file from **target/fibonacci.log** into your IDE debug view or **Audit View**, if available (for example, in **Window → Show View** in some IDEs).

In this example, the **Audit View** shows the original assertion of the **Fibonacci** object with a **sequence** field of **50**, done from Java code. From there on, the **Audit View** shows the continual recursion of the rule, where each asserted **Fibonacci** object causes the **"Recurse"** rule to become activated and to fire again.

Figure 9.7. Rule "Recurse" in Audit View



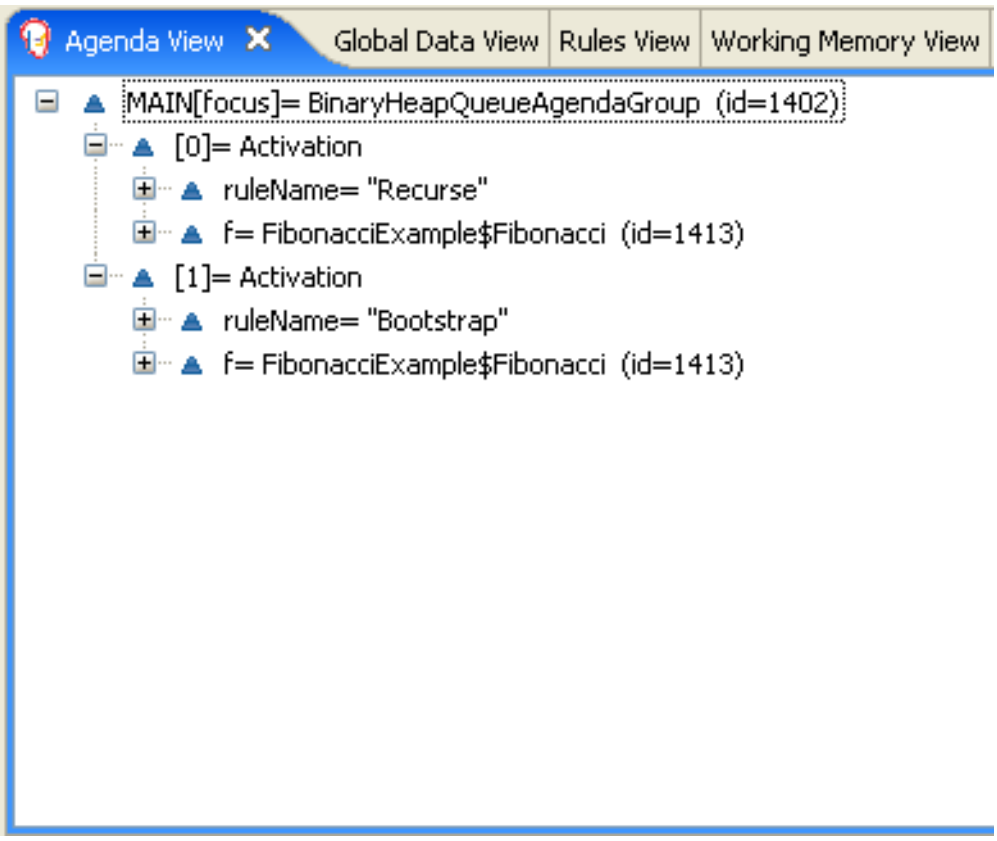
When a **Fibonacci** object with a **sequence** field of **2** is asserted, the **"Bootstrap"** rule is matched and activated along with the **"Recurse"** rule. Notice the multiple restrictions on field **sequence** that test for equality with **1** or **2**:

Rule "Bootstrap"

```
rule "Bootstrap"
  when
    f : Fibonacci( sequence == 1 || == 2, value == -1 ) // multi-restriction
  then
    modify ( f ){ value = 1 };
    System.out.println( f.sequence + " == " + f.value );
end
```

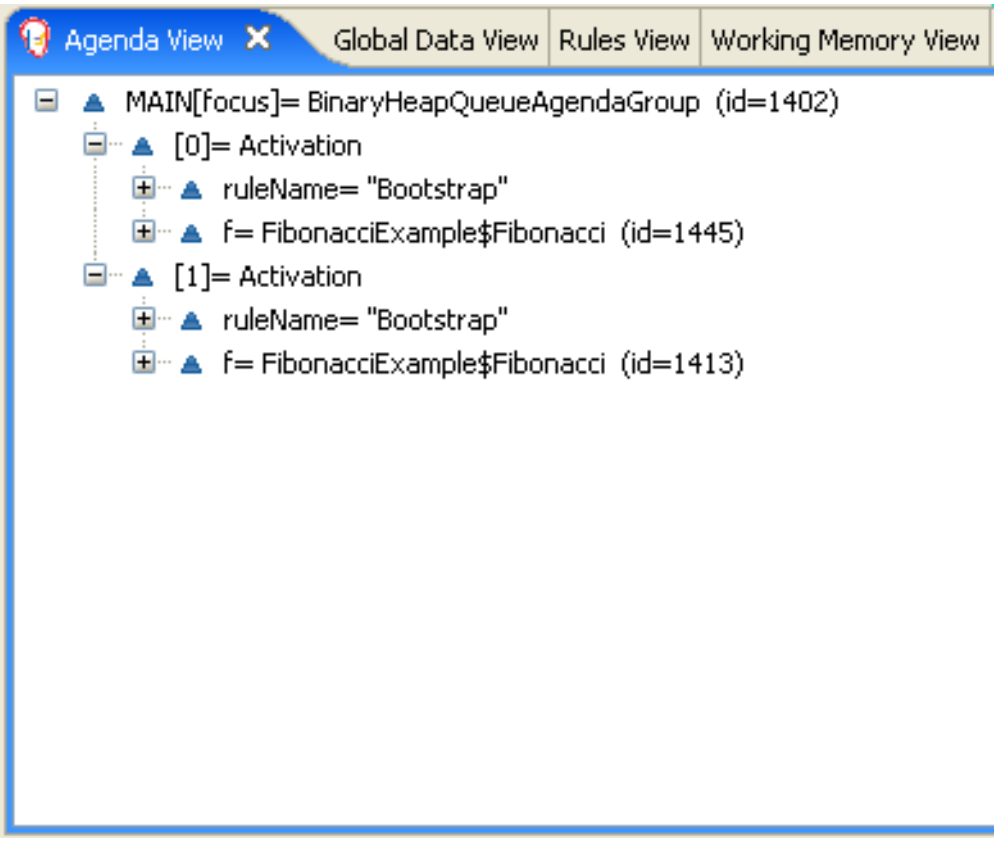
You can also use the **Agenda View** in your IDE to investigate the state of the decision engine agenda. The **"Bootstrap"** rule does not fire yet because the **"Recurse"** rule has a higher salience value.

Figure 9.8. Rules "Recurse" and "Bootstrap" in Agenda View 1



When a **Fibonacci** object with a **sequence** of **1** is asserted, the **"Bootstrap"** rule is matched again, causing two activations for this rule. The **"Recurse"** rule does not match and activate because the **not** conditional element stops the rule matching as soon as a **Fibonacci** object with a **sequence** of **1** exists.

Figure 9.9. Rules "Recurse" and "Bootstrap" in Agenda View 2



The **"Bootstrap"** rule sets the objects with a **sequence** of **1** and **2** to a value of **1**. Now that you have two **Fibonacci** objects with values not equal to **-1**, the **"Calculate"** rule is able to match.

At this point in the example, nearly 50 **Fibonacci** objects exist in the working memory. You need to select a suitable triple to calculate each of their values in turn. If you use three Fibonacci patterns in a rule without field constraints to confine the possible cross products, the result would be 50x49x48 possible combinations, leading to about 125,000 possible rule firings, most of them incorrect.

The **"Calculate"** rule uses field constraints to evaluate the three Fibonacci patterns in the correct order. This technique is called *cross-product matching*.

The first pattern finds any **Fibonacci** object with a value **!= -1** and binds both the pattern and the field. The second **Fibonacci** object does the same thing, but adds an additional field constraint to ensure that its sequence is greater by one than the **Fibonacci** object bound to **f1**. When this rule fires for the first time, you know that only sequences **1** and **2** have values of **1**, and the two constraints ensure that **f1** references sequence **1** and that **f2** references sequence **2**.

The final pattern finds the **Fibonacci** object with a value equal to **-1** and with a sequence one greater than **f2**.

At this point in the example, three **Fibonacci** objects are correctly selected from the available cross products, and you can calculate the value for the third **Fibonacci** object that is bound to **f3**.

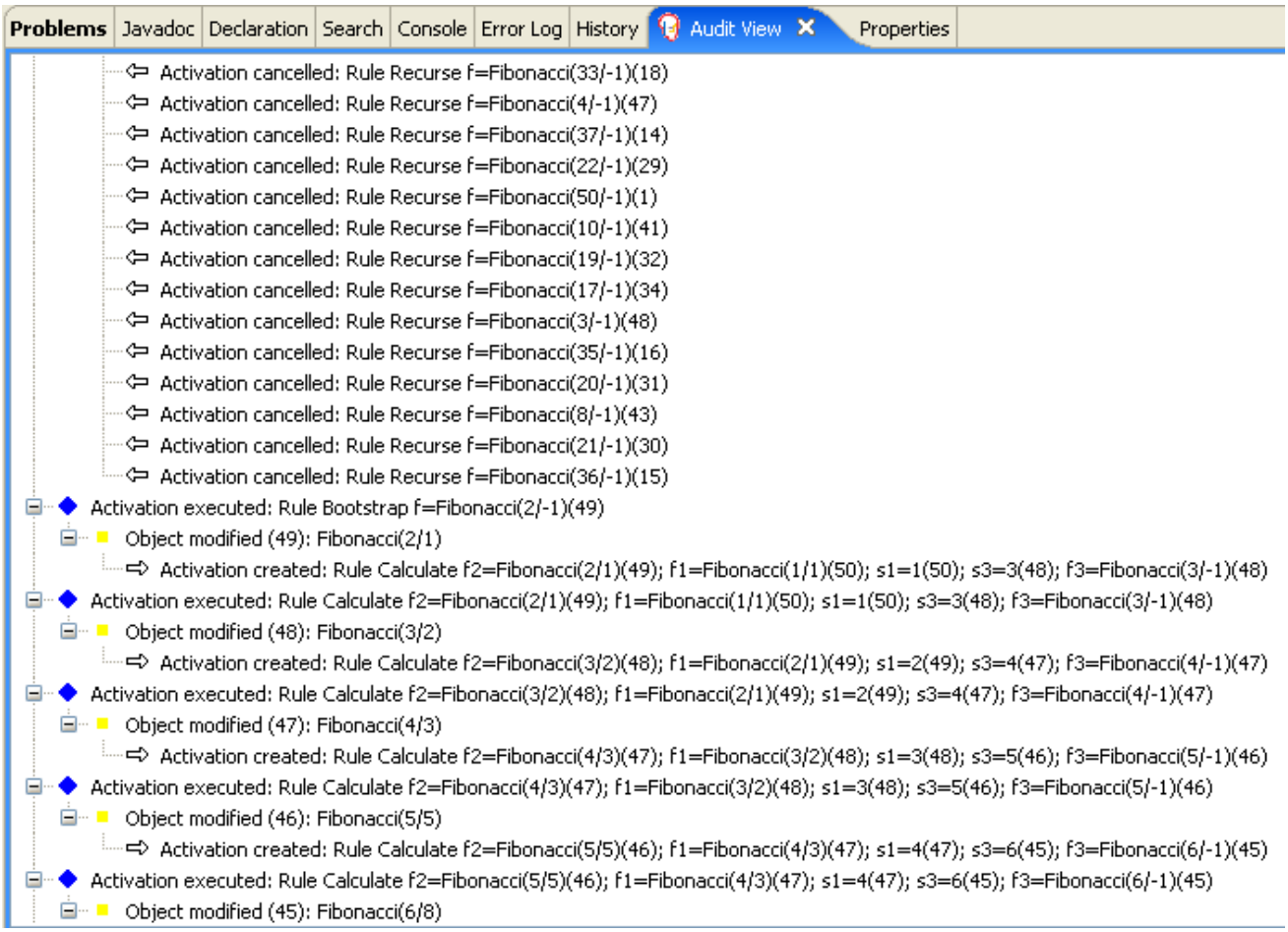
Rule "Calculate"

```
rule "Calculate"
  when
    // Bind f1 and s1.
    f1 : Fibonacci( s1 : sequence, value != -1 )
    // Bind f2 and v2, refer to bound variable s1.
    f2 : Fibonacci( sequence == (s1 + 1), v2 : value != -1 )
    // Bind f3 and s3, alternative reference of f2.sequence.
    f3 : Fibonacci( s3 : sequence == (f2.sequence + 1 ), value == -1 )
  then
    // Note the various referencing techniques.
    modify ( f3 ) { value = f1.value + v2 };
    System.out.println( s3 + " == " + f3.value );
end
```

The **modify** statement updates the value of the **Fibonacci** object bound to **f3**. This means that you now have another new **Fibonacci** object with a value not equal to **-1**, which allows the **"Calculate"** rule to re-match and calculate the next Fibonacci number.

The debug view or **Audit View** of your IDE shows how the firing of the last **"Bootstrap"** rule modifies the **Fibonacci** object, enabling the **"Calculate"** rule to match, which then modifies another **Fibonacci** object that enables the **"Calculate"** rule to match again. This process continues until the value is set for all **Fibonacci** objects.

Figure 9.10. Rules in Audit View



9.5. PRICING EXAMPLE DECISIONS (DECISION TABLES)

The Pricing example decision set demonstrates how to use a spreadsheet decision table for calculating the retail cost of an insurance policy in tabular format instead of directly in a DRL file.

The following is an overview of the Pricing example:

- Name: **decisiontable**
- Main class: **org.drools.examples.decisiontable.PricingRuleDTExample** (in **src/main/java**)
- Module: **drools-examples**
- **Type**: Java application
- Rule file: **org.drools.examples.decisiontable.ExamplePolicyPricing.xls** (in **src/main/resources**)
- **Objective**: Demonstrates use of spreadsheet decision tables to define rules

Spreadsheet decision tables are XLS or XLSX spreadsheets that contain business rules defined in a tabular format. You can include spreadsheet decision tables with standalone Red Hat Decision Manager projects or upload them to projects in Business Central. Each row in a decision table is a rule, and each column is a condition, an action, or another rule attribute. After you create and upload your decision tables into your Red Hat Decision Manager project, the rules you defined are compiled into Drools Rule Language (DRL) rules as with all other rule assets.

The purpose of the Pricing example is to provide a set of business rules to calculate the base price and a discount for a car driver applying for a specific type of insurance policy. The driver's age and history and the policy type all contribute to calculate the basic premium, and additional rules calculate potential discounts for which the driver might be eligible.

To execute the example, run the **org.drools.examples.decisiontable.PricingRuleDTExample** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

```
Cheapest possible
BASE PRICE IS: 120
DISCOUNT IS: 20
```

The code to execute the example follows the typical execution pattern: the rules are loaded, the facts are inserted, and a stateless KIE session is created. The difference in this example is that the rules are defined in an **ExamplePolicyPricing.xls** file instead of a DRL file or other source. The spreadsheet file is loaded into the decision engine using templates and DRL rules.

Spreadsheet decision table setup

The **ExamplePolicyPricing.xls** spreadsheet contains two decision tables in the first tab:

- **Base pricing rules**
- **Promotional discount rules**

As the example spreadsheet demonstrates, you can use only the first tab of a spreadsheet to create decision tables, but multiple tables can be within a single tab. Decision tables do not necessarily follow top-down logic, but are more of a means to capture data resulting in rules. The evaluation of the rules is not necessarily in the given order, because all of the normal mechanics of the decision engine still apply. This is why you can have multiple decision tables in the same tab of a spreadsheet.

The decision tables are executed through the corresponding rule template files **BasePricing.drt** and **PromotionalPricing.drt**. These template files reference the decision tables through their template parameter and directly reference the various headers for the conditions and actions in the decision tables.

BasePricing.drt rule template file

```
template header
age[]
profile
priorClaims
policyType
base
reason

package org.drools.examples.decisiontable;

template "Pricing bracket"
age
policyType
base

rule "Pricing bracket_@{row.rowNumber}"
  when
```

```
    Driver(age >= @{age0}, age <= @{age1}
        , priorClaims == "@{priorClaims}"
        , locationRiskProfile == "@{profile}"
    )
    policy: Policy(type == "@{policyType}")
  then
    policy.setBasePrice(@{base});
    System.out.println("@{reason}");
end
end template
```

PromotionalPricing.drt rule template file

```
template header
age[]
priorClaims
policyType
discount

package org.drools.examples.decisiontable;

template "discounts"
age
priorClaims
policyType
discount

rule "Discounts_@{row.rowNumber}"
  when
    Driver(age >= @{age0}, age <= @{age1}, priorClaims == "@{priorClaims}")
    policy: Policy(type == "@{policyType}")
  then
    policy.applyDiscount(@{discount});
end
end template
```

The rules are executed through the **kmodule.xml** reference of the KIE Session **DTableWithTemplateKB**, which specifically mentions the **ExamplePolicyPricing.xls** spreadsheet and is required for successful execution of the rules. This execution method enables you to execute the rules as a standalone unit (as in this example) or to include the rules in a packaged knowledge JAR (KJAR) file, so that the spreadsheet is packaged along with the rules for execution.

The following section of the **kmodule.xml** file is required for the execution of the rules and spreadsheet to work successfully:

```
    <kbase name="DecisionTableKB" packages="org.drools.examples.decisiontable">
        <ksession name="DecisionTableKS" type="stateless"/>
    </kbase>

    <kbase name="DTableWithTemplateKB" packages="org.drools.examples.decisiontable-template">
        <ruleTemplate dtable="org/drools/examples/decisiontable-
template/ExamplePolicyPricingTemplateData.xls"
                      template="org/drools/examples/decisiontable-template/BasePricing.drt"
                      row="3" col="3"/>
        <ruleTemplate dtable="org/drools/examples/decisiontable-
```

```
template/ExamplePolicyPricingTemplateData.xls"
                template="org/drools/examples/decisiontable-template/PromotionalPricing.drt"
                row="18" col="3"/>
        <ksession name="DTableWithTemplateKS"/>
    </kbase>
```

As an alternative to executing the decision tables using rule template files, you can use the **DecisionTableConfiguration** object and specify an input spreadsheet as the input type, such as **DecisionTableInputType.xls**:

```
DecisionTableConfiguration dtableconfiguration =
    KnowledgeBuilderFactory.newDecisionTableConfiguration();
    dtableconfiguration.setInputType( DecisionTableInputType.XLS );

    KnowledgeBuilder kbuilder = KnowledgeBuilderFactory.newKnowledgeBuilder();

    Resource xlsRes = ResourceFactory.newClassPathResource( "ExamplePolicyPricing.xls",
                                                getClass() );
    kbuilder.add( xlsRes,
                ResourceType.DTABLE,
                dtableconfiguration );
```

The Pricing example uses two fact types:

- **Driver**
- **Policy**.

The example sets the default values for both facts in their respective Java classes **Driver.java** and **Policy.java**. The **Driver** is 30 years old, has had no prior claims, and currently has a risk profile of **LOW**. The **Policy** that the driver is applying for is **COMPREHENSIVE**.

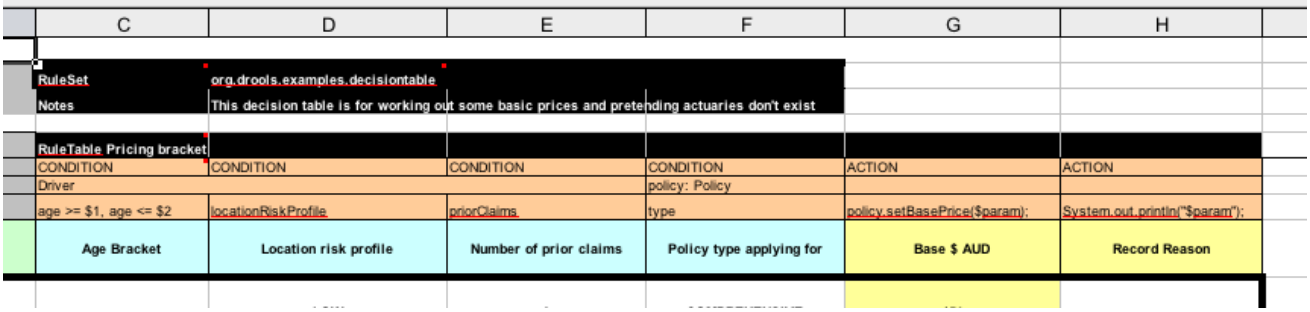
In any decision table, each row is considered a different rule and each column is a condition or an action. Each row is evaluated in a decision table unless the agenda is cleared upon execution.

Decision table spreadsheets (XLS or XLSX) require two key areas that define rule data:

- A **RuleSet** area
- A **RuleTable** area

The **RuleSet** area of the spreadsheet defines elements that you want to apply globally to all rules in the same package (not only the spreadsheet), such as a rule set name or universal rule attributes. The **RuleTable** area defines the actual rules (rows) and the conditions, actions, and other rule attributes (columns) that constitute that rule table within the specified rule set. A decision table spreadsheet can contain multiple **RuleTable** areas, but only one **RuleSet** area.

Figure 9.11. Decision table configuration



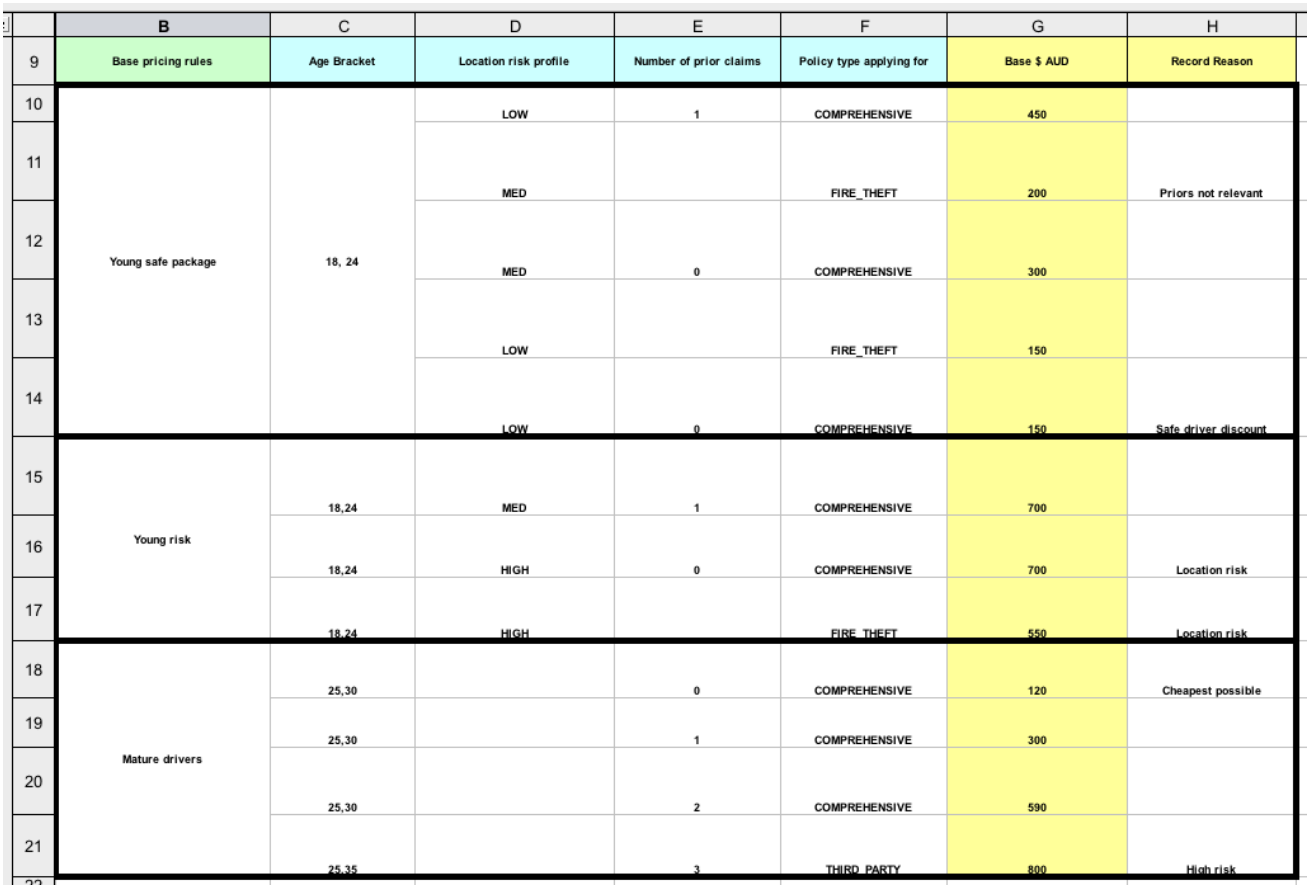
	C	D	E	F	G	H
	RuleSet	org.drools.examples.decisiontable				
	Notes	This decision table is for working out some basic prices and pretending actuaries don't exist				
	RuleTable Pricing bracket					
	CONDITION	CONDITION	CONDITION	CONDITION	ACTION	ACTION
	Driver			policy: Policy		
	age >= $1, age <= $2	locationRiskProfile	priorClaims	type	policy.setBasePrice($param);	System.out.println("$param");
	Age Bracket	Location risk profile	Number of prior claims	Policy type applying for	Base $ AUD	Record Reason

The **RuleTable** area also defines the objects to which the rule attributes apply, in this case **Driver** and **Policy**, followed by constraints on the objects. For example, the **Driver** object constraint that defines the **Age Bracket** column is **age >= $1, age <= $2**, where the comma-separated range is defined in the table column values, such as **18,24**.

Base pricing rules

The **Base pricing rules** decision table in the Pricing example evaluates the age, risk profile, number of claims, and policy type of the driver and produces the base price of the policy based on these conditions.

Figure 9.12. Base price calculation



	B	C	D	E	F	G	H
9	Base pricing rules	Age Bracket	Location risk profile	Number of prior claims	Policy type applying for	Base $ AUD	Record Reason
10	Young safe package	18, 24	LOW	1	COMPREHENSIVE	450	
11			MED		FIRE_THEFT	200	Priors not relevant
12			MED	0	COMPREHENSIVE	300	
13			LOW		FIRE_THEFT	150	
14			LOW	0	COMPREHENSIVE	150	Safe driver discount
15	Young risk	18,24	MED	1	COMPREHENSIVE	700	
16		18,24	HIGH	0	COMPREHENSIVE	700	Location risk
17		18,24	HIGH		FIRE_THEFT	550	Location risk
18	Mature drivers	25,30		0	COMPREHENSIVE	120	Cheapest possible
19		25,30		1	COMPREHENSIVE	300	
20		25,30		2	COMPREHENSIVE	590	
21		25,35		3	THIRD_PARTY	800	High risk

The **Driver** attributes are defined in the following table columns:

- **Age Bracket**: The age bracket has a definition for the condition **age >=$1, age <=$2**, which defines the condition boundaries for the driver's age. This condition column highlights the use of **$1 and $2**, which is comma delimited in the spreadsheet. You can write these values as **18,24** or **18, 24** and both formats work in the execution of the business rules.

- **Location risk profile**: The risk profile is a string that the example program passes always as **LOW** but can be changed to reflect **MED** or **HIGH**.
- **Number of prior claims**: The number of claims is defined as an integer that the condition column must exactly equal to trigger the action. The value is not a range, only exact matches.

The **Policy** of the decision table is used in both the conditions and the actions of the rule and has attributes defined in the following table columns:

- **Policy type applying for**: The policy type is a condition that is passed as a string that defines the type of coverage: **COMPREHENSIVE**, **FIRE_THEFT**, or **THIRD_PARTY**.
- **Base $ AUD**: The **basePrice** is defined as an **ACTION** that sets the price through the constraint **policy.setBasePrice($param);** based on the spreadsheet cells corresponding to this value. When you execute the corresponding DRL rule for this decision table, the **then** portion of the rule executes this action statement on the true conditions matching the facts and sets the base price to the corresponding value.
- **Record Reason**: When the rule successfully executes, this action generates an output message to the **System.out** console reflecting which rule fired. This is later captured in the application and printed.

The example also uses the first column on the left to categorize rules. This column is for annotation only and has no affect on rule execution.

Promotional discount rules

The **Promotional discount rules** decision table in the Pricing example evaluates the age, number of prior claims, and policy type of the driver to generate a potential discount on the price of the insurance policy.

Figure 9.13. Discount calculation

29	Promotional discount rules	Age Bracket	Number of prior claims	Policy type applying for	Discount %
30	Rewards for safe drivers	18,24	0	COMPREHENSIVE	1
31		18,24	0	FIRE_THEFT	2
32		25,30	1	COMPREHENSIVE	5
33		25,30	2	COMPREHENSIVE	1
34		25,30	0	COMPREHENSIVE	20

This decision table contains the conditions for the discount for which the driver might be eligible. Similar to the base price calculation, this table evaluates the **Age**, **Number of prior claims** of the driver, and the **Policy type applying for** to determine a **Discount %** rate to be applied. For example, if the driver is 30 years old, has no prior claims, and is applying for a **COMPREHENSIVE** policy, the driver is given a discount of **20** percent.

9.6. PET STORE EXAMPLE DECISIONS (AGENDA GROUPS, GLOBAL VARIABLES, CALLBACKS, AND GUI INTEGRATION)

The Pet Store example decision set demonstrates how to use agenda groups and global variables in rules and how to integrate Red Hat Decision Manager rules with a graphical user interface (GUI), in this case a Swing-based desktop application. The example also demonstrates how to use callbacks to interact with a running decision engine to update the GUI based on changes in the working memory at run time.

The following is an overview of the Pet Store example:

- Name: **petstore**
- Main class: **org.drools.examples.petstore.PetStoreExample** (in **src/main/java**)
- Module: **drools-examples**
- **Type**: Java application
- Rule file: **org.drools.examples.petstore.PetStore.drl** (in **src/main/resources**)
- **Objective**: Demonstrates rule agenda groups, global variables, callbacks, and GUI integration

In the Pet Store example, the sample **PetStoreExample.java** class defines the following principal classes (in addition to several classes to handle Swing events):

- **Petstore** contains the **main()** method.
- **PetStoreUI** is responsible for creating and displaying the Swing-based GUI. This class contains several smaller classes, mainly for responding to various GUI events, such as user mouse clicks.
- **TableModel** holds the table data. This class is essentially a JavaBean that extends the Swing class **AbstractTableModel**.
- **CheckoutCallback** enables the GUI to interact with the rules.
- **Ordershow** keeps the items that you want to buy.
- **Purchase** stores details of the order and the products that you are buying.
- **Product** is a JavaBean containing details of the product available for purchase and its price.

Much of the Java code in this example is either plain JavaBean or Swing based. For more information about Swing components, see the Java tutorial on Creating a GUI with JFC/Swing.

Rule execution behavior in the Pet Store example

Unlike other example decision sets where the facts are asserted and fired immediately, the Pet Store example does not execute the rules until more facts are gathered based on user interaction. The example executes rules through a **PetStoreUI** object, created by a constructor, that accepts the **Vector** object **stock** for collecting the products. The example then uses an instance of the **CheckoutCallback** class containing the rule base that was previously loaded.

Pet Store KIE container and fact execution setup

```
// KieServices is the factory for all KIE services.
KieServices ks = KieServices.Factory.get();

// Create a KIE container on the class path.
KieContainer kc = ks.getKieClasspathContainer();

// Create the stock.
Vector<Product> stock = new Vector<Product>();
stock.add( new Product( "Gold Fish", 5 ) );
stock.add( new Product( "Fish Tank", 25 ) );
stock.add( new Product( "Fish Food", 2 ) );
```

```
// A callback is responsible for populating the working memory and for firing all rules.
PetStoreUI ui = new PetStoreUI( stock,
                                new CheckoutCallback( kc ) );
ui.createAndShowGUI();
```

The Java code that fires the rules is in the **CheckoutCallBack.checkout()** method. This method is triggered when the user clicks **Checkout** in the UI.

Rule execution from CheckoutCallBack.checkout()

```
public String checkout(JFrame frame, List<Product> items) {
    Order order = new Order();

    // Iterate through list and add to cart.
    for ( Product p: items ) {
        order.addItem( new Purchase( order, p ) );
    }

    // Add the JFrame to the ApplicationData to allow for user interaction.

    // From the KIE container, a KIE session is created based on
    // its definition and configuration in the META-INF/kmodule.xml file.
    KieSession ksession = kcontainer.newKieSession("PetStoreKS");

    ksession.setGlobal( "frame", frame );
    ksession.setGlobal( "textArea", this.output );

    ksession.insert( new Product( "Gold Fish", 5 ) );
    ksession.insert( new Product( "Fish Tank", 25 ) );
    ksession.insert( new Product( "Fish Food", 2 ) );

    ksession.insert( new Product( "Fish Food Sample", 0 ) );

    ksession.insert( order );

    // Execute rules.
    ksession.fireAllRules();

    // Return the state of the cart
    return order.toString();
}
```

The example code passes two elements into the **CheckoutCallBack.checkout()** method. One element is the handle for the **JFrame** Swing component surrounding the output text frame, found at the bottom of the GUI. The second element is a list of order items, which comes from the **TableModel** that stores the information from the **Table** area at the upper-right section of the GUI.

The **for** loop transforms the list of order items coming from the GUI into the **Order** JavaBean, also contained in the file **PetStoreExample.java**.

In this case, the rule is firing in a stateless KIE session because all of the data is stored in Swing components and is not executed until the user clicks **Checkout** in the UI. Each time the user clicks **Checkout**, the content of the list is moved from the Swing **TableModel** into the KIE session working memory and is then executed with the **ksession.fireAllRules()** method.

Within this code, there are nine calls to **KieSession**. The first of these creates a new **KieSession** from the **KieContainer** (the example passed in this **KieContainer** from the **CheckoutCallBack** class in the **main()** method). The next two calls pass in the two objects that hold the global variables in the rules: the Swing text area and the Swing frame used for writing messages. More inserts put information on products into the **KieSession**, as well as the order list. The final call is the standard **fireAllRules()**.

Pet Store rule file imports, global variables, and Java functions

The **PetStore.drl** file contains the standard package and import statements to make various Java classes available to the rules. The rule file also includes *global variables* to be used within the rules, defined as **frame** and **textArea**. The global variables hold references to the Swing components **JFrame** and **JTextArea** components that were previously passed on by the Java code that called the **setGlobal()** method. Unlike standard variables in rules, which expire as soon as the rule has fired, global variables retain their value for the lifetime of the KIE session. This means the contents of these global variables are available for evaluation on all subsequent rules.

PetStore.drl package, imports, and global variables

```
package org.drools.examples;

import org.kie.api.runtime.KieRuntime;
import org.drools.examples.petstore.PetStoreExample.Order;
import org.drools.examples.petstore.PetStoreExample.Purchase;
import org.drools.examples.petstore.PetStoreExample.Product;
import java.util.ArrayList;
import javax.swing.JOptionPane;

import javax.swing.JFrame;

global JFrame frame
global javax.swing.JTextArea textArea
```

The **PetStore.drl** file also contains two functions that the rules in the file use:

PetStore.drl Java functions

```
function void doCheckout(JFrame frame, KieRuntime krt) {
        Object[] options = {"Yes",
                            "No"};

        int n = JOptionPane.showOptionDialog(frame,
                                        "Would you like to checkout?",
                                        "",
                                        JOptionPane.YES_NO_OPTION,
                                        JOptionPane.QUESTION_MESSAGE,
                                        null,
                                        options,
                                        options[0]);

       if (n == 0) {
            krt.getAgenda().getAgendaGroup( "checkout" ).setFocus();
       }
}

function boolean requireTank(JFrame frame, KieRuntime krt, Order order, Product fishTank, int total)
{
```

```
        Object[] options = {"Yes",
                            "No"};

        int n = JOptionPane.showOptionDialog(frame,
                                    "Would you like to buy a tank for your " + total + " fish?",
                                    "Purchase Suggestion",
                                    JOptionPane.YES_NO_OPTION,
                                    JOptionPane.QUESTION_MESSAGE,
                                    null,
                                    options,
                                    options[0]);

        System.out.print( "SUGGESTION: Would you like to buy a tank for your "
                          + total + " fish? - " );

        if (n == 0) {
            Purchase purchase = new Purchase( order, fishTank );
            krt.insert( purchase );
            order.addItem( purchase );
            System.out.println( "Yes" );
        } else {
            System.out.println( "No" );
        }
        return true;
    }
```

The two functions perform the following actions:

- **doCheckout()** displays a dialog that asks the user if she or he wants to check out. If the user does, the focus is set to the **checkout** agenda group, enabling rules in that group to (potentially) fire.
- **requireTank()** displays a dialog that asks the user if she or he wants to buy a fish tank. If the user does, a new fish tank **Product** is added to the order list in the working memory.

NOTE

For this example, all rules and functions are within the same rule file for efficiency. In a production environment, you typically separate the rules and functions in different files or build a static Java method and import the files using the import function, such as **import function my.package.name.hello**.

Pet Store rules with agenda groups

Most of the rules in the Pet Store example use agenda groups to control rule execution. Agenda groups allow you to partition the decision engine agenda to provide more execution control over groups of rules. By default, all rules are in the agenda group **MAIN**. You can use the **agenda-group** attribute to specify a different agenda group for the rule.

Initially, a working memory has its focus on the agenda group **MAIN**. Rules in an agenda group only fire when the group receives the focus. You can set the focus either by using the method **setFocus()** or the rule attribute **auto-focus**. The **auto-focus** attribute enables the rule to be given a focus automatically for its agenda group when the rule is matched and activated.

The Pet Store example uses the following agenda groups for rules:

- **"init"**
- **"evaluate"**
- **"show items"**
- **"checkout"**

For example, the sample rule **"Explode Cart"** uses the **"init"** agenda group to ensure that it has the option to fire and insert shopping cart items into the KIE session working memory:

Rule "Explode Cart"

```
// Insert each item in the shopping cart into the working memory.
rule "Explode Cart"
    agenda-group "init"
    auto-focus true
    salience 10
  when
    $order : Order( grossTotal == -1 )
    $item : Purchase() from $order.items
  then
    insert( $item );
    kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "show items" ).setFocus();
    kcontext.getKnowledgeRuntime().getAgenda().getAgendaGroup( "evaluate" ).setFocus();
end
```

This rule matches against all orders that do not yet have their **grossTotal** calculated. The execution loops for each purchase item in that order.

The rule uses the following features related to its agenda group:

- **agenda-group "init"** defines the name of the agenda group. In this case, only one rule is in the group. However, neither the Java code nor a rule consequence sets the focus to this group, and therefore it relies on the **auto-focus** attribute for its chance to fire.
- **auto-focus true** ensures that this rule, while being the only rule in the agenda group, gets a chance to fire when **fireAllRules()** is called from the Java code.
- **kcontext….setFocus()** sets the focus to the **"show items"** and **"evaluate"** agenda groups, enabling their rules to fire. In practice, you loop through all items in the order, insert them into memory, and then fire the other rules after each insertion.

The **"show items"** agenda group contains only one rule, **"Show Items"**. For each purchase in the order currently in the KIE session working memory, the rule logs details to the text area at the bottom of the GUI, based on the **textArea** variable defined in the rule file.

Rule "Show Items"

```
rule "Show Items"
    agenda-group "show items"
  when
    $order : Order()
    $p : Purchase( order == $order )
```

```
  then
   textArea.append( $p.product + "\n");
end
```

The **"evaluate"** agenda group also gains focus from the **"Explode Cart"** rule. This agenda group contains two rules, **"Free Fish Food Sample"** and **"Suggest Tank"**, which are executed in that order.

Rule "Free Fish Food Sample"

```
// Free fish food sample when users buy a goldfish if they did not already buy
// fish food and do not already have a fish food sample.
rule "Free Fish Food Sample"
    agenda-group "evaluate" (1)
  when
    $order : Order()
    not ( $p : Product( name == "Fish Food") && Purchase( product == $p ) ) (2)
    not ( $p : Product( name == "Fish Food Sample") && Purchase( product == $p ) ) (3)
    exists ( $p : Product( name == "Gold Fish") && Purchase( product == $p ) ) (4)
    $fishFoodSample : Product( name == "Fish Food Sample" );
  then
    System.out.println( "Adding free Fish Food Sample to cart" );
    purchase = new Purchase($order, $fishFoodSample);
    insert( purchase );
    $order.addItem( purchase );
end
```

The rule **"Free Fish Food Sample"** fires only if all of the following conditions are true:

1. The agenda group **"evaluate"** is being evaluated in the rules execution.
2. User does not already have fish food.
3. User does not already have a free fish food sample.
4. User has a goldfish in the order.

If the order facts meet all of these requirements, then a new product is created (Fish Food Sample) and is added to the order in working memory.

Rule "Suggest Tank"

```
// Suggest a fish tank if users buy more than five goldfish and
// do not already have a tank.
rule "Suggest Tank"
    agenda-group "evaluate"
  when
    $order : Order()
    not ( $p : Product( name == "Fish Tank") && Purchase( product == $p ) ) (1)
    ArrayList( $total : size > 5 ) from collect( Purchase( product.name == "Gold Fish" ) ) (2)
    $fishTank : Product( name == "Fish Tank" )
  then
    requireTank(frame, kcontext.getKieRuntime(), $order, $fishTank, $total);
end
```

The rule **"Suggest Tank"** fires only if the following conditions are true:

1. User does not have a fish tank in the order.
2. User has more than five fish in the order.

When the rule fires, it calls the **requireTank()** function defined in the rule file. This function displays a dialog that asks the user if she or he wants to buy a fish tank. If the user does, a new fish tank **Product** is added to the order list in the working memory. When the rule calls the **requireTank()** function, the rule passes the **frame** global variable so that the function has a handle for the Swing GUI.

The **"do checkout"** rule in the Pet Store example has no agenda group and no **when** conditions, so the rule is always executed and considered part of the default **MAIN** agenda group.

Rule "do checkout"

```
rule "do checkout"
  when
  then
    doCheckout(frame, kcontext.getKieRuntime());
end
```

When the rule fires, it calls the **doCheckout()** function defined in the rule file. This function displays a dialog that asks the user if she or he wants to check out. If the user does, the focus is set to the **checkout** agenda group, enabling rules in that group to (potentially) fire. When the rule calls the **doCheckout()** function, the rule passes the **frame** global variable so that the function has a handle for the Swing GUI.

NOTE

This example also demonstrates a troubleshooting technique if results are not executing as you expect: You can remove the conditions from the **when** statement of a rule and test the action in the **then** statement to verify that the action is performed correctly.

The **"checkout"** agenda group contains three rules for processing the order checkout and applying any discounts: **"Gross Total"**, **"Apply 5% Discount"**, and **"Apply 10% Discount"**.

Rules "Gross Total", "Apply 5% Discount", and "Apply 10% Discount"

```
rule "Gross Total"
    agenda-group "checkout"
  when
    $order : Order( grossTotal == -1)
    Number( total : doubleValue ) from accumulate( Purchase( $price : product.price ),
                                                   sum( $price ) )
  then
    modify( $order ) { grossTotal = total }
    textArea.append( "\ngross total=" + total + "\n" );
end

rule "Apply 5% Discount"
    agenda-group "checkout"
  when
    $order : Order( grossTotal >= 10 && < 20 )
```

```
  then
    $order.discountedTotal = $order.grossTotal * 0.95;
    textArea.append( "discountedTotal total=" + $order.discountedTotal + "\n" );
end

rule "Apply 10% Discount"
    agenda-group "checkout"
  when
    $order : Order( grossTotal >= 20 )
  then
    $order.discountedTotal = $order.grossTotal * 0.90;
    textArea.append( "discountedTotal total=" + $order.discountedTotal + "\n" );
end
```

If the user has not already calculated the gross total, the **Gross Total** accumulates the product prices into a total, puts this total into the KIE session, and displays it through the Swing **JTextArea** using the **textArea** global variable.

If the gross total is between **10** and **20** (currency units), the **"Apply 5% Discount"** rule calculates the discounted total, adds it to the KIE session, and displays it in the text area.

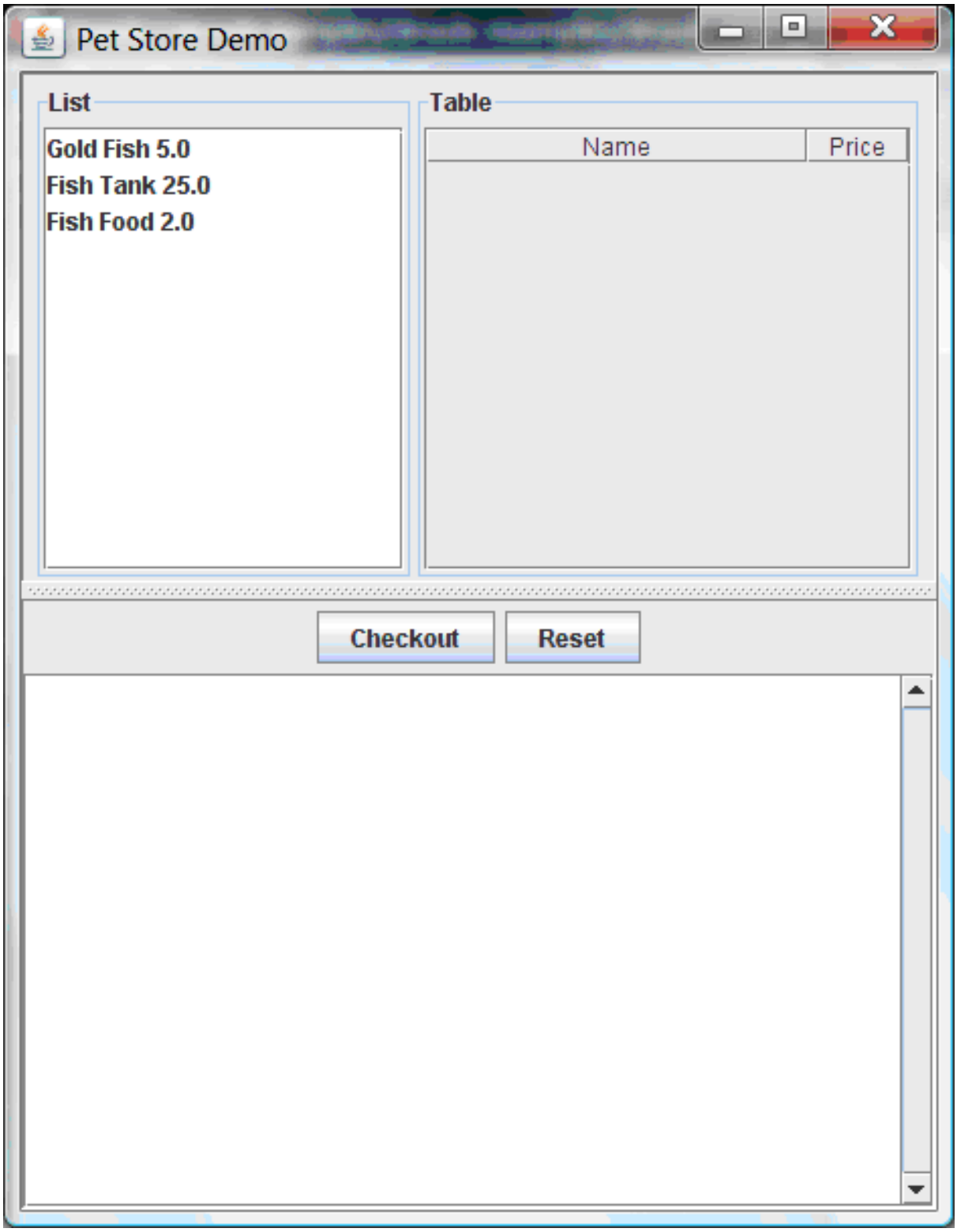
If the gross total is not less than **20**, the **"Apply 10% Discount"** rule calculates the discounted total, adds it to the KIE session, and displays it in the text area.

Pet Store example execution

Similar to other Red Hat Decision Manager decision examples, you execute the Pet Store example by running the **org.drools.examples.petstore.PetStoreExample** class as a Java application in your IDE.

When you execute the Pet Store example, the **Pet Store Demo** GUI window appears. This window displays a list of available products (upper left), an empty list of selected products (upper right), **Checkout** and **Reset** buttons (middle), and an empty system messages area (bottom).

Figure 9.14. Pet Store example GUI after launch

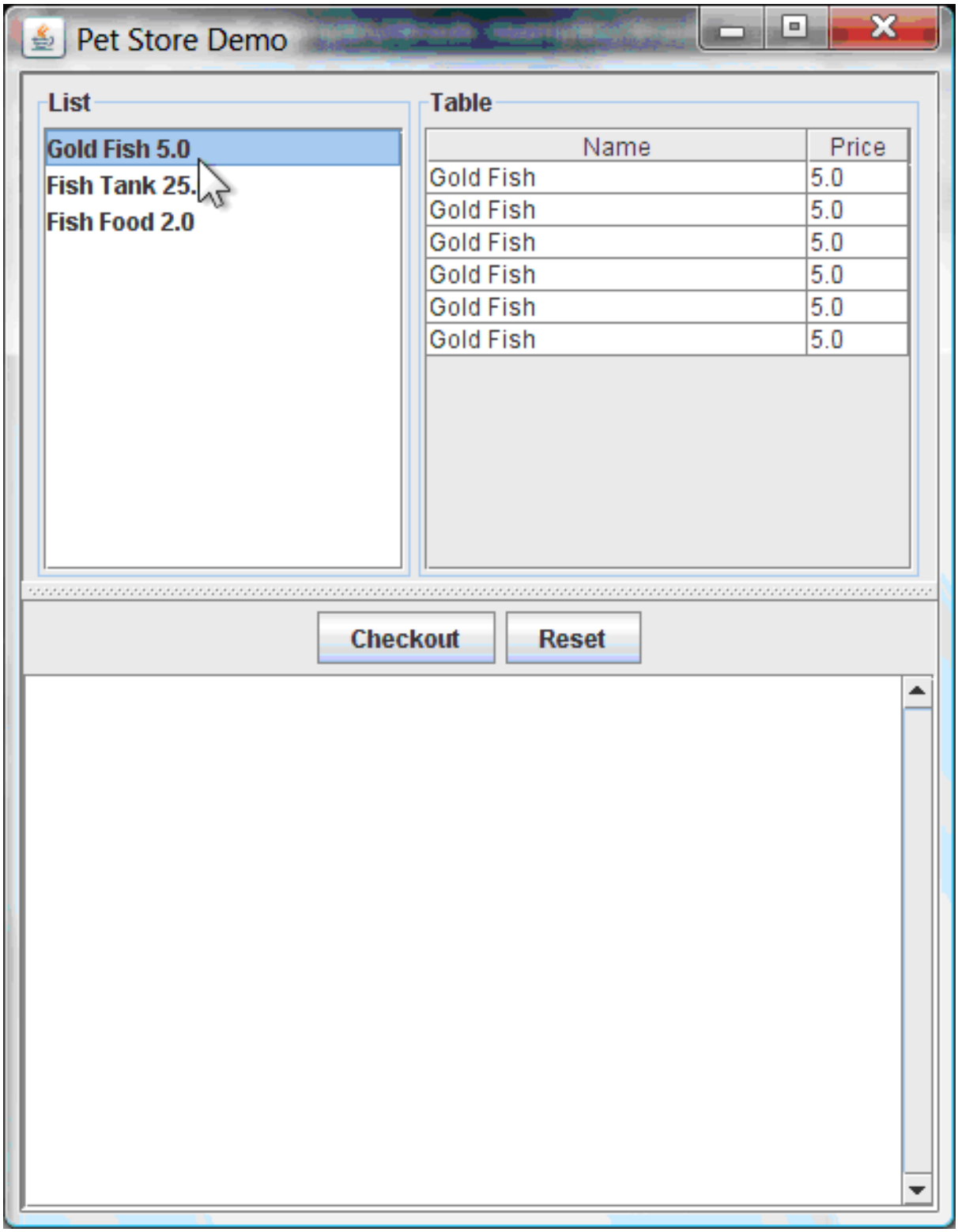


The following events occurred in this example to establish this execution behavior:

1. The **main()** method has run and loaded the rule base but has not yet fired the rules. So far, this is the only code in connection with rules that has been run.
2. A new **PetStoreUI** object has been created and given a handle for the rule base, for later use.
3. Various Swing components have performed their functions, and the initial UI screen is displayed and waits for user input.

You can click various products from the list to explore the UI setup:

Figure 9.15. Explore the Pet Store example GUI



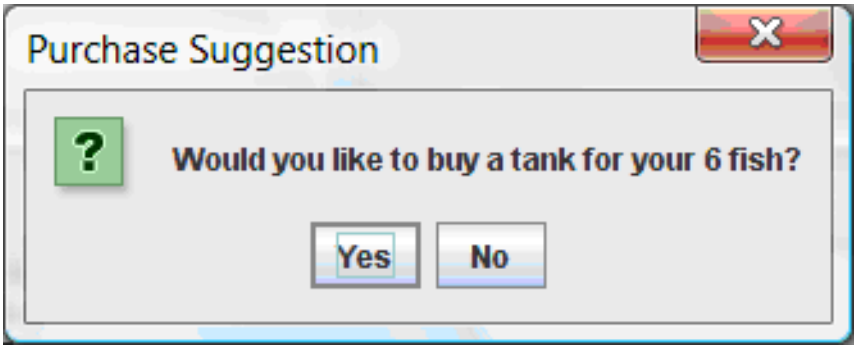
No rules code has been fired yet. The UI uses Swing code to detect user mouse clicks and add selected products to the **TableModel** object for display in the upper-right corner of the UI. This example illustrates the Model-View-Controller design pattern.

When you click **Checkout**, the rules are then fired in the following way:

1. Method **CheckOutCallBack.checkout()** is called (eventually) by the Swing class waiting for a user to click **Checkout**. This inserts the data from the **TableModel** object (upper-right corner of the UI) into the KIE session working memory. The method then fires the rules.

2. The **"Explode Cart"** rule is the first to fire, with the **auto-focus** attribute set to **true**. The rule loops through all of the products in the cart, ensures that the products are in the working memory, and then gives the **"show Items"** and **"evaluate"** agenda groups the option to fire. The rules in these groups add the contents of the cart to the text area (bottom of the UI), evaluate if you are eligible for free fish food, and determine whether to ask if you want to buy a fish tank.

Figure 9.16. Fish tank qualification



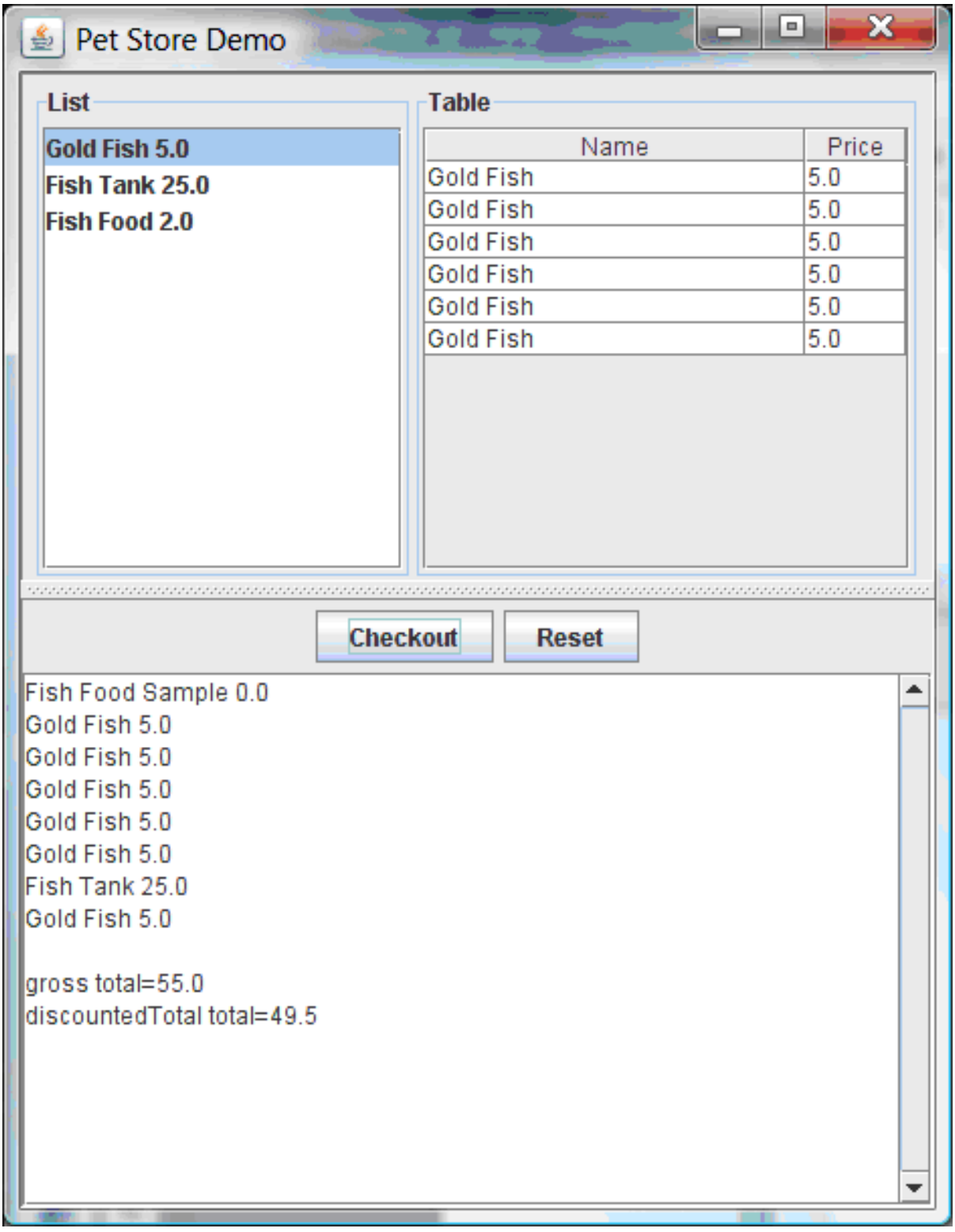
3. The **"do checkout"** rule is the next to fire because no other agenda group currently has focus and because it is part of the default **MAIN** agenda group. This rule always calls the **doCheckout()** function, which asks you if you want to check out.

4. The **doCheckout()** function sets the focus to the **"checkout"** agenda group, giving the rules in that group the option to fire.

5. The rules in the **"checkout"** agenda group display the contents of the cart and apply the appropriate discount.

6. Swing then waits for user input to either select more products (and cause the rules to fire again) or to close the UI.

Figure 9.17. Pet Store example GUI after all rules have fired



You can add more **System.out** calls to demonstrate this flow of events in your IDE console:

System.out output in the IDE console

```
Adding free Fish Food Sample to cart
SUGGESTION: Would you like to buy a tank for your 6 fish? - Yes
```

9.7. HONEST POLITICIAN EXAMPLE DECISIONS (TRUTH MAINTENANCE AND SALIENCE)

The Honest Politician example decision set demonstrates the concept of truth maintenance with logical insertions and the use of salience in rules.

The following is an overview of the Honest Politician example:

- Name: **honestpolitician**
- Main class: **org.drools.examples.honestpolitician.HonestPoliticianExample** (in **src/main/java**)
- Module: **drools-examples**
- **Type**: Java application
- Rule file: **org.drools.examples.honestpolitician.HonestPolitician.drl** (in **src/main/resources**)
- **Objective**: Demonstrates the concept of truth maintenance based on the logical insertion of facts and the use of salience in rules

The basic premise of the Honest Politician example is that an object can only exist while a statement is true. A rule consequence can logically insert an object with the **insertLogical()** method. This means the object remains in the KIE session working memory as long as the rule that logically inserted it remains true. When the rule is no longer true, the object is automatically retracted.

In this example, rule execution causes a group of politicians to change from being honest to being dishonest as a result of a corrupt corporation. As each politician is evaluated, they start out with their honesty attribute being set to **true**, but a rule fires that makes the politicians no longer honest. As they switch their state from being honest to dishonest, they are then removed from the working memory. The rule salience notifies the decision engine how to prioritize any rules that have a salience defined for them, otherwise utilizing the default salience value of **0**. Rules with a higher salience value are given higher priority when ordered in the activation queue.

Politician and Hope classes

The sample class **Politician** in the example is configured for an honest politician. The **Politician** class is made up of a String item **name** and a Boolean item **honest**:

Politician class

```
public class Politician {
    private String name;
    private boolean honest;
    ...
}
```

The **Hope** class determines if a **Hope** object exists. This class has no meaningful members, but is present in the working memory as long as society has hope.

Hope class

```
public class Hope {

    public Hope() {

    }
  }
```

Rule definitions for politician honesty

In the Honest Politician example, when at least one honest politician exists in the working memory, the **"We have an honest Politician"** rule logically inserts a new **Hope** object. As soon as all politicians become dishonest, the **Hope** object is automatically retracted. This rule has a **salience** attribute with a value of **10** to ensure that it fires before any other rule, because at that stage the **"Hope is Dead"** rule is true.

Rule "We have an honest politician"

```
rule "We have an honest Politician"
    salience 10
  when
    exists( Politician( honest == true ) )
  then
    insertLogical( new Hope() );
end
```

As soon as a **Hope** object exists, the **"Hope Lives"** rule matches and fires. This rule also has a **salience** value of **10** so that it takes priority over the **"Corrupt the Honest"** rule.

Rule "Hope Lives"

```
rule "Hope Lives"
    salience 10
  when
    exists( Hope() )
  then
    System.out.println("Hurrah!!! Democracy Lives");
end
```

Initially, four honest politicians exist so this rule has four activations, all in conflict. Each rule fires in turn, corrupting each politician so that they are no longer honest. When all four politicians have been corrupted, no politicians have the property **honest == true**. The rule **"We have an honest Politician"** is no longer true and the object it logically inserted (due to the last execution of **new Hope()**) is automatically retracted.

Rule "Corrupt the Honest"

```
rule "Corrupt the Honest"
  when
    politician : Politician( honest == true )
    exists( Hope() )
  then
    System.out.println( "I'm an evil corporation and I have corrupted " + politician.getName() );
    modify ( politician ) { honest = false };
end
```

With the **Hope** object automatically retracted through the truth maintenance system, the conditional element **not** applied to **Hope** is no longer true so that the **"Hope is Dead"** rule matches and fires.

Rule "Hope is Dead"

```
rule "Hope is Dead"
  when
```

```
    not( Hope() )
  then
    System.out.println( "We are all Doomed!!! Democracy is Dead" );
end
```

Example execution and audit trail

In the **HonestPoliticianExample.java** class, the four politicians with the honest state set to **true** are inserted for evaluation against the defined business rules:

HonestPoliticianExample.java class execution

```
public static void execute( KieContainer kc ) {
        KieSession ksession = kc.newKieSession("HonestPoliticianKS");

        final Politician p1 = new Politician( "President of Umpa Lumpa", true );
        final Politician p2 = new Politician( "Prime Minster of Cheeseland", true );
        final Politician p3 = new Politician( "Tsar of Pringapopaloo", true );
        final Politician p4 = new Politician( "Omnipotence Om", true );

        ksession.insert( p1 );
        ksession.insert( p2 );
        ksession.insert( p3 );
        ksession.insert( p4 );

        ksession.fireAllRules();

        ksession.dispose();
    }
```

To execute the example, run the **org.drools.examples.honestpolitician.HonestPoliticianExample** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

Execution output in the IDE console

```
Hurrah!!! Democracy Lives
I'm an evil corporation and I have corrupted President of Umpa Lumpa
I'm an evil corporation and I have corrupted Prime Minster of Cheeseland
I'm an evil corporation and I have corrupted Tsar of Pringapopaloo
I'm an evil corporation and I have corrupted Omnipotence Om
We are all Doomed!!! Democracy is Dead
```

The output shows that, while there is at least one honest politician, democracy lives. However, as each politician is corrupted by some corporation, all politicians become dishonest, and democracy is dead.

To better understand the execution flow of this example, you can modify the **HonestPoliticianExample.java** class to include a **DebugRuleRuntimeEventListener** listener and an audit logger to view execution details:

HonestPoliticianExample.java class with an audit logger

```
package org.drools.examples.honestpolitician;

import org.kie.api.KieServices;
```

```
import org.kie.api.event.rule.DebugAgendaEventListener; (1)
import org.kie.api.event.rule.DebugRuleRuntimeEventListener;
import org.kie.api.runtime.KieContainer;
import org.kie.api.runtime.KieSession;

public class HonestPoliticianExample {

    /**
     * @param args
     */
    public static void main(final String[] args) {
     KieServices ks = KieServices.Factory.get(); (2)
     //ks = KieServices.Factory.get();
        KieContainer kc = KieServices.Factory.get().getKieClasspathContainer();
        System.out.println(kc.verify().getMessages().toString());
        //execute( kc );
        execute( ks, kc); (3)
    }

    public static void execute( KieServices ks, KieContainer kc ) { (4)
        KieSession ksession = kc.newKieSession("HonestPoliticianKS");

        final Politician p1 = new Politician( "President of Umpa Lumpa", true );
        final Politician p2 = new Politician( "Prime Minster of Cheeseland", true );
        final Politician p3 = new Politician( "Tsar of Pringapopaloo", true );
        final Politician p4 = new Politician( "Omnipotence Om", true );

        ksession.insert( p1 );
        ksession.insert( p2 );
        ksession.insert( p3 );
        ksession.insert( p4 );

        // The application can also setup listeners (5)
        ksession.addEventListener( new DebugAgendaEventListener() );
        ksession.addEventListener( new DebugRuleRuntimeEventListener() );

        // Set up a file-based audit logger.
        ks.getLoggers().newFileLogger( ksession, "./target/honestpolitician" ); (6)

        ksession.fireAllRules();

        ksession.dispose();
    }

}
```

(1) Adds to your imports the packages that handle the **DebugAgendaEventListener** and **DebugRuleRuntimeEventListener**

(2) Creates a **KieServices Factory** and a **ks** element to produce the logs because this audit log is not available at the **KieContainer** level

(3) Modifies the **execute** method to use both **KieServices** and **KieContainer**

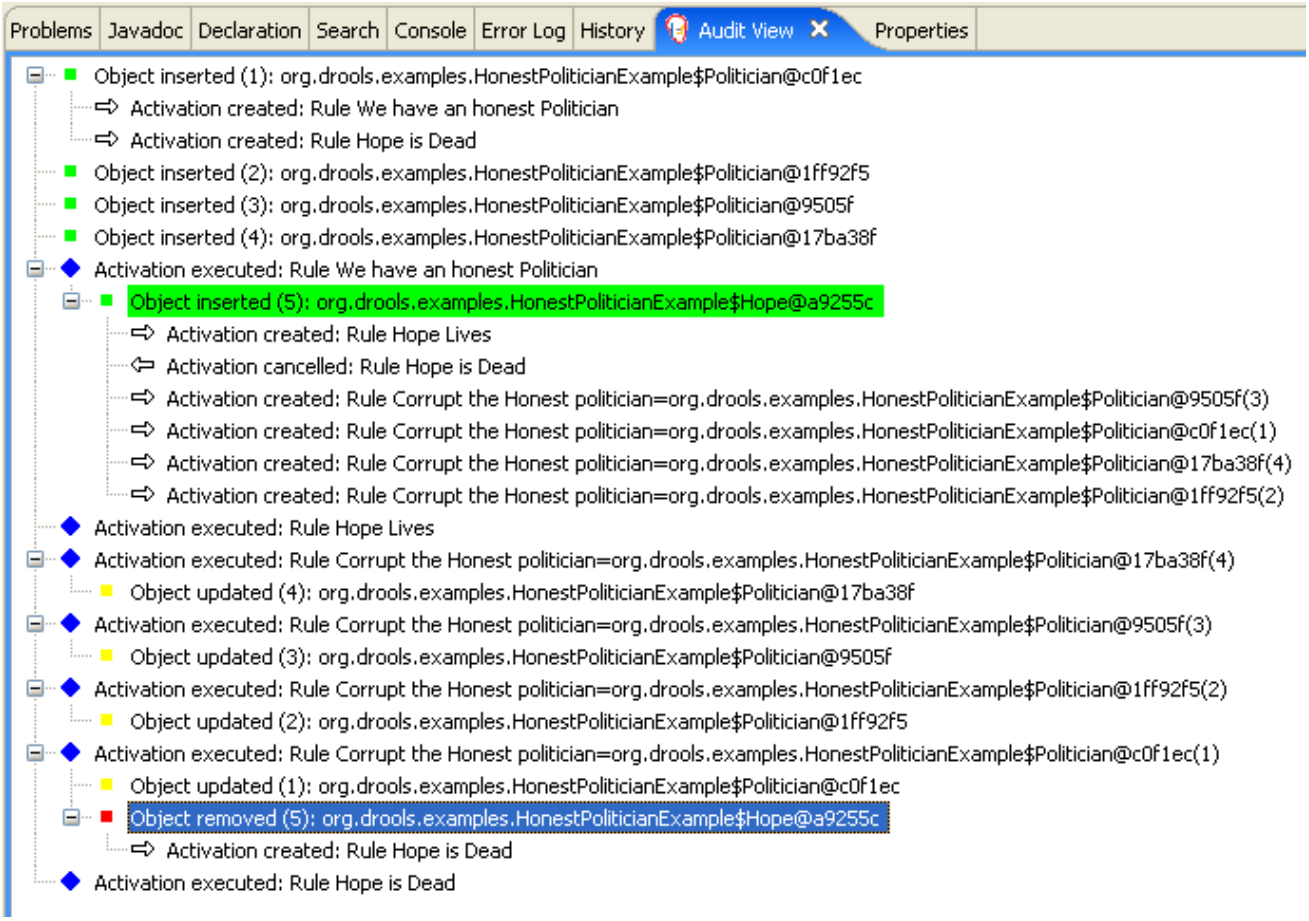
(4) Modifies the **execute** method to pass in **KieServices** in addition to the **KieContainer**

5. Creates the listeners
6. Builds the log that can be passed into the debug view or **Audit View** or your IDE after executing of the rules

When you run the Honest Politician with this modified logging capability, you can load the audit log file from **target/honestpolitician.log** into your IDE debug view or **Audit View**, if available (for example, in **Window → Show View** in some IDEs).

In this example, the **Audit View** shows the flow of executions, insertions, and retractions as defined in the example classes and rules:

Figure 9.18. Honest Politician example Audit View



When the first politician is inserted, two activations occur. The rule **"We have an honest Politician"** is activated only one time for the first inserted politician because it uses an **exists** conditional element, which matches when at least one politician is inserted. The rule **"Hope is Dead"** is also activated at this stage because the **Hope** object is not yet inserted. The rule **"We have an honest Politician"** fires first because it has a higher **salience** value than the rule **"Hope is Dead"**, and inserts the **Hope** object (highlighted in green). The insertion of the **Hope** object activates the rule **"Hope Lives"** and deactivates the rule **"Hope is Dead"**. The insertion also activates the rule **"Corrupt the Honest"** for each inserted honest politician. The rule **"Hope Lives"** is executed and prints **"Hurrah!!! Democracy Lives"**.

Next, for each politician, the rule **"Corrupt the Honest"** fires, printing **"I'm an evil corporation and I have corrupted X"**, where **X** is the name of the politician, and modifies the politician honesty value to **false**. When the last honest politician is corrupted, **Hope** is automatically retracted by the truth maintenance system (highlighted in blue). The green highlighted area shows the origin of the currently selected blue highlighted area. After the **Hope** fact is retracted, the rule **"Hope is dead"** fires, printing **"We are all Doomed!!! Democracy is Dead"**.

9.8. SUDOKU EXAMPLE DECISIONS (COMPLEX PATTERN MATCHING, CALLBACKS, AND GUI INTEGRATION)

The Sudoku example decision set, based on the popular number puzzle Sudoku, demonstrates how to use rules in Red Hat Decision Manager to find a solution in a large potential solution space based on various constraints. This example also shows how to integrate Red Hat Decision Manager rules into a graphical user interface (GUI), in this case a Swing-based desktop application, and how to use callbacks to interact with a running decision engine to update the GUI based on changes in the working memory at run time.

The following is an overview of the Sudoku example:

- Name: **sudoku**
- Main class: **org.drools.examples.sudoku.SudokuExample** (in **src/main/java**)
- Module: **drools-examples**
- **Type**: Java application
- Rule files: **org.drools.examples.sudoku.*.drl** (in **src/main/resources**)
- **Objective**: Demonstrates complex pattern matching, problem solving, callbacks, and GUI integration

Sudoku is a logic-based number placement puzzle. The objective is to fill a 9x9 grid so that each column, each row, and each of the nine 3x3 zones contains the digits from 1 to 9 only one time. The puzzle setter provides a partially completed grid and the puzzle solver's task is to complete the grid with these constraints.

The general strategy to solve the problem is to ensure that when you insert a new number, it must be unique in its particular 3x3 zone, row, and column. This Sudoku example decision set uses Red Hat Decision Manager rules to solve Sudoku puzzles from a range of difficulty levels, and to attempt to resolve flawed puzzles that contain invalid entries.

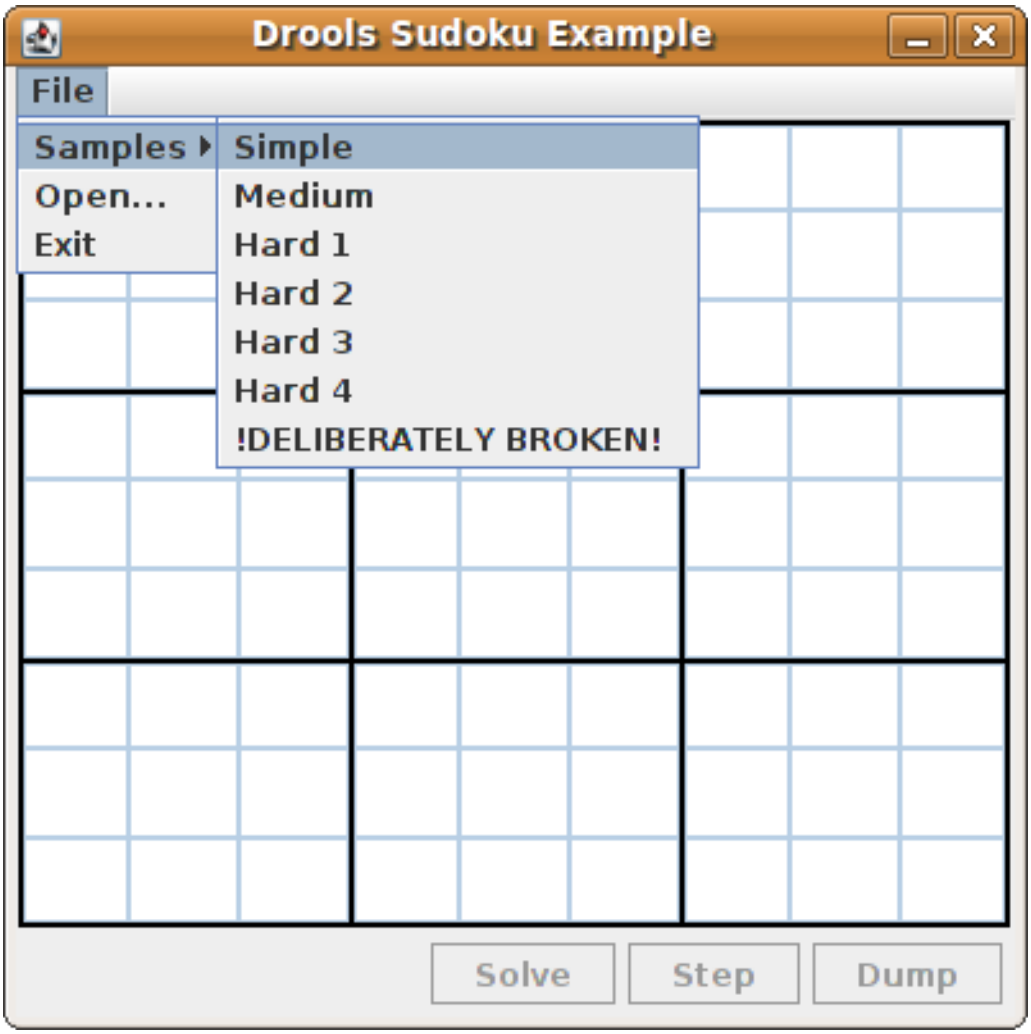
Sudoku example execution and interaction

Similar to other Red Hat Decision Manager decision examples, you execute the Sudoku example by running the **org.drools.examples.sudoku.SudokuExample** class as a Java application in your IDE.

When you execute the Sudoku example, the **Drools Sudoku Example** GUI window appears. This window contains an empty grid, but the program comes with various grids stored internally that you can load and solve.

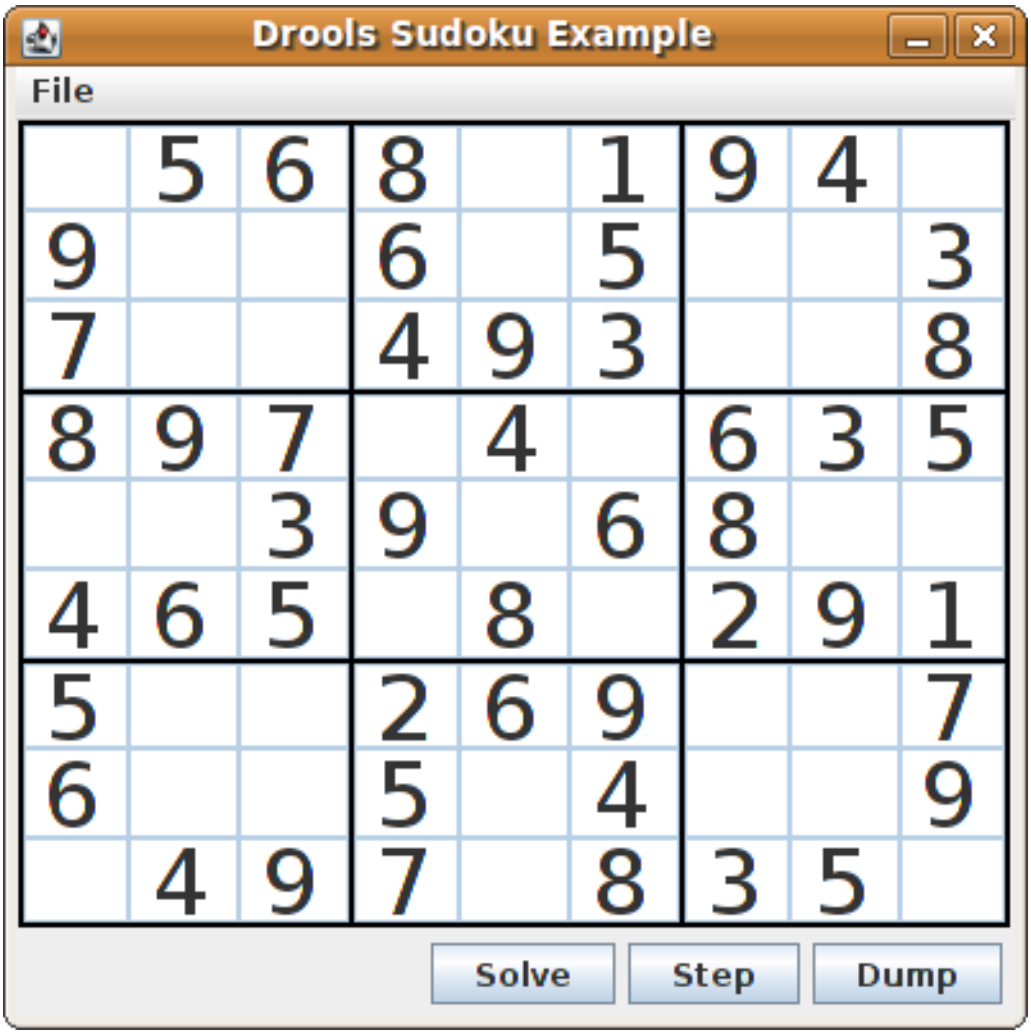
Click **File → Samples → Simple** to load one of the examples. Notice that all buttons are disabled until a grid is loaded.

Figure 9.19. Sudoku example GUI after launch



When you load the **Simple** example, the grid is filled according to the puzzle's initial state.

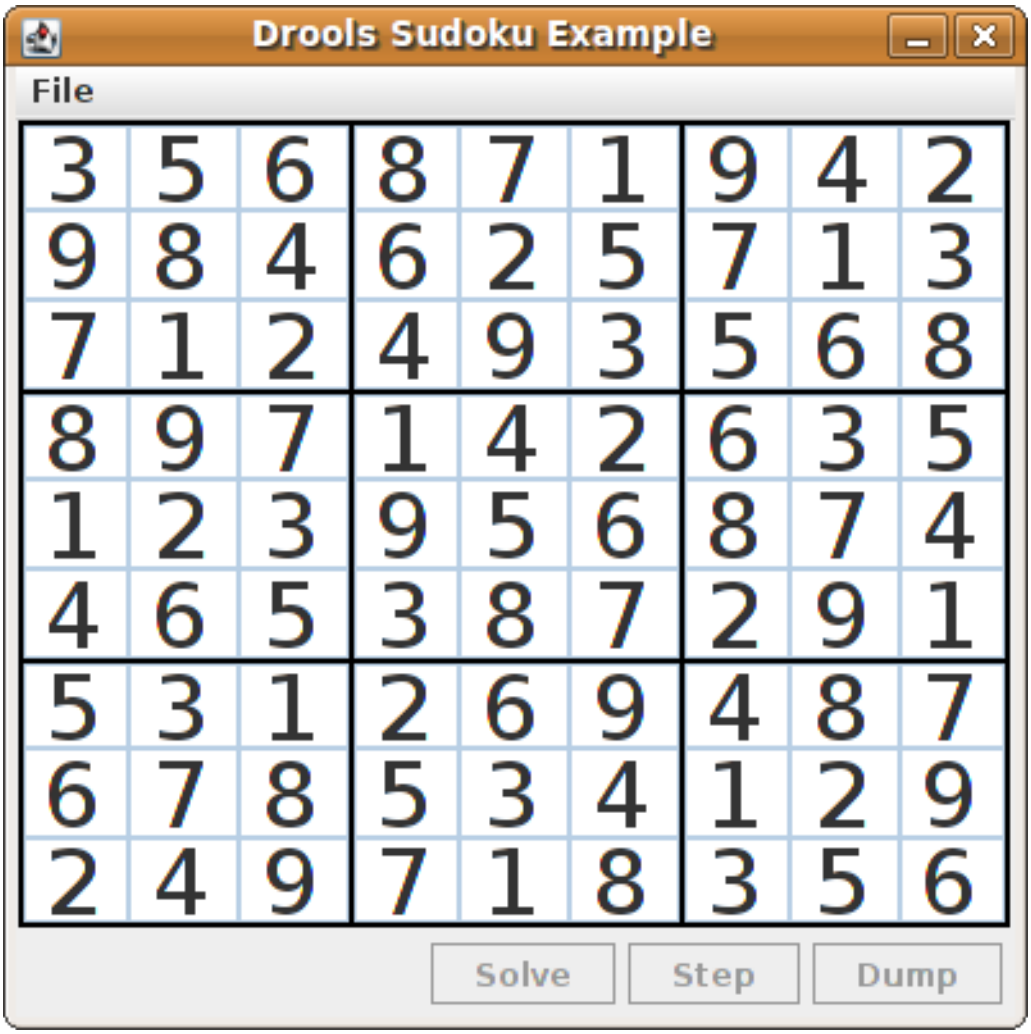
Figure 9.20. Sudoku example GUI after loading Simple sample



Choose from the following options:

- Click **Solve** to fire the rules defined in the Sudoku example that fill out the remaining values and that make the buttons inactive again.

Figure 9.21. Simple sample solved



- Click **Step** to see the next digit found by the rule set. The console window in your IDE displays detailed information about the rules that are executing to solve the step.

Step execution output in the IDE console

```
single 8 at [0,1]
column elimination due to [1,2]: remove 9 from [4,2]
hidden single 9 at [1,2]
row elimination due to [2,8]: remove 7 from [2,4]
remove 6 from [3,8] due to naked pair at [3,2] and [3,7]
hidden pair in row at [4,6] and [4,4]
```

- Click **Dump** to see the state of the grid, with cells showing either the established value or the remaining possibilities.

Dump execution output in the IDE console

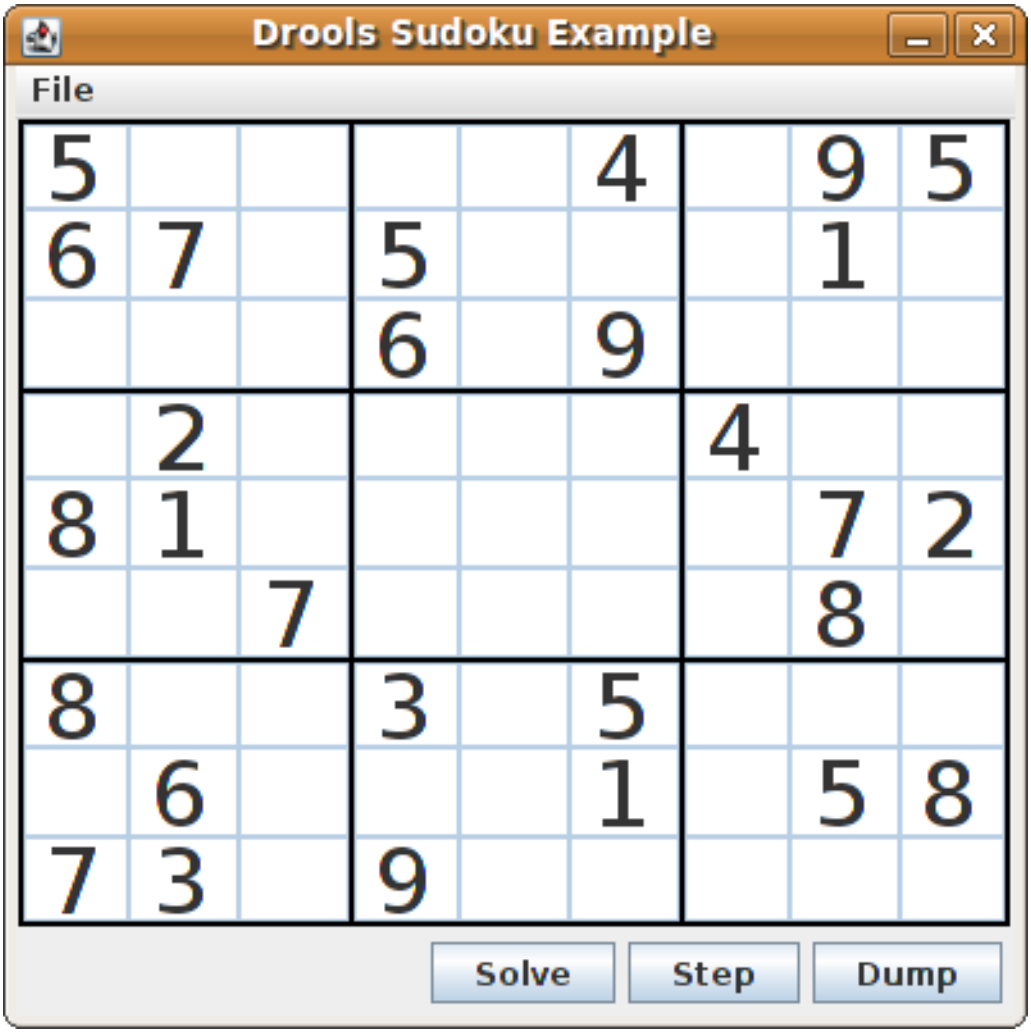
```
      Col: 0     Col: 1     Col: 2     Col: 3     Col: 4     Col: 5     Col: 6     Col: 7     Col: 8
Row 0:  123456789  --- 5 ---  --- 6 ---  --- 8 ---  123456789  --- 1 ---  --- 9 ---  --- 4 ---
123456789
Row 1:  --- 9 ---  123456789  123456789  --- 6 ---  123456789  --- 5 ---  123456789
123456789  --- 3 ---
Row 2:  --- 7 ---  123456789  123456789  --- 4 ---  --- 9 ---  --- 3 ---  123456789  123456789
--- 8 ---
```

```
Row 3:  --- 8 ---  --- 9 ---  --- 7 ---  123456789  --- 4 ---  123456789  --- 6 ---  --- 3 ---  --- 5 ---
Row 4:  123456789  123456789  --- 3 ---  --- 9 ---  123456789  --- 6 ---  --- 8 ---  123456789
123456789
Row 5:  --- 4 ---  --- 6 ---  --- 5 ---  123456789  --- 8 ---  123456789  --- 2 ---  --- 9 ---  --- 1 ---
Row 6:  --- 5 ---  123456789  123456789  --- 2 ---  --- 6 ---  --- 9 ---  123456789  123456789
--- 7 ---
Row 7:  --- 6 ---  123456789  123456789  --- 5 ---  123456789  --- 4 ---  123456789
123456789  --- 9 ---
Row 8:  123456789  --- 4 ---  --- 9 ---  --- 7 ---  123456789  --- 8 ---  --- 3 ---  --- 5 ---
123456789
```

The Sudoku example includes a deliberately broken sample file that the rules defined in the example can resolve.

Click **File → Samples → !DELIBERATELY BROKEN!** to load the broken sample. The grid starts with some issues, for example, the value **5** appears two times in the first row, which is not allowed.

Figure 9.22. Broken Sudoku example initial state



Click **Solve** to apply the solving rules to this invalid grid. The associated solving rules in the Sudoku example detect the issues in the sample and attempts to solve the puzzle as far as possible. This process does not complete and leaves some cells empty.

The solving rule activity is displayed in the IDE console window:

Detected issues in the broken sample

```
cell [0,8]: 5 has a duplicate in row 0
cell [0,0]: 5 has a duplicate in row 0
cell [6,0]: 8 has a duplicate in col 0
cell [4,0]: 8 has a duplicate in col 0
Validation complete.
```

Figure 9.23. Broken sample solution attempt

The sample Sudoku files labeled **Hard** are more complex and the solving rules might not be able to solve them. The unsuccessful solution attempt is displayed in the IDE console window:

Hard sample unresolved

```
Validation complete.
...
Sorry - can't solve this grid.
```

The rules that work to solve the broken sample implement standard solving techniques based on the sets of values that are still candidates for a cell. For example, if a set contains a single value, then this is the value for the cell. For a single occurrence of a value in one of the groups of nine cells, the rules insert a fact of type **Setting** with the solution value for some specific cell. This fact causes the elimination of this value from all other cells in any of the groups the cell belongs to and the value is retracted.

Other rules in the example reduce the permissible values for some cells. The rules **"naked pair"**, **"hidden pair in row"**, **"hidden pair in column"**, and **"hidden pair in square"** eliminate possibilities but do not establish solutions. The rules **"X-wings in rows"**, "`X-wings in columns"`, **"intersection removal**

row", and **"intersection removal column"** perform more sophisticated eliminations.

Sudoku example classes

The package **org.drools.examples.sudoku.swing** contains the following core set of classes that implement a framework for Sudoku puzzles:

- The **SudokuGridModel** class defines an interface that is implemented to store a Sudoku puzzle as a 9x9 grid of **Cell** objects.
- The **SudokuGridView** class is a Swing component that can visualize any implementation of the **SudokuGridModel** class.
- The **SudokuGridEvent** and **SudokuGridListener** classes communicate state changes between the model and the view. Events are fired when a cell value is resolved or changed.
- The **SudokuGridSamples** class provides partially filled Sudoku puzzles for demonstration purposes.

NOTE

This package does not have any dependencies on Red Hat Decision Manager libraries.

The package **org.drools.examples.sudoku** contains the following core set of classes that implement the elementary **Cell** object and its various aggregations:

- The **CellFile** class, with subtypes **CellRow**, **CellCol**, and **CellSqr**, all of which are subtypes of the **CellGroup** class.
- The **Cell** and **CellGroup** subclasses of **SetOfNine**, which provides a property **free** with the type **Set<Integer>**. For a **Cell** class, the set represents the individual candidate set. For a **CellGroup** class, the set is the union of all candidate sets of its cells (the set of digits that still need to be allocated).
  In the Sudoku example are 81 **Cell** and 27 **CellGroup** objects and a linkage provided by the **Cell** properties **cellRow**, **cellCol**, and **cellSqr**, and by the **CellGroup** property **cells** (a list of **Cell** objects). With these components, you can write rules that detect the specific situations that permit the allocation of a value to a cell or the elimination of a value from some candidate set.
- The **Setting** class is used to trigger the operations that accompany the allocation of a value. The presence of a **Setting** fact is used in all rules that detect a new situation in order to avoid reactions to inconsistent intermediary states.
- The **Stepping** class is used in a low priority rule to execute an emergency halt when a **"Step"** does not terminate regularly. This behavior indicates that the program cannot solve the puzzle.
- The main class **org.drools.examples.sudoku.SudokuExample** implements a Java application combining all of these components.

Sudoku validation rules (validate.drl)

The **validate.drl** file in the Sudoku example contains validation rules that detect duplicate numbers in cell groups. They are combined in a **"validate"** agenda group that enables the rules to be explicitly activated after a user loads the puzzle.

The **when** conditions of the three rules **"duplicate in cell ..."** all function in the following ways:

- The first condition in the rule locates a cell with an allocated value.

- The second condition in the rule pulls in any of the three cell groups to which the cell belongs.
- The final condition finds a cell (other than the first one) with the same value as the first cell and in the same row, column, or square, depending on the rule.

Rules "duplicate in cell ..."

```
rule "duplicate in cell row"
  when
    $c: Cell( $v: value != null )
    $cr: CellRow( cells contains $c )
    exists Cell( this != $c, value == $v, cellRow == $cr )
  then
    System.out.println( "cell " + $c.toString() + " has a duplicate in row " + $cr.getNumber() );
end

rule "duplicate in cell col"
  when
    $c: Cell( $v: value != null )
    $cc: CellCol( cells contains $c )
    exists Cell( this != $c, value == $v, cellCol == $cc )
  then
    System.out.println( "cell " + $c.toString() + " has a duplicate in col " + $cc.getNumber() );
end

rule "duplicate in cell sqr"
  when
    $c: Cell( $v: value != null )
    $cs: CellSqr( cells contains $c )
    exists Cell( this != $c, value == $v, cellSqr == $cs )
  then
    System.out.println( "cell " + $c.toString() + " has duplicate in its square of nine." );
end
```

The rule **"terminate group"** is the last to fire. This rule prints a message and stops the sequence.

Rule "terminate group"

```
rule "terminate group"
    salience -100
  when
  then
    System.out.println( "Validation complete." );
    drools.halt();
end
```

Sudoku solving rules (sudoku.drl)

The **sudoku.drl** file in the Sudoku example contains three types of rules: one group handles the allocation of a number to a cell, another group detects feasible allocations, and the third group eliminates values from candidate sets.

The rules **"set a value"**, **"eliminate a value from Cell"**, and **"retract setting"** depend on the presence of a **Setting** object. The first rule handles the assignment to the cell and the operations for removing the value from the **free** sets of the three groups of the cell. This group also reduces a counter that, when zero, returns control to the Java application that has called **fireUntilHalt()**.

The purpose of the rule **"eliminate a value from Cell"** is to reduce the candidate lists of all cells that are related to the newly assigned cell. Finally, when all eliminations have been made, the rule **"retract setting"** retracts the triggering **Setting** fact.

Rules "set a value", "eliminate a value from a Cell", and "retract setting"

```
// A Setting object is inserted to define the value of a Cell.
// Rule for updating the cell and all cell groups that contain it
rule "set a value"
  when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )

    // A matching Cell, with no value set
    $c: Cell( rowNo == $rn, colNo == $cn, value == null,
              $cr: cellRow, $cc: cellCol, $cs: cellSqr )

    // Count down
    $ctr: Counter( $count: count )
  then
    // Modify the Cell by setting its value.
    modify( $c ){ setValue( $v ) }
    // System.out.println( "set cell " + $c.toString() );
    modify( $cr ){ blockValue( $v ) }
    modify( $cc ){ blockValue( $v ) }
    modify( $cs ){ blockValue( $v ) }
    modify( $ctr ){ setCount( $count - 1 ) }
end

// Rule for removing a value from all cells that are siblings
// in one of the three cell groups
rule "eliminate a value from Cell"
  when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )

    // The matching Cell, with the value already set
    Cell( rowNo == $rn, colNo == $cn, value == $v, $exCells: exCells )

    // For all Cells that are associated with the updated cell
    $c: Cell( free contains $v ) from $exCells
  then
    // System.out.println( "clear " + $v + " from cell " + $c.posAsString()  );
    // Modify a related Cell by blocking the assigned value.
    modify( $c ){ blockValue( $v ) }
end

// Rule for eliminating the Setting fact
rule "retract setting"
  when
    // A Setting with row and column number, and a value
    $s: Setting( $rn: rowNo, $cn: colNo, $v: value )

    // The matching Cell, with the value already set
    $c: Cell( rowNo == $rn, colNo == $cn, value == $v )
```

```
    // This is the negation of the last pattern in the previous rule.
    // Now the Setting fact can be safely retracted.
    not( $x: Cell( free contains $v )
         and
         Cell( this == $c, exCells contains $x ) )
  then
    // System.out.println( "done setting cell " + $c.toString() );
    // Discard the Setter fact.
    delete( $s );
    // Sudoku.sudoku.consistencyCheck();
end
```

Two solving rules detect a situation where an allocation of a number to a cell is possible. The rule **"single"** fires for a **Cell** with a candidate set containing a single number. The rule **"hidden single"** fires when no cell exists with a single candidate, but when a cell exists containing a candidate, this candidate is absent from all other cells in one of the three groups to which the cell belongs. Both rules create and insert a **Setting** fact.

Rules "single" and "hidden single"

```
// Detect a set of candidate values with cardinality 1 for some Cell.
// This is the value to be set.
rule "single"
  when
    // Currently no setting underway
    not Setting()

    // One element in the "free" set
    $c: Cell( $rn: rowNo, $cn: colNo, freeCount == 1 )
  then
    Integer i = $c.getFreeValue();
    if (explain) System.out.println( "single " + i + " at " + $c.posAsString() );
    // Insert another Setter fact.
    insert( new Setting( $rn, $cn, i ) );
end

// Detect a set of candidate values with a value that is the only one
// in one of its groups. This is the value to be set.
rule "hidden single"
  when
    // Currently no setting underway
    not Setting()
    not Cell( freeCount == 1 )

    // Some integer
    $i: Integer()

    // The "free" set contains this number
    $c: Cell( $rn: rowNo, $cn: colNo, freeCount > 1, free contains $i )

    // A cell group contains this cell $c.
    $cg: CellGroup( cells contains $c )
    // No other cell from that group contains $i.
    not ( Cell( this != $c, free contains $i ) from $cg.getCells() )
  then
    if (explain) System.out.println( "hidden single " + $i + " at " + $c.posAsString() );
```

```
    // Insert another Setter fact.
    insert( new Setting( $rn, $cn, $i ) );
end
```

Rules from the largest group, either individually or in groups of two or three, implement various solving techniques used for solving Sudoku puzzles manually.

The rule **"naked pair"** detects identical candidate sets of size **2** in two cells of a group. These two values may be removed from all other candidate sets of that group.

Rule "naked pair"

```
// A "naked pair" is two cells in some cell group with their sets of
// permissible values being equal with cardinality 2. These two values
// can be removed from all other candidate lists in the group.
rule "naked pair"
  when
    // Currently no setting underway
    not Setting()
    not Cell( freeCount == 1 )

    // One cell with two candidates
    $c1: Cell( freeCount == 2, $f1: free, $r1: cellRow, $rn1: rowNo, $cn1: colNo, $b1: cellSqr )

    // The containing cell group
    $cg: CellGroup( freeCount > 2, cells contains $c1 )

    // Another cell with two candidates, not the one we already have
    $c2: Cell( this != $c1, free == $f1 /*** , rowNo >= $rn1, colNo >= $cn1 ***/ ) from $cg.cells

    // Get one of the "naked pair".
    Integer( $v: intValue ) from $c1.getFree()

    // Get some other cell with a candidate equal to one from the pair.
    $c3: Cell( this != $c1 && != $c2, freeCount > 1, free contains $v ) from $cg.cells
  then
    if (explain) System.out.println( "remove " + $v + " from " + $c3.posAsString() + " due to naked pair
at " + $c1.posAsString() + " and " + $c2.posAsString() );
    // Remove the value.
    modify( $c3 ){ blockValue( $v ) }
end
```

The three rules **"hidden pair in ..."** functions similarly to the rule **"naked pair"**. These rules detect a subset of two numbers in exactly two cells of a group, with neither value occurring in any of the other cells of the group. This means that all other candidates can be eliminated from the two cells harboring the hidden pair.

Rules "hidden pair in ..."

```
// If two cells within the same cell group contain candidate sets with more than
// two values, with two values being in both of them but in none of the other
// cells, then we have a "hidden pair". We can remove all other candidates from
// these two cells.
rule "hidden pair in row"
  when
```

```
    // Currently no setting underway
    not Setting()
    not Cell( freeCount == 1 )

    // Establish a pair of Integer facts.
    $i1: Integer()
    $i2: Integer( this > $i1 )

    // Look for a Cell with these two among its candidates. (The upper bound on
    // the number of candidates avoids a lot of useless work during startup.)
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free contains $i1 && contains $i2,
$cellRow: cellRow )

    // Get another one from the same row, with the same pair among its candidates.
    $c2: Cell( this != $c1, cellRow == $cellRow, freeCount > 2, free contains $i1 && contains $i2 )

    // Ascertain that no other cell in the group has one of these two values.
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellRow.getCells() )
  then
    if( explain) System.out.println( "hidden pair in row at " + $c1.posAsString() + " and " +
$c2.posAsString() );
    // Set the candidate lists of these two Cells to the "hidden pair".
    modify( $c1 ){ blockExcept( $i1, $i2 ) }
    modify( $c2 ){ blockExcept( $i1, $i2 ) }
end

rule "hidden pair in column"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i1: Integer()
    $i2: Integer( this > $i1 )
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free contains $i1 && contains $i2,
$cellCol: cellCol )
    $c2: Cell( this != $c1, cellCol == $cellCol, freeCount > 2, free contains $i1 && contains $i2 )
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellCol.getCells() )
  then
    if (explain) System.out.println( "hidden pair in column at " + $c1.posAsString() + " and " +
$c2.posAsString() );
    modify( $c1 ){ blockExcept( $i1, $i2 ) }
    modify( $c2 ){ blockExcept( $i1, $i2 ) }
end

rule "hidden pair in square"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i1: Integer()
    $i2: Integer( this > $i1 )
    $c1: Cell( $rn1: rowNo, $cn1: colNo, freeCount > 2 && < 9, free contains $i1 && contains $i2,
            $cellSqr: cellSqr )
    $c2: Cell( this != $c1, cellSqr == $cellSqr, freeCount > 2, free contains $i1 && contains $i2 )
    not( Cell( this != $c1 && != $c2, free contains $i1 || contains $i2 ) from $cellSqr.getCells() )
  then
```

```
    if (explain) System.out.println( "hidden pair in square " + $c1.posAsString() + " and " +
$c2.posAsString() );
    modify( $c1 ){ blockExcept( $i1, $i2 ) }
    modify( $c2 ){ blockExcept( $i1, $i2 ) }
end
```

Two rules deal with **"X-wings"** in rows and columns. When only two possible cells for a value exist in each of two different rows (or columns) and these candidates lie also in the same columns (or rows), then all other candidates for this value in the columns (or rows) can be eliminated. When you follow the pattern sequence in one of these rules, notice how the conditions that are conveniently expressed by words such as **same** or **only** result in patterns with suitable constraints or that are prefixed with **not**.

Rules "X-wings in ..."

```
rule "X-wings in rows"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    $ca1: Cell( freeCount > 1, free contains $i,
                $ra: cellRow, $rano: rowNo,         $c1: cellCol,        $c1no: colNo )
    $cb1: Cell( freeCount > 1, free contains $i,
                $rb: cellRow, $rbno: rowNo > $rano,      cellCol == $c1 )
    not( Cell( this != $ca1 && != $cb1, free contains $i ) from $c1.getCells() )

    $ca2: Cell( freeCount > 1, free contains $i,
                cellRow == $ra, $c2: cellCol,       $c2no: colNo > $c1no )
    $cb2: Cell( freeCount > 1, free contains $i,
                cellRow == $rb,      cellCol == $c2 )
    not( Cell( this != $ca2 && != $cb2, free contains $i ) from $c2.getCells() )

    $cx: Cell( rowNo == $rano || == $rbno, colNo != $c1no && != $c2no,
               freeCount > 1, free contains $i )
  then
    if (explain) {
       System.out.println( "X-wing with " + $i + " in rows " +
          $ca1.posAsString() + " - " + $cb1.posAsString() +
          $ca2.posAsString() + " - " + $cb2.posAsString() + ", remove from " + $cx.posAsString() );
    }
    modify( $cx ){ blockValue( $i ) }
end

rule "X-wings in columns"
  when
    not Setting()
    not Cell( freeCount == 1 )

    $i: Integer()
    $ca1: Cell( freeCount > 1, free contains $i,
                $c1: cellCol, $c1no: colNo,         $ra: cellRow,        $rano: rowNo )
    $ca2: Cell( freeCount > 1, free contains $i,
                $c2: cellCol, $c2no: colNo > $c1no,      cellRow == $ra )
    not( Cell( this != $ca1 && != $ca2, free contains $i ) from $ra.getCells() )

    $cb1: Cell( freeCount > 1, free contains $i,
```

```
            cellCol == $c1, $rb: cellRow,  $rbno: rowNo > $rano )
  $cb2: Cell( freeCount > 1, free contains $i,
            cellCol == $c2,      cellRow == $rb )
  not( Cell( this != $cb1 && != $cb2, free contains $i ) from $rb.getCells() )

  $cx: Cell( colNo == $c1no || == $c2no, rowNo != $rano && != $rbno,
          freeCount > 1, free contains $i )
 then
  if (explain) {
     System.out.println( "X-wing with " + $i + " in columns " +
        $ca1.posAsString() + " - " + $ca2.posAsString() +
        $cb1.posAsString() + " - " + $cb2.posAsString() + ", remove from " + $cx.posAsString()  );
  }
  modify( $cx ){ blockValue( $i ) }
end
```

The two rules **"intersection removal ..."** are based on the restricted occurrence of some number within one square, either in a single row or in a single column. This means that this number must be in one of those two or three cells of the row or column and can be removed from the candidate sets of all other cells of the group. The pattern establishes the restricted occurrence and then fires for each cell outside of the square and within the same cell file.

Rules "intersection removal ..."

```
rule "intersection removal column"
 when
  not Setting()
  not Cell( freeCount == 1 )

  $i: Integer()
  // Occurs in a Cell
  $c: Cell( free contains $i, $cs: cellSqr, $cc: cellCol )
  // Does not occur in another cell of the same square and a different column
  not Cell( this != $c, free contains $i, cellSqr == $cs, cellCol != $cc )

  // A cell exists in the same column and another square containing this value.
  $cx: Cell( freeCount > 1, free contains $i, cellCol == $cc, cellSqr != $cs )
 then
  // Remove the value from that other cell.
  if (explain) {
     System.out.println( "column elimination due to " + $c.posAsString() +
                  ": remove " + $i + " from " + $cx.posAsString() );
  }
  modify( $cx ){ blockValue( $i ) }
end

rule "intersection removal row"
 when
  not Setting()
  not Cell( freeCount == 1 )

  $i: Integer()
  // Occurs in a Cell
  $c: Cell( free contains $i, $cs: cellSqr, $cr: cellRow )
  // Does not occur in another cell of the same square and a different row.
  not Cell( this != $c, free contains $i, cellSqr == $cs, cellRow != $cr )
```

```
    // A cell exists in the same row and another square containing this value.
    $cx: Cell( freeCount > 1, free contains $i, cellRow == $cr, cellSqr != $cs )
  then
    // Remove the value from that other cell.
    if (explain) {
        System.out.println( "row elimination due to " + $c.posAsString() +
                            ": remove " + $i + " from " + $cx.posAsString() );
    }
    modify( $cx ){ blockValue( $i ) }
end
```

These rules are sufficient for many but not all Sudoku puzzles. To solve very difficult grids, the rule set requires more complex rules. (Ultimately, some puzzles can be solved only by trial and error.)

9.9. CONWAY'S GAME OF LIFE EXAMPLE DECISIONS (RULEFLOW GROUPS AND GUI INTEGRATION)

The Conway's Game of Life example decision set, based on the famous cellular automaton by John Conway, demonstrates how to use ruleflow groups in rules to control rule execution. The example also demonstrates how to integrate Red Hat Decision Manager rules with a graphical user interface (GUI), in this case a Swing-based implementation of Conway's Game of Life.

The following is an overview of the Conway's Game of Life (Conway) example:

- Name: **conway**
- Main classes: **org.drools.examples.conway.ConwayRuleFlowGroupRun**, **org.drools.examples.conway.ConwayAgendaGroupRun** (in **src/main/java**)
- Module: **droolsjbpm-integration-examples**
- **Type**: Java application
- Rule files: **org.drools.examples.conway.*.drl** (in **src/main/resources**)
- **Objective**: Demonstrates ruleflow groups and GUI integration

NOTE

The Conway's Game of Life example is separate from most of the other example decision sets in Red Hat Decision Manager and is located in **~/rhdm-7.5.1-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples** of the **Red Hat Decision Manager 7.5.1 Source Distribution**from the Red Hat Customer Portal.

In Conway's Game of Life, a user interacts with the game by creating an initial configuration or an advanced pattern with defined properties and then observing how the initial state evolves. The objective of the game is to show the development of a population, generation by generation. Each generation results from the preceding one, based on the simultaneous evaluation of all cells.

The following basic rules govern what the next generation looks like:

- If a live cell has fewer than two live neighbors, it dies of loneliness.

- If a live cell has more than three live neighbors, it dies from overcrowding.
- If a dead cell has exactly three live neighbors, it comes to life.

Any cell that does not meet any of those criteria is left as is for the next generation.

The Conway's Game of Life example uses Red Hat Decision Manager rules with **ruleflow-group** attributes to define the pattern implemented in the game. The example also contains a version of the decision set that achieves the same behavior using agenda groups. Agenda groups enable you to partition the decision engine agenda to provide execution control over groups of rules. By default, all rules are in the agenda group **MAIN**. You can use the **agenda-group** attribute to specify a different agenda group for the rule.

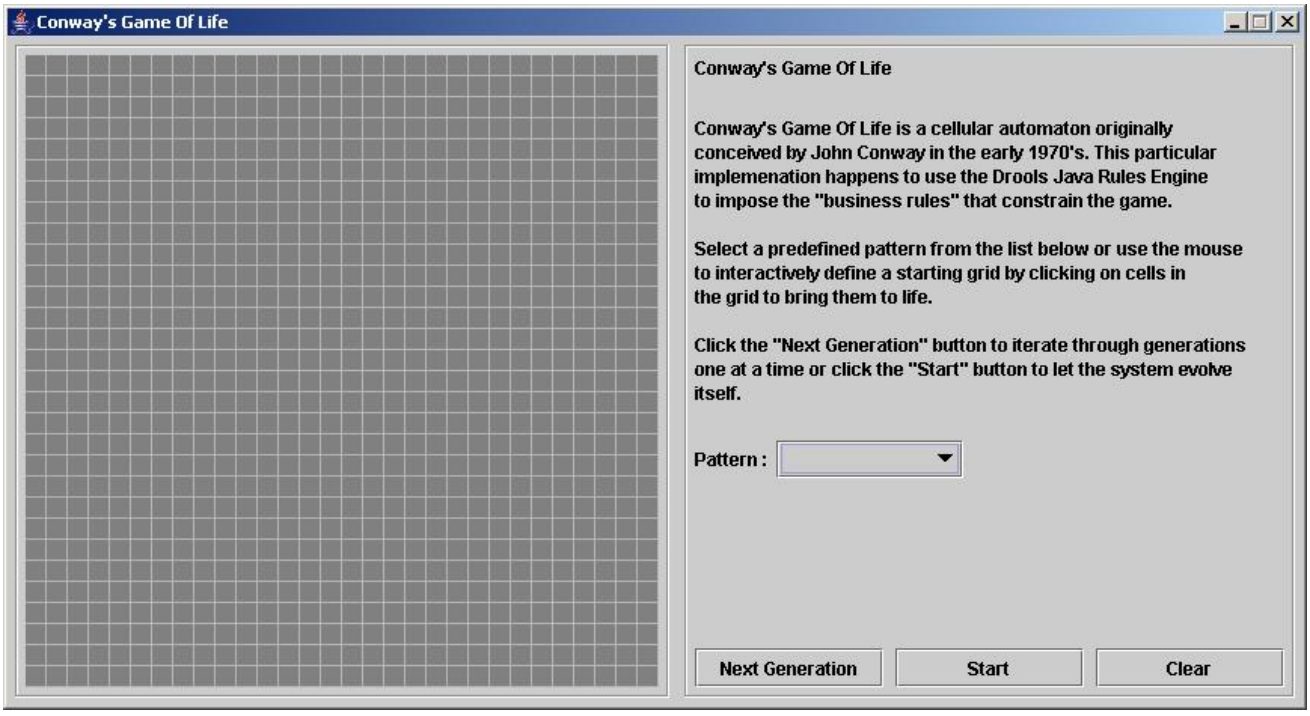
This overview does not explore the version of the Conway example using agenda groups. For more information about agenda groups, see the Red Hat Decision Manager example decision sets that specifically address agenda groups.

Conway example execution and interaction

Similar to other Red Hat Decision Manager decision examples, you execute the Conway ruleflow example by running the **org.drools.examples.conway.ConwayRuleFlowGroupRun** class as a Java application in your IDE.

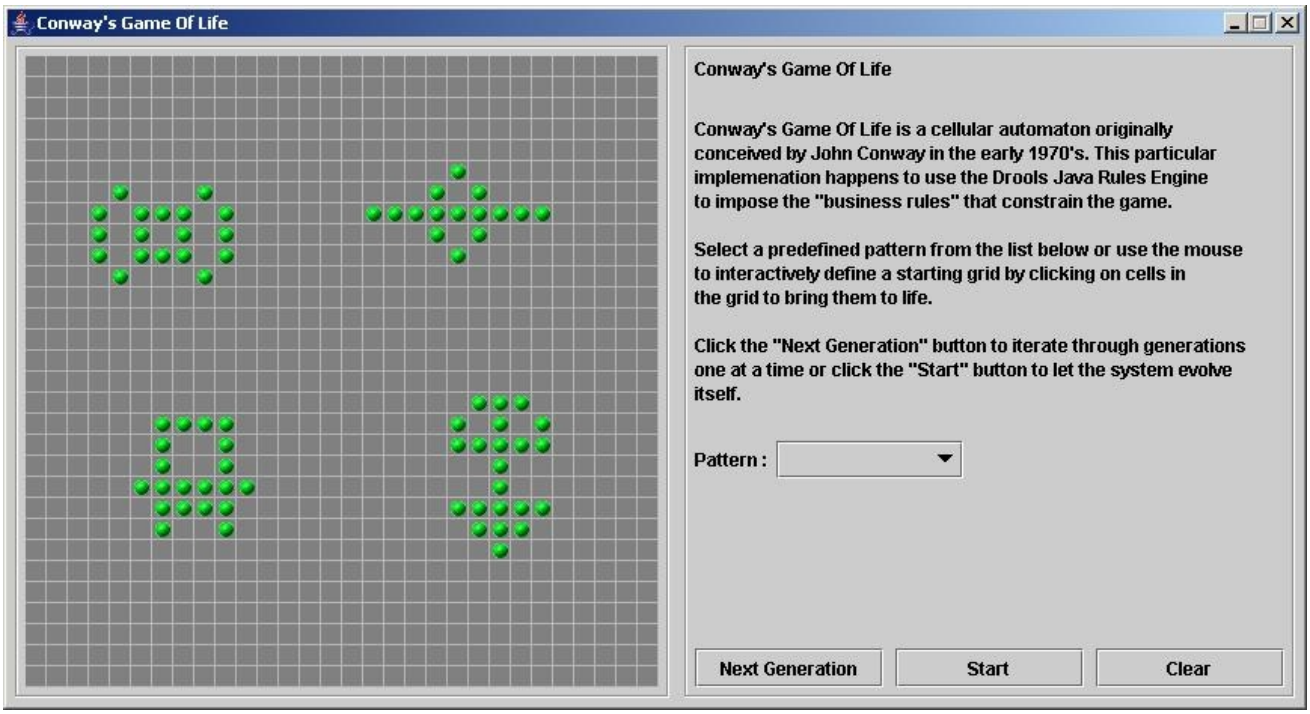
When you execute the Conway example, the **Conway's Game of Life** GUI window appears. This window contains an empty grid, or "arena" where the life simulation takes place. Initially the grid is empty because no live cells are in the system yet.

Figure 9.24. Conway example GUI after launch



Select a predefined pattern from the **Pattern** drop-down menu and click **Next Generation** to click through each population generation. Each cell is either alive or dead, where live cells contain a green ball. As the population evolves from the initial pattern, cells live or die relative to neighboring cells, according to the rules of the game.

Figure 9.25. Generation evolution in Conway example



Neighbors include not only cells to the left, right, top, and bottom but also cells that are connected diagonally, so that each cell has a total of eight neighbors. Exceptions are the corner cells, which have only three neighbors, and the cells along the four borders, with five neighbors each.

You can manually intervene to create or kill cells by clicking the cell.

To run through an evolution automatically from the initial pattern, click **Start**.

Conway example rules with ruleflow groups

The rules in the **ConwayRuleFlowGroupRun** example use ruleflow groups to control rule execution. A ruleflow group is a group of rules associated by the **ruleflow-group** rule attribute. These rules can only fire when the group is activated. The group itself can only become active when the elaboration of the ruleflow diagram reaches the node representing the group.

The Conway example uses the following ruleflow groups for rules:

- **"register neighbor"**
- **"evaluate"**
- **"calculate"**
- **"reset calculate"**
- **"birth"**
- **"kill"**
- **"kill all"**

All of the **Cell** objects are inserted into the KIE session and the **"register ..."** rules in the ruleflow group **"register neighbor"** are allowed to execute by the ruleflow process. This group of four rules creates **Neighbor** relations between some cell and its northeastern, northern, northwestern, and western neighbors.

This relation is bidirectional and handles the other four directions. Border cells do not require any special treatment. These cells are not paired with neighboring cells where there is not any.

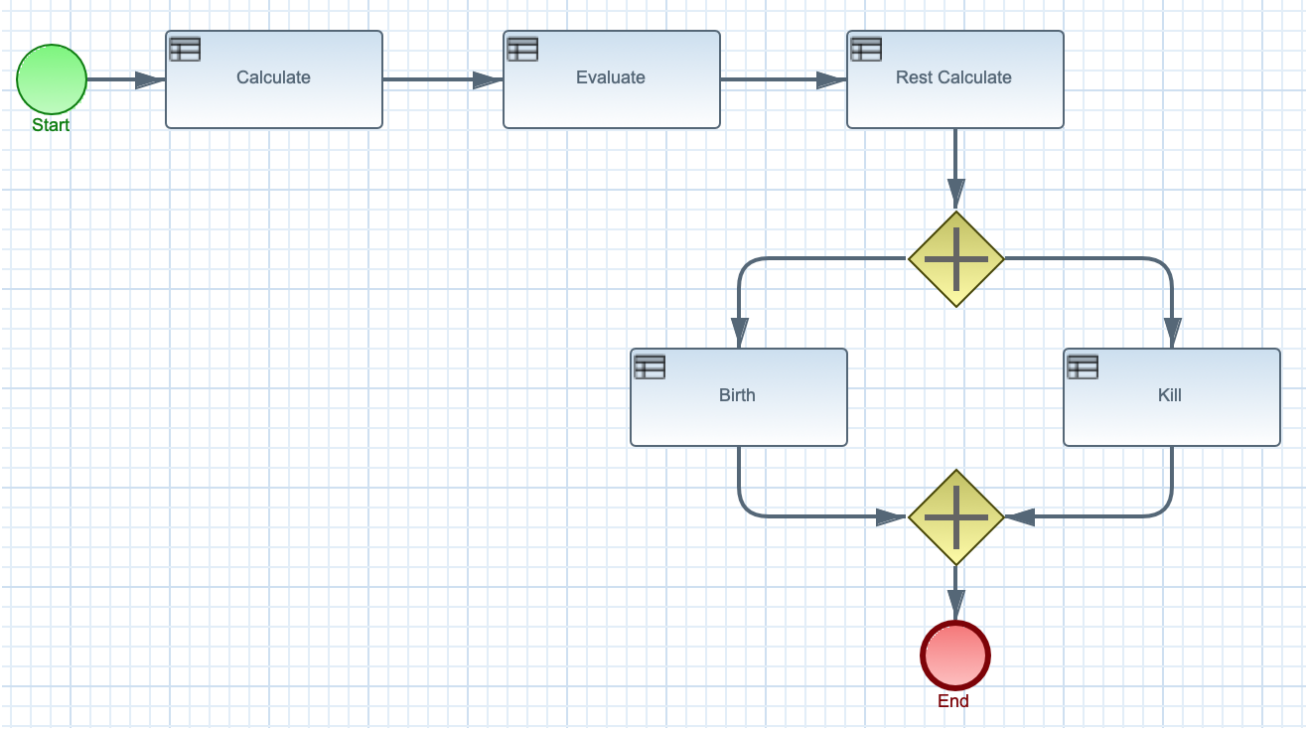
By the time all activations have fired for these rules, all cells are related to all their neighboring cells.

Rules "register ..."

```
rule "register north east"
    ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $northEast : Cell( row  == ($row - 1), col == ( $col + 1 ) )
  then
    insert( new Neighbor( $cell, $northEast ) );
    insert( new Neighbor( $northEast, $cell ) );
end

rule "register north"
    ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $north : Cell( row  == ($row - 1), col == $col )
  then
    insert( new Neighbor( $cell, $north ) );
    insert( new Neighbor( $north, $cell ) );
end

rule "register north west"
    ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $northWest : Cell( row  == ($row - 1), col == ( $col - 1 ) )
  then
    insert( new Neighbor( $cell, $northWest ) );
    insert( new Neighbor( $northWest, $cell ) );
end

rule "register west"
    ruleflow-group "register neighbor"
  when
    $cell: Cell( $row : row, $col : col )
    $west : Cell( row  == $row, col == ( $col - 1 ) )
  then
    insert( new Neighbor( $cell, $west ) );
    insert( new Neighbor( $west, $cell ) );
end
```

After all the cells are inserted, some Java code applies the pattern to the grid, setting certain cells to **Live**. Then, when the user clicks **Start** or **Next Generation**, the example executes the **Generation** ruleflow. This ruleflow manages all changes of cells in each generation cycle.

Figure 9.26. Generation ruleflow



The ruleflow process enters the **"evaluate"** ruleflow group and any active rules in the group can fire. The rules **"Kill the ..."** and **"Give Birth"** in this group apply the game rules to birth or kill cells. The example uses the **phase** attribute to drive the reasoning of the **Cell** object by specific groups of rules. Typically, the phase is tied to a ruleflow group in the ruleflow process definition.

Notice that the example does not change the state of any **Cell** objects at this point because it must complete the full evaluation before those changes can be applied. The example sets the cell to a **phase** that is either **Phase.KILL** or **Phase.BIRTH**, which is used later to control actions applied to the **Cell** object.

Rules "Kill the ..." and "Give Birth"

```
rule "Kill The Lonely"
    ruleflow-group "evaluate"
    no-loop
  when
    // A live cell has fewer than 2 live neighbors.
    theCell: Cell( liveNeighbors < 2, cellState == CellState.LIVE,
                   phase == Phase.EVALUATE )
  then
    modify( theCell ){
        setPhase( Phase.KILL );
    }
end

rule "Kill The Overcrowded"
    ruleflow-group "evaluate"
    no-loop
  when
    // A live cell has more than 3 live neighbors.
    theCell: Cell( liveNeighbors > 3, cellState == CellState.LIVE,
                   phase == Phase.EVALUATE )
  then
```

```
    modify( theCell ){
        setPhase( Phase.KILL );
    }
end

rule "Give Birth"
    ruleflow-group "evaluate"
    no-loop
  when
    // A dead cell has 3 live neighbors.
    theCell: Cell( liveNeighbors == 3, cellState == CellState.DEAD,
                   phase == Phase.EVALUATE )
  then
    modify( theCell ){
        theCell.setPhase( Phase.BIRTH );
    }
end
```

After all **Cell** objects in the grid have been evaluated, the example uses the **"reset calculate"** rule to clear any activations in the **"calculate"** ruleflow group. The example then enters a split in the ruleflow that enables the rules **"kill"** and **"birth"** to fire, if the ruleflow group is activated. These rules apply the state change.

Rules "reset calculate", "kill", and "birth"

```
rule "reset calculate"
    ruleflow-group "reset calculate"
  when
  then
    WorkingMemory wm = drools.getWorkingMemory();
    wm.clearRuleFlowGroup( "calculate" );
end

rule "kill"
    ruleflow-group "kill"
    no-loop
  when
    theCell: Cell( phase == Phase.KILL )
  then
    modify( theCell ){
        setCellState( CellState.DEAD ),
        setPhase( Phase.DONE );
    }
end

rule "birth"
    ruleflow-group "birth"
    no-loop
  when
    theCell: Cell( phase == Phase.BIRTH )
  then
    modify( theCell ){
        setCellState( CellState.LIVE ),
        setPhase( Phase.DONE );
    }
end
```

-

At this stage, several **Cell** objects have been modified with the state changed to either **LIVE** or **DEAD**. When a cell becomes live or dead, the example uses the **Neighbor** relation in the rules **"Calculate ..."** to iterate over all surrounding cells, increasing or decreasing the **liveNeighbor** count. Any cell that has its count changed is also set to to the **EVALUATE** phase to make sure it is included in the reasoning during the evaluation stage of the ruleflow process.

After the live count has been determined and set for all cells, the ruleflow process ends. If the user initially clicked **Start**, the decision engine restarts the ruleflow at that point. If the user initially clicked **Next Generation**, the user can request another generation.

Rules "Calculate ..."

```
rule "Calculate Live"
    ruleflow-group "calculate"
    lock-on-active
  when
    theCell: Cell( cellState == CellState.LIVE )
    Neighbor( cell == theCell, $neighbor : neighbor )
  then
    modify( $neighbor ){
        setLiveNeighbors( $neighbor.getLiveNeighbors() + 1 ),
        setPhase( Phase.EVALUATE );
    }
end

rule "Calculate Dead"
    ruleflow-group "calculate"
    lock-on-active
  when
    theCell: Cell( cellState == CellState.DEAD )
    Neighbor( cell == theCell, $neighbor : neighbor )
  then
    modify( $neighbor ){
        setLiveNeighbors( $neighbor.getLiveNeighbors() - 1 ),
        setPhase( Phase.EVALUATE );
    }
end
```

9.10. HOUSE OF DOOM EXAMPLE DECISIONS (BACKWARD CHAINING AND RECURSION)

The House of Doom example decision set demonstrates how the decision engine uses backward chaining and recursion to reach defined goals or subgoals in a hierarchical system.

The following is an overview of the House of Doom example:

- Name: **backwardchaining**
- Main class: **org.drools.examples.backwardchaining.HouseOfDoomMain** (in **src/main/java**)
- Module: **drools-examples**
- **Type**: Java application

- Rule file: **org.drools.examples.backwardchaining.BC-Example.drl** (in **src/main/resources**)
- **Objective**: Demonstrates backward chaining and recursion

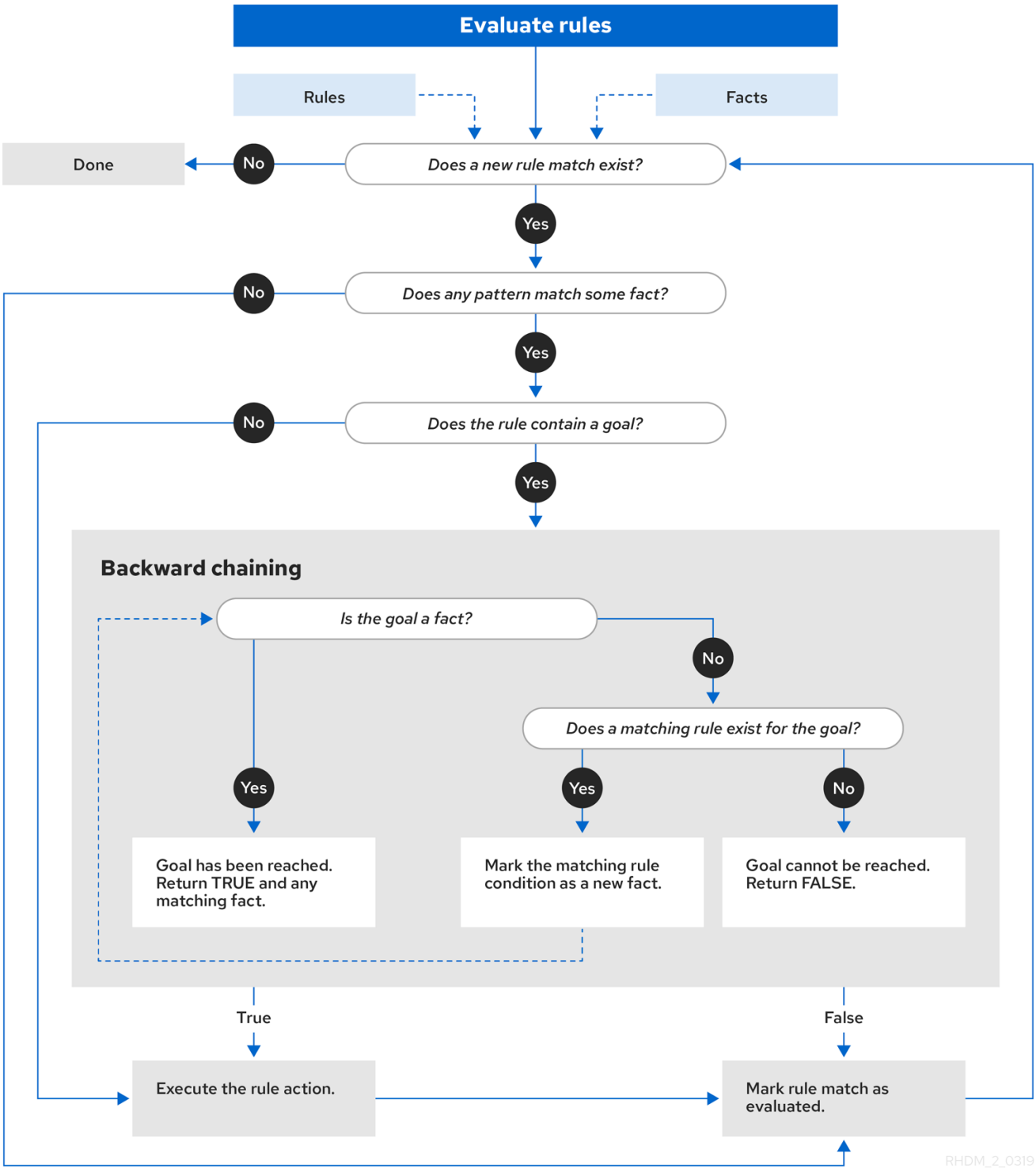
A backward-chaining rule system is a goal-driven system that starts with a conclusion that the decision engine attempts to satisfy, often using recursion. If the system cannot reach the conclusion or goal, it searches for subgoals, which are conclusions that complete part of the current goal. The system continues this process until either the initial conclusion is satisfied or all subgoals are satisfied.

In contrast, a forward-chaining rule system is a data-driven system that starts with a fact in the working memory of the decision engine and reacts to changes to that fact. When objects are inserted into working memory, any rule conditions that become true as a result of the change are scheduled for execution by the agenda.

The decision engine in Red Hat Decision Manager uses both forward and backward chaining to evaluate rules.

The following diagram illustrates how the decision engine evaluates rules using forward chaining overall with a backward-chaining segment in the logic flow:

Figure 9.27. Rule evaluation logic using forward and backward chaining



The House of Doom example uses rules with various types of queries to find the location of rooms and items within the house. The sample class **Location.java** contains the **item** and **location** elements used in the example. The sample class **HouseOfDoomMain.java** inserts the items or rooms in their respective locations in the house and executes the rules.

Items and locations in HouseOfDoomMain.java class

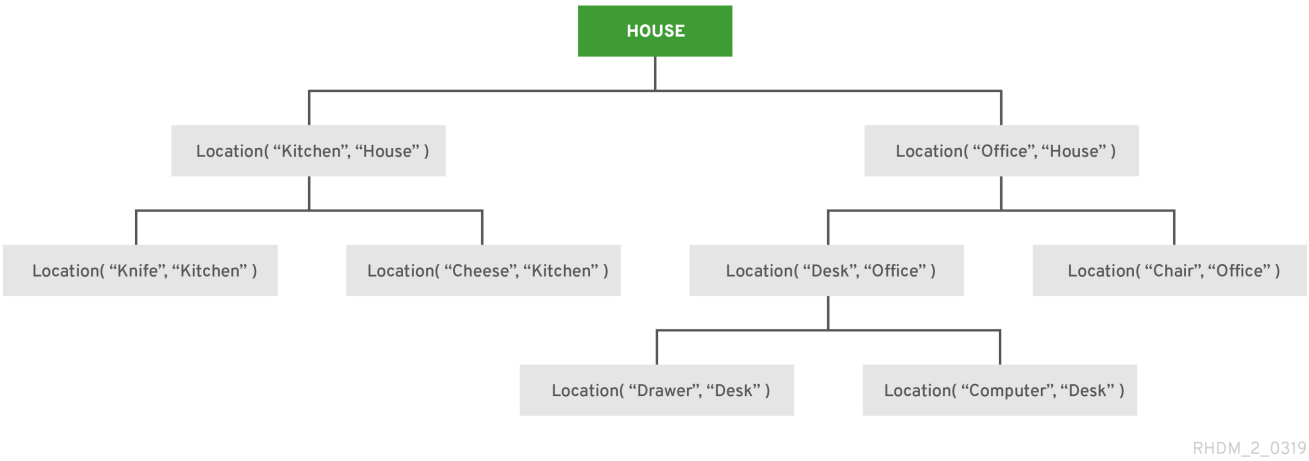
```
ksession.insert( new Location("Office", "House") );
ksession.insert( new Location("Kitchen", "House") );
ksession.insert( new Location("Knife", "Kitchen") );
ksession.insert( new Location("Cheese", "Kitchen") );
```

```
ksession.insert( new Location("Desk", "Office") );
ksession.insert( new Location("Chair", "Office") );
ksession.insert( new Location("Computer", "Desk") );
ksession.insert( new Location("Drawer", "Desk") );
```

The example rules rely on backward chaining and recursion to determine the location of all items and rooms in the house structure.

The following diagram illustrates the structure of the House of Doom and the items and rooms within it:

Figure 9.28. House of Doom structure



To execute the example, run the **org.drools.examples.backwardchaining.HouseOfDoomMain** class as a Java application in your IDE.

After the execution, the following output appears in the IDE console window:

Execution output in the IDE console

```
go1
Office is in the House
---
go2
Drawer is in the House
---
go3
---
Key is in the Office
---
go4
Chair is in the Office
Desk is in the Office
Key is in the Office
Computer is in the Office
Drawer is in the Office
---
go5
Chair is in Office
Desk is in Office
Drawer is in Desk
Key is in Drawer
```

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

All rules in the example have fired to detect the location of all items in the house and to print the location of each in the output.

Recursive query and related rules

A recursive query repeatedly searches through the hierarchy of a data structure for relationships between elements.

In the House of Doom example, the **BC-Example.drl** file contains an **isContainedIn** query that most of the rules in the example use to recursively evaluate the house data structure for data inserted into the decision engine:

Recursive query in BC-Example.drl

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

The rule **"go"** prints every string inserted into the system to determine how items are implemented, and the rule **"go1"** calls the query **isContainedIn**:

Rules "go" and "go1"

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

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

The example inserts the **"go1"** string into the decision engine and activates the **"go1"** rule to detect that item **Office** is in the location **House**:

Insert string and fire rules

```
ksession.insert( "go1" );
ksession.fireAllRules();
```

Rule "go1" output in the IDE console

```
go1
Office is in the House
```

Transitive closure rule

Transitive closure is a relationship between an element contained in a parent element that is multiple levels higher in a hierarchical structure.

The rule **"go2"** identifies the transitive closure relationship of the **Drawer** and the **House**: The **Drawer** is in the **Desk** in the **Office** in the **House**.

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

The example inserts the **"go2"** string into the decision engine and activates the **"go2"** rule to detect that item **Drawer** is ultimately within the location **House**:

Insert string and fire rules

```
ksession.insert( "go2" );
ksession.fireAllRules();
```

Rule "go2" output in the IDE console

```
go2
Drawer is in the House
```

The decision engine determines this outcome based on the following logic:

1. The query recursively searches through several levels in the house to detect the transitive closure between **Drawer** and **House**.
2. Instead of using **Location(x, y;)**, the query uses the value of **(z, y;)** because **Drawer** is not directly in **House**.
3. The **z** argument is currently unbound, which means it has no value and returns everything that is in the argument.
4. The **y** argument is currently bound to **House**, so **z** returns **Office** and **Kitchen**.

5. The query gathers information from the **Office** and checks recursively if the **Drawer** is in the **Office**. The query line **isContainedIn(x, z;)** is called for these parameters.

6. No instance of **Drawer** exists directly in **Office**, so no match is found.

7. With **z** unbound, the query returns data within the **Office** and determines that **z == Desk**.

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

8. The **isContainedIn** query recursively searches three times, and on the third time, the query detects an instance of **Drawer** in **Desk**.

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

9. After this match on the first location, the query recursively searches back up the structure to determine that the **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 the rule is satisfied.

Reactive query rule

A reactive query searches through the hierarchy of a data structure for relationships between elements and is dynamically updated when elements in the structure are modified.

The rule **"go3"** functions as a reactive query that detects if a new item **Key** ever becomes present in the **Office** by transitive closure: A **Key** in the **Drawer** in the **Office**.

Rule "go3"

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

The example inserts the **"go3"** string into the decision engine and activates the **"go3"** rule. Initially, this rule is not satisfied because no item **Key** exists in the house structure, so the rule produces no output.

Insert string and fire rules

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

Rule "go3" output in the IDE console (unsatisfied)

```
go3
```

The example then inserts a new item **Key** in the location **Drawer**, which is in **Office**. This change satisfies the transitive closure in the **"go3"** rule and the output is populated accordingly.

Insert new item location and fire rules

```
ksession.insert( new Location("Key", "Drawer") );
ksession.fireAllRules();
```

Rule "go3" output in the IDE console (satisfied)

```
Key is in the Office
```

This change also adds another level in the structure that the query includes in subsequent recursive searches.

Queries with unbound arguments in rules

A query with one or more unbound arguments returns all undefined (unbound) items within a defined (bound) argument of the query. If all arguments in a query are unbound, then the query returns all items within the scope of the query.

The rule **"go4"** uses an unbound argument **thing** to search for all items within the bound argument **Office**, instead of using a bound argument to search for a specific item in the **Office**:

Rule "go4"

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

The example inserts the **"go4"** string into the decision engine and activates the **"go4"** rule to return all items in the **Office**:

Insert string and fire rules

```
ksession.insert( "go4" );
ksession.fireAllRules();
```

Rule "go4" output in the IDE console

```
go4
Chair is in the Office
Desk is in the Office
Key is in the Office
Computer is in the Office
Drawer is in the Office
```

The rule **"go5"** uses both unbound arguments **thing** and **location** to search for all items and their locations in the entire **House** data structure:

Rule "go5"

```
rule "go5"
  when
    String( this == "go5" )
```

```
    isContainedIn(thing, location; )
  then
    System.out.println(thing + " is in " + location );
end
```

The example inserts the **"go5"** string into the decision engine and activates the **"go5"** rule to return all items and their locations in the **House** data structure:

Insert string and fire rules

```
ksession.insert( "go5" );
ksession.fireAllRules();
```

Rule "go5" output in the IDE console

```
go5
Chair is in Office
Desk is in Office
Drawer is in Desk
Key is in Drawer
Kitchen is in House
Cheese is in Kitchen
Knife is in Kitchen
Computer is in Desk
Office is in House
Key is in Office
Drawer is in House
Computer is in House
Key is in House
Desk is in House
Chair is in House
Knife is in House
Cheese is in House
Computer is in Office
Drawer is in Office
Key is in Desk
```

CHAPTER 10. ADDITIONAL RESOURCES

- *Designing your decision management architecture for Red Hat Decision Manager*
- *Getting started with decision services*
- *Designing a decision service using DRL rules*
- *Packaging and deploying a Red Hat Decision Manager project*

APPENDIX A. VERSIONING INFORMATION

Documentation last updated on Thursday, October 31, 2019.