Red Hat Process Automation Manager 7.8

Red Hat Process Automation Manager のデシジョンエンジン

Last Updated: 2020-09-18

Red Hat Process Automation Manager 7.8 Red Hat Process Automation Manager のデシジョンエンジン

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

法律上の通知

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.

概要

本書では、お客様が Red Hat Process Automation Manager で、ビジネスルールシステムおよびデシジョンサービスを作成する際に検討すべき、Red Hat Process Automation Manager のデシジョンエンジンに関する基本的な概念および機能について説明します。

目次

前書き 4

第1章 RED HAT PROCESS AUTOMATION MANAGER のデシジョンエンジン 5

第2章 KIE セッション 6
2.1. ステートレスな KIE セッション 6
2.1.1. ステートレスな KIE セッションのグローバル変数 9
2.2. ステートフルな KIE セッション 10
2.3. KIE セッションプール 13

第3章 デシジョンエンジンにおける推論と真理維持 15
3.1. デシジョンエンジンのファクト等価モード 19

第4章 デシジョンエンジンにおける実行制御 21
4.1. ルールの顕著性 21
4.2. ルールのアジェンダグループ 22
4.3. ルールのアクティベーショングループ 23
4.4. デシジョンエンジンにおけるルール実行モードおよびスレッドの安全性 24
4.5. デシジョンエンジンにおけるファクトの伝播モード 26
4.6. アジェンダ評価フィルター 27
4.7. DRL ルールセットのルールユニット 28
4.7.1. ルールユニットのデータソース 32
4.7.2. ルールユニットの実行制御 32
4.7.3. ルールユニットのアイデンティティーの競合 37

第5章 デシジョンエンジンにおける PHREAK ルールアルゴリズム 40
5.1. PHREAK でのルール評価 40
5.1.1. 前向き連鎖と後向き連鎖を使用したルール評価 44
5.2. ルールベースの設定 45
5.3. PHREAK における順次モード 48

第6章 複合イベント処理 (CEP) 50
6.1. 複合イベント処理 (CEP) におけるイベント 51
6.2. ファクトのイベントとしての宣言 51
6.3. イベントのメタデータタグ 52
6.4. デシジョンエンジンのイベント処理モード 54
6.4.1. デシジョンエンジンのストリームモードにおける負のパターン 56
6.5. ファクトタイプに対するプロパティー変更の設定およびリスナー 57
6.6. イベントの一時オペレーター 60
6.7. デシジョンエンジンにおけるセッションクロックの実装 69
6.8. イベントストリームとエントリーポイント 70
6.8.1. ルールデータのエントリーポイントの宣言 71
6.9. 時間または長さのスライディングウィンドウ 72
6.9.1. ルールデータのスライディングタイムウィンドウを宣言 72
6.10. イベントのメモリー管理 73

第7章 デシジョンエンジンクエリーおよびライブクエリー 75

第8章 デシジョンエンジンのイベントリスナーおよびデバッグロギング 77
8.1. デシジョンエンジンでのロギングユーティリティーの設定 78

第9章 RED HAT PROCESS AUTOMATION MANAGER の IDE 向けのデシジョン例 80
9.1. IDE での RED HAT PROCESS AUTOMATION MANAGER のデシジョン例のインポートと実行 80
9.2. HELLO WORLD のデシジョン例 (基本ルールおよびデバッグ) 83

9.3. 状態のデシジョン例 (前向き連鎖および競合解決) 86
顕著性を使用した状態の例 89
アジェンダグループを使用した状態の例 92
状態の例に含まれる動的なファクト 93
9.4. フィボナッチのデシジョン例 (再帰および競合解決) 94
9.5. 価格設定のデシジョン例 (デシジョンテーブル) 100
スプレッドシートのデシジョンテーブルの設定 101
基本価格のルール 104
プロモーション割引ルール 105
9.6. ペットショップのデシジョン例 (アジェンダグループ、グローバル変数、コールバック、GUI 統合) 105
ペットショップの例でのルール実行動作 106
ペットショップのルールファイルのインポート、グローバル変数、Java 関数 108
アジェンダグループを使用したペットショップルール 109
ペットショップ例の実行 113
9.7. 誠実な政治家のデシジョン例 (真理維持および顕著性) 117
Politician および Hope クラス 118
政治家の誠実性に関するルール定義 119
実行と監査証跡 120
9.8. 数独のデシジョン例 (複雑なパターン一致、コールバック、GUI 統合) 123
数独例の実行および対話 123
数独例のクラス 129
数独の検証ルール (validate.drl) 129
数独の解決ルール (sudoku.drl) 130
9.9. CONWAY の GAME OF LIFE のデシジョン例 (ルールフローグループおよび GUI 統合) 137
Conway 例の実行および対話 138
ルールグループを使用する Conway 例のルール 139
9.10. HOUSE OF DOOM のデシジョン例 (後向き連鎖および再帰) 143
再帰クエリーおよび関連のルール 147
推移閉包ルール 148
リアクティブクエリールール 149
ルールにバインドなしの引数が含まれたクエリー 150

第10章 デシジョンエンジン使用時のパフォーマンスチューニングに関する考慮点 152

第11章 関連資料 154

付録A バージョン情報 155

前書き

ビジネスルールの開発者として、Red Hat Process Automation Manager のデシジョンエンジンを理解することで、より効果的なビジネスアセットおよびよりスケーラブルなデシジョン管理アーキテクチャーの設計が可能となります。デシジョンエンジンは、Red Hat Process Automation Manager のコンポーネントで、データを保存、処理、および評価してビジネスルールを実行し、お客様が定義したデシジョンを達成します。本書では、お客様が Red Hat Process Automation Manager でビジネスルールシステムおよびデシジョンサービスを作成する際に検討すべき、デシジョンエンジンに関する基本的な概念および機能について説明します。

第1章 RED HAT PROCESS AUTOMATION MANAGER のデシジョンエンジン

デシジョンエンジンは、Red Hat Process Automation Manager のルールエンジンです。デシジョンエンジンは、データを保存、処理、および評価して、お客様が定義するビジネスルールまたはデシジョンモデルを実行します。デシジョンエンジンの基本的な機能は、受信データ、または **ファクト** をルールの条件に一致させ、そのルールを実行するかどうか、そしてどのように実行するかを決定することです。

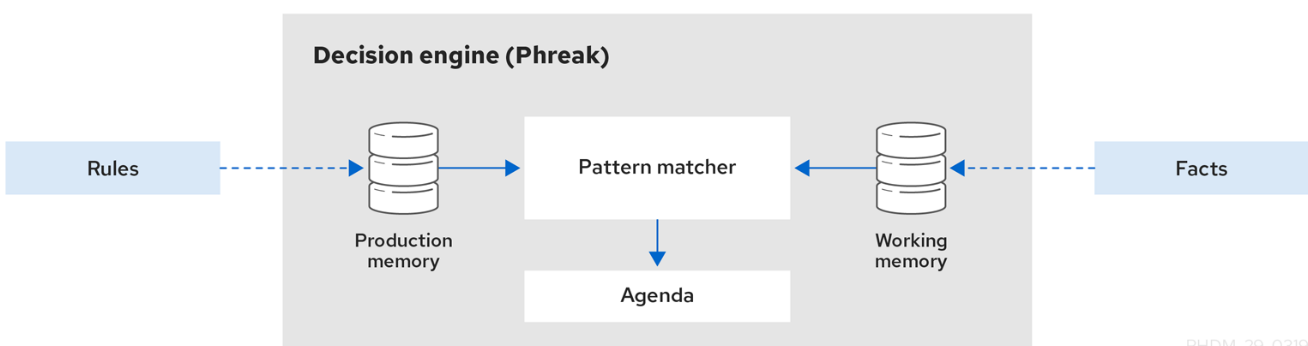
デシジョンエンジンは、以下の基本コンポーネントを使用して動作します。

- **ルール:** お客様が定義するビジネスルールまたは DMN デシジョン。すべてのルールは、ルールをトリガーする条件およびルールが指示するアクションを最小限含む必要があります。
- **ファクト:** デシジョンエンジンで入力または変更されるデータで、デシジョンエンジンがルールの条件と一致させ、適用可能なルールを実行します。
- **プロダクションメモリー:** デシジョンエンジンのルールが格納されている場所。
- **ワーキングメモリー:** デシジョンエンジンのファクトが格納されている場所。
- **アジェンダ:** 有効化されたルールが登録され、実行に備えて (必要に応じて) 並べ替えられた場所。

ビジネスユーザーまたは自動化システムが Red Hat Process Automation Manager にルール関連の情報を追加または更新した場合、その情報は 1 つ以上のファクトという形でデシジョンエンジンのワーキングメモリーに挿入されます。デシジョンエンジンは、それらのファクトをプロダクションメモリーに格納されたルールの条件と一致させ、適用可能なルールの実行を決定します。(ファクトをルールに一致させるこの処理は、**パターン一致** と呼ばれることが多いです。) ルールの条件が一致すると、デシジョンエンジンはアジェンダのルールを有効化して登録します。続いてアジェンダでは、デシジョンエンジンが実行に備えて優先的なルールまたは競合するルールをソートします。

以下の図は、デシジョンエンジンのこれらの基本コンポーネントを表しています。

図1.1 基本となるデシジョンエンジンコンポーネントの概要



デシジョンエンジンのルールやファクトの動作に関する詳細や例については、*3章デシジョンエンジンにおける推論と真理維持* を参照してください。

これらのコアなコンセプトにより、デシジョンエンジンの他のより高度なコンポーネント、処理、およびサブ処理についての理解が深まるようになります。その結果、Red Hat Process Automation Manager でより効果的なビジネスアセットを設計できるようになります。

第2章 KIE セッション

Red Hat Process Automation Manager では、KIE セッションはランタイムデータを保存して実行します。KIE セッションは KIE ベースから作成されるか、またはプロジェクトの KIE モジュール記述子ファイル (**kmodule.xml**) で KIE セッションを定義している場合は、KIE コンテナーから直接作成されます。

kmodule.xml ファイルの KIE セッション設定例

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

KIE ベースは、プロジェクトの KIE モジュール記述子ファイル (**kmodule.xml**) で定義するリポジトリーで、Red Hat Process Automation Manager のすべてのルール、処理、およびその他のビジネスアセットが含まれていますが、ランタイムのデータは一切含まれていません。

kmodule.xml ファイルの KIE ベース設定例

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

KIE セッションは、ステートレスでも、ステートフルでも可能です。ステートレスな KIE セッションでは、KIE セッションの以前の呼び出し (以前のセッション状態) からのデータは、次のセッションの呼び出しまでに破棄されます。ステートフルな KIE セッションでは、そのデータは保持されます。使用する KIE セッションのタイプは、プロジェクトの要件と、さまざまなアセット呼び出しからのデータをどのように維持するかにより決まります。

2.1. ステートレスな KIE セッション

ステートレスな KIE セッションは、推論を使用せずに、時間の経過とともにファクトを繰り返し変更していくセッションです。ステートレスな KIE セッションでは、KIE セッションの以前の呼び出し (以前のセッション状態) からのデータはセッションの呼び出し間で破棄されますが、ステートフルな KIE セッションではそのデータは保持されます。ステートレスな KIE セッションは、生成する結果が KIE ベースのコンテンツと、特定の時点で実行するために KIE セッションに渡されるデータによって決定されるという点で、関数と同様に動作します。KIE セッションには、以前に KIE セッションに渡されたデータのメモリーはありません。

以下のユースケースで、ステートレスな KIE セッションは一般的に使用されます

- **検証**、住宅ローンの対象となるかを検証するなど
- **計算**、住宅ローンのプレミアムの計算など

- **ルーティングとフィルタリング**、受信した電子メールをフォルダーにソートしたり、受信した電子メールを送信先に送信したりすることなど

たとえば、以下の運転免許のデータモデルと DRL ルールのサンプルをご覧ください。

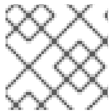
運転免許申請のデータモデル

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

運転免許申請の DRL ルールのサンプル

```
package com.company.license

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

Is of valid age ルールは、18歳未満の申請者全員を失格にします。 **Applicant** オブジェクトがデシジョンエンジンに挿入されると、デシジョンエンジンは各ルールの制約を評価し、一致するものを探します。**"objectType"** 制約は常に暗黙的で、その後に明示的なフィールド制約の任意の数が評価されます。変数 **$a** は、ルール結果で一致したオブジェクトを参照するバインディング変数です。



注記

ドル記号 (**$**) はオプションで、変数名とフィールド名を識別する上で役立ちます。

この例では、ルールのサンプルと Red Hat Process Automation Manager プロジェクトの ~/**resources** フォルダー内の他のすべてのファイルは、以下のコードで構築されます。

KIE コンテナーの作成

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

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

このコードは、クラスパスで見つかったすべてのルールファイルをコンパイルし、このコンパイルの結果である **KieModule** オブジェクトを **KieContainer** に追加します。

最後に、**StatelessKieSession** オブジェクトが **KieContainer** からインスタンス化され、指定したデータに対して実行されます。

ステートレスな KIE セッションをインスタンス化し、データを入力

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

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

assertTrue(applicant.isValid());

ksession.execute(applicant);

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

ステートレスな KIE セッションの設定では、**execute()** の呼び出しは **KieSession** オブジェクトをインスタンス化するコンビネーションメソッドとして機能し、すべてのユーザーデータを追加してユーザーコマンドを実行し、**fireAllRules()** を呼び出してから、**dispose()** を呼び出します。したがって、ステートレスな KIE セッションでは、ステートフルな KIE セッションの時のようにセッションの呼び出し後に **fireAllRules()** または **dispose()** を呼び出す必要はありません。

この場合、指定された申請者は18歳未満であるため、申請は拒否されます。

より複雑なユースケースについては、以下の例を参照してください。この例では、ステートレスな KIE セッションを使用し、コレクションなどの反復可能なオブジェクトのリストに対してルールを実行します。

運転免許申請の拡張データモデル

```
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
}
```

運転免許申請の拡張 DRL ルールセット

```
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
```

ステートレスな KIE セッションで実行が反復可能な拡張 Java ソース

```
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. **Arrays.asList()** メソッドによって生成されたオブジェクトの反復可能なコレクションに対してルールを実行するメソッド。すべてのコレクション要素は、一致したルールが実行される前に挿入されます。**execute(Object object)** および **execute(Iterable objects)** メソッドは、**BatchExecutor** インターフェースから派生した **execute(Command command)** メソッドを包むラッパーです。

2. **CommandFactory** インターフェースを使用したオブジェクトの反復可能なコレクションの実行。

3. 多くのさまざまなコマンドまたは結果出力識別子と作業するための **BatchExecutor** および **CommandFactory** 設定。**CommandFactory** インターフェースは、**StartProcess**、**Query**、および **SetGlobal** など、**BatchExecutor** で使用できる他のコマンドをサポートしています。

2.1.1. ステートレスな KIE セッションのグローバル変数

StatelessKieSession オブジェクトは、セッションスコープのグローバル、デリゲートグローバル、または実行スコープのグローバルとして解決されるように設定できるグローバル変数 (グローバル) をサポートしています。

- **セッションスコープのグローバル:** セッションスコープのグローバルの場合、メソッド **getGlobals()** を使用して、KIEセッショングローバルへのアクセスを提供する **Globals** インスタンスを返すことができます。これらのグローバルは、すべての実行呼び出しに使用されます。実行呼び出しは異なるスレッドで同時に実行される可能性があるため、可変グローバルでは注意してください。

  セッションスコープのグローバル

  ```
  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);
  ```

- **デリゲートグローバル:** デリゲートグローバルの場合、識別子を値にマップする内部コレクションに値を保存できるように、(**setGlobal(String, Object)** を使用して) 値をグローバルに割り当てることができます。この内部コレクションの識別子は、すべての提供されるデリゲートの中で優先されます。この内部コレクションで識別子が見つからない場合に、デリゲートグローバルが (もしあれば) 使用されます。

- **実行スコープのグローバル:** 実行スコープのグローバルの場合、**Command** オブジェクトを使用して、実行特有のグローバル解決用に **CommandExecutor** インターフェースに渡されるグローバルを設定できます。

CommandExecutor インターフェースでは、グローバル、挿入されたファクト、およびクエリー結果の out identifier を使用してデータをエクスポートすることもできます。

グローバル、挿入されたファクト、およびクエリー結果の out identifier

```
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. ステートフルな KIE セッション

ステートフルな KIE セッションは、推論を使用して、時間の経過とともにファクトを繰り返し変更していくセッションです。ステートフルな KIE セッションでは、KIE セッションの以前の呼び出し (以前のセッション状態) からのデータは、セッションの呼び出し間で保持されますが、ステートレスな KIE セッションではそのデータは破棄されます。

警告

ステートフルな KIE セッションの実行後に **dispose()** メソッドを呼び出して、セッションの呼び出し間でメモリーリークが発生しないようにしてください。

以下のユースケースで、ステートフルな KIE セッションは一般的に使用されます。

- **監視**、株式市場の監視や購入プロセスの自動化など

- **診断**、不具合検出プロセスまたは医療診断プロセスの実行など

- **ロジスティックス**、荷物の追跡や配達のプロビジョニングなど
- **コンプライアンスの確保**、市場取引における合法性の検証など

たとえば、以下の火災報知機のデータモデルと DRL ルールのサンプルをご覧ください。

スプリンクラーと火災報知機のデータモデル

```
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 { }
```

スプリンクラーとアラームを有効にするための DRL ルールセットのサンプル

```
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
```

When there is a fire turn on the sprinkler ルールの場合、火災が発生すると、その部屋に対する **Fire** クラスのインスタンスが作成され、KIE セッションに挿入されます。このルールは、**Fire** インスタンスに一致する特定の **room** に制約を追加し、その部屋のスプリンクラーのみがチェックされるようにします。このルールが実行されると、スプリンクラーが有効になります。他のルールのサンプルは、これに基づいてアラームをいつ有効または無効にするかを決定します。

ステートレスな KIE セッションは、標準的な Java 構文に依存してフィールドを変更しますが、ステートフルな KIEセッションはルールの **modify** ステートメントに依存して、変更をデシジョンエンジンに通知します。次に、デシジョンエンジンは変更を判断し、後続のルール実行への影響を評価します。このプロセスは、**推論** および **真理維持** を使用するデシジョンエンジンの機能の一部であり、ステートフルな KIE セッションでは不可欠となります。

この例では、ルールのサンプルと Red Hat Process Automation Manager プロジェクトの **~/resources** フォルダー内の他のすべてのファイルは、以下のコードで構築されます。

KIE コンテナーの作成

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

このコードは、クラスパスで見つかったすべてのルールファイルをコンパイルし、このコンパイルの結果である **KieModule** オブジェクトを **KieContainer** に追加します。

最後に、**KieSession** オブジェクトが **KieContainer** からインスタンス化され、指定したデータに対して実行されます。

ステートフルな KIE セッションをインスタンス化してデータを入力

```
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();
```

コンソールの出力

```
> Everything is ok
```

データが追加されると、デシジョンエンジンはすべてのパターン一致を完了しますが、ルールは実行されていないため、設定済みの検証メッセージが表示されます。新しいデータがルール条件をトリガーすると、デシジョンエンジンはルールを実行してアラームを有効にし、後で有効になったアラームをキャンセルします。

新しいデータを入力してルールをトリガー

```
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();
```

コンソールの出力

```
> 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();
```

コンソールの出力

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

この場合、返された **FactHandle** オブジェクトの参照が保持されます。ファクトハンドルは、挿入されたインスタンスへの内部エンジン参照であり、後でインスタンスを撤回または変更できるようにします。

この例が示すように、以前のステートフルな KIE セッションのデータと結果 (有効化されたアラーム) は、後続のセッションの呼び出し (アラームのキャンセル) に影響します。

2.3. KIE セッションプール

大量の KIE ランタイムデータと多くのシステムアクティビティーのあるユースケースでは、頻繁に KIE セッションが作成および破棄される可能性があります。頻繁な作成/破棄は必ずしも多大な時間を要するとは限りませんが、これが何百万回も繰り返されると、このプロセスがボトルネックとなり、膨大なクリーンアップ作業が必要となります。

このようなボリュームの高いケースには、多くの個別の KIE セッションの代わりに、KIE セッションプールを使用できます。KIE セッションプールを使用するには、KIE コンテナーから KIE セッションプールを取得し、プールでの KIE セッションの最初の数を定義して、そのプールから KIE セッションを通常どおりに作成します。

KIE セッションプールの例

```
// 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();
```

この例では、KIE セッションプールは 10 KIE セッションで起動しますが、必要な KIE セッション数を指定できます。この整数値は、初めにプールのみで作成された KIE セッション数です。実行中のアプリケーションで必要な場合は、プールの KIE セッション数を動的に増やすことができます。

KIE セッションプールを定義した後、次に KIE セッションを通常どおりに使用し、**dispose()** を呼び出すと、KIE セッションはリセットされ、破棄されずにプールにプッシュされます。

KIE セッションプールは通常、ステートフルな KIE セッションに適用されますが、KIE セッションプールは複数の **execute()** 呼び出しで再利用するステートレスな KIE セッションにも影響を及ぼす場合があります。KIE コンテナーから直接ステートレスな KIE セッションを作成すると、KIE セッションは引き続き、**execute()** 呼び出しごとに新規の KIE セッションを内部で作成します。反対に、KIE セッションプールからステートレスな KIE セッションを作成する場合、KIE セッションはプールが提供する特定の KIE セッションのみを内部で使用します。

KIE セッションプールの使用を終了すると、メモリーリークを回避するために **shutdown()** メソッドを呼び出すことができます。または、KIE コンテナーで **dispose()** を呼び出して、KIE コンテナーから作成されたすべてのプールをシャットダウンします。

第3章 デシジョンエンジンにおける推論と真理維持

デシジョンエンジンの基本的な機能は、データをビジネスルールに一致させ、ルールを実行するかどうか、そしてどのように実行するかを決定することです。関連データが適切なルールに確実に適用されるように、デシジョンエンジンは既存の知識に基づいて **推論** を作成し、推論された情報に基づいてアクションを実行します。

たとえば、以下の DRL ルールは、バスの乗車パスに関する方針など、大人の年齢要件を決定します。

年齢要件を定義するためのルール

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

このルールに基づいて、デシジョンエンジンはバス利用者が大人か子供かを推論し、指定されたアクション (**then** の結果) を実行します。18 歳以上のすべての人には、ワーキングメモリーに **IsAdult** のインスタンスが挿入されています。続いて、年齢とバスの乗車パスの推論されたこの関係は、以下のようなルールセグメントなどのルールで呼び出すことができます。

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

多くの場合、ルールシステムの新しいデータは他のルール実行の結果であり、この新しいデータは他のルール実行に影響を与える可能性があります。デシジョンエンジンがルール実行の結果としてデータをアサートする場合、デシジョンエンジンは真理維持を使用してアサーションを正当化し、推論された情報を他のルールに適用する時に真理を強制します。真理維持は、不一致の特定と矛盾の処理にも役立ちます。たとえば、2 つのルールが実行され、矛盾したアクションが発生した場合、デシジョンエンジンは以前に計算された結論からの仮定に基づいてアクションを選択します。

デシジョンエンジンは、記述挿入または論理挿入のいずれかを使用してファクトを挿入します。

- **記述挿入: insert()** で定義されます。記述挿入の後、通常はファクトが明示的に取り消されます。(**挿入** という用語が一般的に使用される場合は **記述挿入** を指します。)
- **論理挿入: insertLogical()** で定義されます。論理挿入の後、挿入されたファクトは、ファクトを挿入したルールの条件が true でなくなると自動的に取り消されます。論理挿入をサポートする条件がない場合、ファクトは取り消されます。論理的に挿入されたファクトは、デシジョンエンジンによって **正当化される** と見なされます。

たとえば、以下の DRL ルールのサンプルでは、ファクトの記述挿入を使用して、子供用または大人用のバスの乗車パスを発行するための年齢要件を決定します。

バスの乗車パスを発行するためのルール (記述挿入)

```
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
```

バス利用者の年齢が上がると、子供用から大人用の乗車パスへと移行するため、デシジョンエンジンでこれらのルールを維持することは簡単ではありません。別の方法としては、ファクトの論理挿入を使用して、これらのルールをバス利用者の年齢のルールと乗車パスの種類のルールに分けることができます。ファクトを論理的に挿入することで、ファクトは **when** 節の真理に依存することになります。

以下の DRL ルールは、論理挿入を使用して子供と大人の年齢要件を決定します。

子供および大人の年齢要件 (論理挿入)

```
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
```

重要

論理挿入の場合、ファクトオブジェクトは、Java 標準に従って **java.lang.Object** オブジェクトの **equals** および **hashCode** メソッドをオーバーライドする必要があります。2 つのオブジェクトが等しくなるのは、双方の **equals** メソッドが互いに **true** を返し、双方の **hashCode** メソッドが同じ値を返す場合です。詳細については、お使いの Java バージョンの Java API ドキュメントを参照してください。

ルールの条件が false の場合、ファクトは自動的に取り消されます。2 つのルールは相互に排他的であるため、この例ではこの動作が役立ちます。この例では、バス利用者が 18 歳未満の場合、ルールは **IsChild** ファクトを論理的に挿入します。利用者が 18 歳以上になると、**IsChild** ファクトが自動的に取り消され、**IsAdult** ファクトが挿入されます。

続いて、以下の DRL ルールが、子供用または大人用のバスの乗車パスを発行するかどうかを決定し、**ChildBusPass** ファクトおよび **AdultBusPass** ファクトを論理的に挿入します。このルールの設定が可能なのは、デシジョンエンジンの真理維持システムが、取り消しセットのカスケードに対する論理的挿入の連鎖をサポートしているからです。

バスの乗車パスを発行するためのルール (論理挿入)

```
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
```

バス利用者が 18 歳になると、**IsChild** ファクトとその利用者の **ChildBusPass** ファクトは取り消されます。これらの条件のセットに、18 歳になると子供用の乗車パスを返却する必要があることを示す別のルールを関連付けることができます。デシジョンエンジンが **ChildBusPass** オブジェクトを自動的に取り消すと、以下のルールが実行され、利用者にリクエストが送信されます。

バスの乗車パス利用者に新しいパスを通知するルール

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

以下のフローチャートは、記述挿入と論理挿入のライフサイクルを示しています。

図3.1 記述挿入

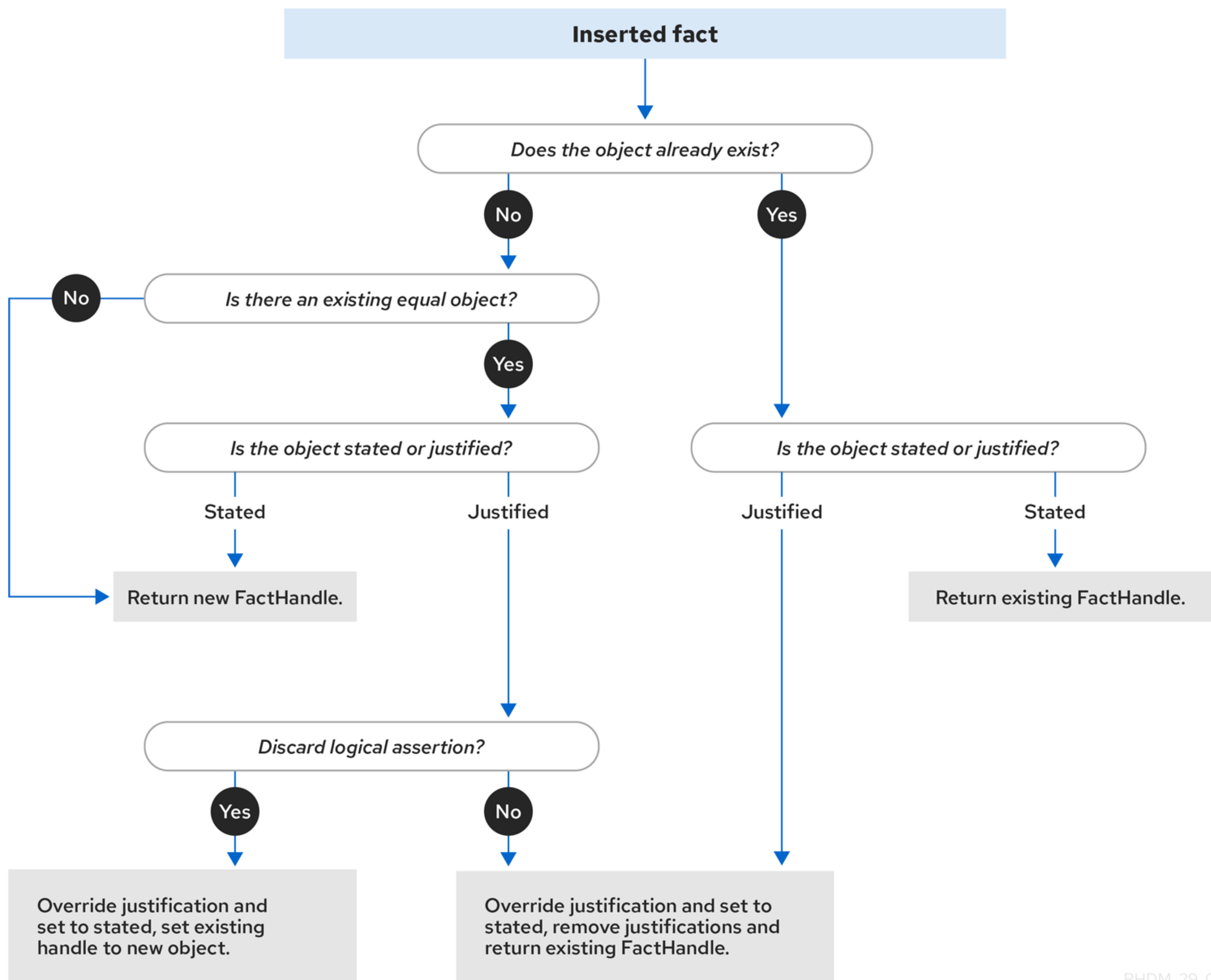
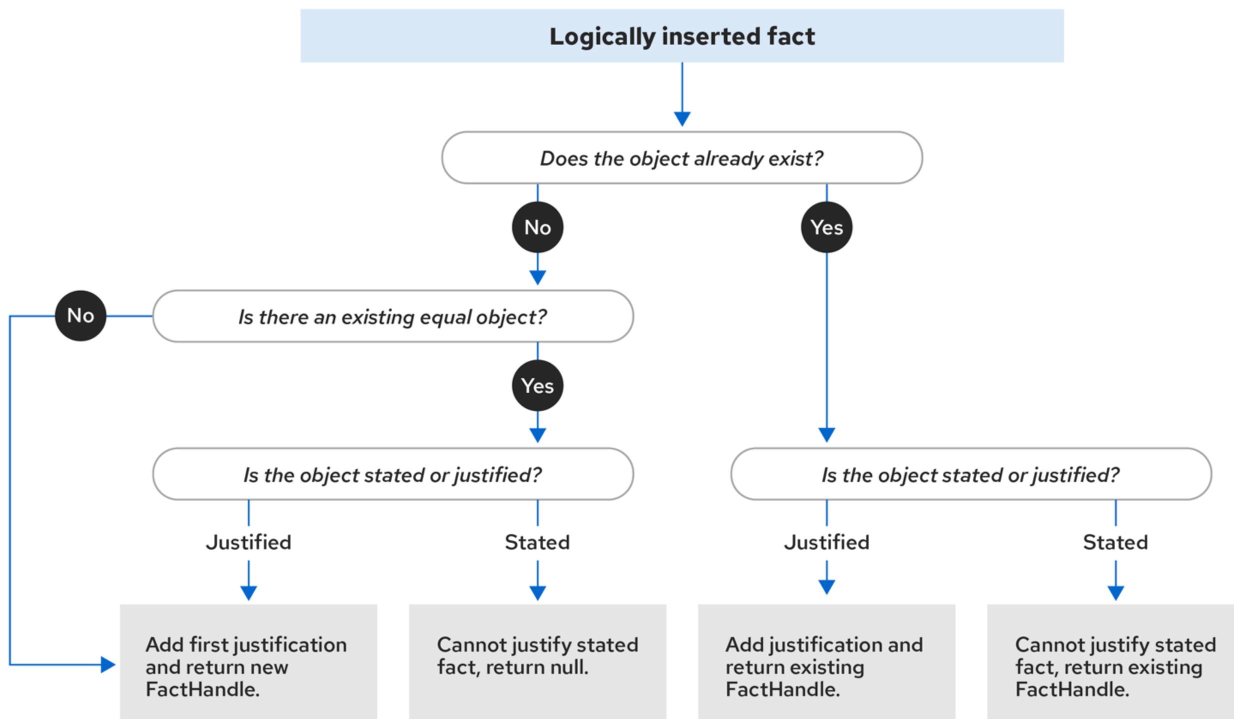


図3.2 論理挿入



ルールの実行中にデシジョンエンジンが論理的にオブジェクトを挿入すると、デシジョンエンジンはルールを実行してオブジェクトを **正当化** します。論理挿入ごとに、等しいオブジェクトは1つしか存在できず、後続の等しい論理挿入はそれぞれ、その論理挿入の正当化カウンターを増やします。ルールの条件が untrue になると、正当化は削除されます。正当化がすべてなくなると、論理オブジェクトは自動的に取り消されます。

3.1. デシジョンエンジンのファクト等価モード

デシジョンエンジンは、挿入されたファクトをデシジョンエンジンが保存および比較する方法を決める以下のファクト等価モードをサポートします。

- **identity**: (デフォルト) デシジョンエンジンは **IdentityHashMap** を使用して、すべての挿入されたファクトを保存します。新しいファクトの挿入ごとに、デシジョンエンジンは、新しい **FactHandle** オブジェクトを返します。ファクトが再度挿入されると、デシジョンエンジンはオリジナルの **FactHandle** オブジェクトを返し、同じファクトに対して繰り返される挿入を無視します。このモードでは、2 つのファクトが同じアイデンティティーを持つまったく同じオブジェクトである場合に限り、デシジョンエンジンにとってこの 2 つのファクトは同じものになります。

- **equality**: デシジョンエンジンは **HashMap** を使用して、すべての挿入されたファクトを保存します。デシジョンエンジンは、挿入されたファクトの **equals()** メソッドに従って、挿入されたファクトが既存のファクトと等しくない場合に限り、新しい **FactHandle** オブジェクトを返します。このモードでは、アイデンティティーに関係なく、2 つのファクトが同じ方法で構成される場合、デシジョンエンジンにとってこの 2 つのファクトは同じものになります。このモードは、オブジェクトを明示的なアイデンティティーではなく、機能の等価性に基づいて評価する必要がある場合に使用します。

ファクト等価モードの説明として、以下のファクトの例をご覧ください。

ファクトの例

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

identity モードの場合、ファクト **p1** および **p2** は **Person** クラスの異なるインスタンスであり、別個のアイデンティティーを持つため、別のオブジェクトとして扱われます。**equality** モードの場合、ファクト **p1** および **p2** は同じ構成を持つため、同じオブジェクトとして扱われます。このような動作の違いは、ファクトハンドルとの対話方法に影響を及ぼします。

たとえば、ファクト **p1** および **p2** をデシジョンエンジンに挿入し、その後 **p1** のファクトハンドルを取得する必要があると仮定します。**identity** モードの場合は、**p1** を指定して、正確なオブジェクトに対してファクトハンドルを返す必要があります。一方、**equality** モードの場合は、**p1**、**p2**、または **new Person("John", 45)** を指定して、ファクトハンドルを返すことができます。

ファクトを挿入して identity モードでファクトハンドルを返すコード例

```
ksession.insert(p1);

ksession.getFactHandle(p1);
```

ファクトを挿入して equality モードでファクトハンドルを返すコード例

```
ksession.insert(p1);

ksession.getFactHandle(p1);

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

ファクト等価モードを設定するには、以下のいずれかのオプションを使用します。

- システムプロパティー **drools.equalityBehavior** を **identity** (デフォルト) または **equality** に設定
- プログラムを用いて KIE ベースを作成中に等価モードを設定

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

- 特定の Red Hat Process Automation Manager プロジェクトの KIE モジュール記述子ファイル (**kmodule.xml**) で等価モードを設定

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

第4章 デシジョンエンジンにおける実行制御

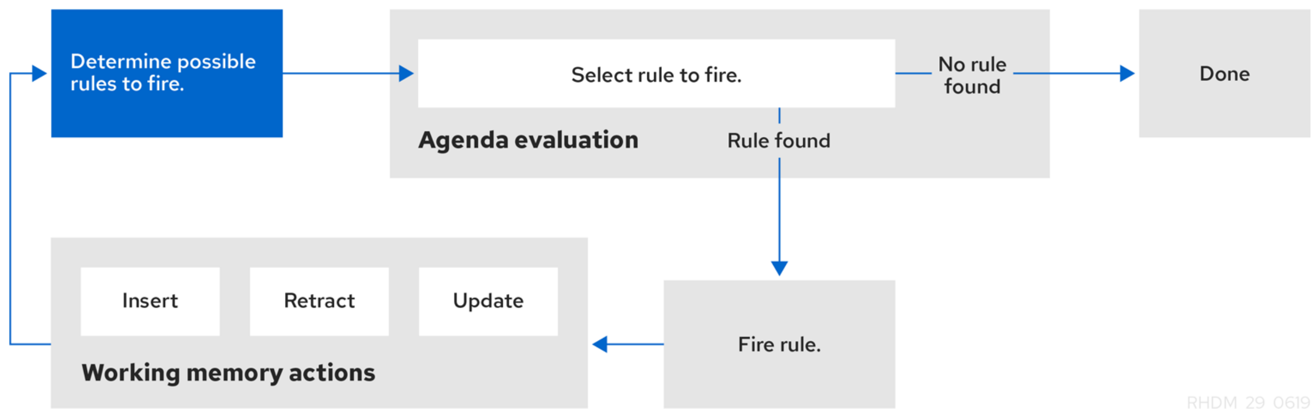
新しいルールデータが、デシジョンエンジンのワーキングメモリーに入ると、ルールが完全に一致し、実行の対象となる場合があります。単一のワーキングメモリーアクションが、複数のルール実行の対象となる可能性があります。ルールが完全に一致すると、デシジョンエンジンはアクティベーションインスタンスを作成し、ルールと一致したファクトを参照し、アクティベーションをデシジョンエンジンのアジェンダに追加します。アジェンダは、競合解決ストラテジーを使用して、これらのルールのアクティベーションが実行される順番を制御します。

Java アプリケーションで **fireAllRules()** を最初に呼び出した後、デシジョンエンジンは 2 つのフェーズを繰り返し循環します:

- **アジェンダ評価:** このフェーズでは、デシジョンエンジンは実行可能なすべてのルールを選択します。実行可能なルールが存在しない場合は、実行サイクルは終了します。実行可能なルールが見つかった場合は、デシジョンエンジンはアジェンダにアクティベーションを登録し、続いてワーキングメモリーアクションフェーズへと進み、ルール結果アクションを実行します。

- **ワーキングメモリーアクション:** このフェーズでは、デシジョンエンジンは、アジェンダに以前登録されたすべての有効化されたルールに対してルール結果アクション (各ルールの **then** 部分) を実行します。すべての結果アクションが完了するか、またはメインの Java アプリケーションプロセスが **fireAllRules()** を再度呼び出した後、デシジョンエンジンはアジェンダ評価フェーズに戻り、ルールを再評価します。

図4.1 デシジョンエンジンにおける 2 つのフェーズの実行プロセス



アジェンダに複数のルールが存在する場合、1 つのルールを実行したことが原因で、別のルールがアジェンダから削除される場合があります。これを回避するために、デシジョンエンジンでいつ、どのようにルールを実行するかを定義できます。ルール実行の順番を定義する一般的な方法には、ルールの顕著性、アジェンダグループ、アクティベーショングループ、および DRL ルールセットのルールユニットを使用する方法があります。

4.1. ルールの顕著性

各ルールには、実行の順番を決定する整数の **salience** 属性があります。ルールの顕著性の値が高いと、アクティベーションキューで順序付けされる際、優先度が高く設定されます。ルールのデフォルトの顕著性の値はゼロですが、顕著性は負の値でも正の値でもかまいません。

たとえば、以下の DRL ルールのサンプルは、以下に示す順序でデシジョンエンジンのスタックにリスト化されています。

```
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
```

RuleB ルールは 2 番目にリストされていますが、 **RuleA** ルールよりも顕著性の値が高いため、最初に実行されます。

4.2. ルールのアジェンダグループ

アジェンダグループは、同じ **agenda-group** ルールの属性によってバインドされている一連のルールです。アジェンダは、デシジョンエンジンのアジェンダのパーティションルールをグループ化します。常に 1 つのグループのみに **フォーカス** が設定され、そのルールのグループは、他のアジェンダグループのルールよりも優先して実行されます。アジェンダグループの **setFocus()** 呼び出しでフォーカスを決定します。**auto-focus** 属性を使用してルールを定義することもできます。これにより、次回ルールが有効になった時に、ルールが割り当てられているアジェンダグループ全体に自動的にフォーカスが設定されます。

Java アプリケーションで **setFocus()** 呼び出しが行われるたびに、デシジョンエンジンは指定されたアジェンダグループをルールスタックの一番上に追加します。デフォルトのアジェンダグループ **"MAIN"** には、指定されたアジェンダグループに属さないすべてのルールが含まれ、別のグループにフォーカスがない限り、スタックで最初に実行されます。

たとえば、以下の DRL ルールのサンプルは、指定されたアジェンダグループに属し、以下に示す順序でデシジョンエンジンのスタックにリスト化されています。

銀行取引アプリケーションにおける DRL ルールのサンプル

```
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
```

この例では、**"report"** アジェンダグループのルールを常に最初に実行し、**"calculation"** アジェンダグループのルールを常に 2 番目に実行する必要があります。その後に、他のアジェンダグループの残りのルールを実行できます。したがって、**"report"** および **"calculation"** グループは、他のルールが実行される前に、この順序で実行されるフォーカスを受け取る必要があります。

アジェンダグループの実行順序にフォーカスを設定

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

また、**clear()** メソッドを使用して、指定のアジェンダグループに属するルールで生成されたアクティベーションをすべて、実行する前にキャンセルすることができます。

その他すべてのルールのアクティベーションの取り消し

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

4.3. ルールのアクティベーショングループ

アクティベーショングループは、同じ **activation-group** ルールの属性によってバインドされている一連のルールです。このグループでは、実行できるルールは 1 つだけです。実行されるそのグループのルールの条件が一致すると、そのアクティベーショングループからの他のすべての保留中のルール実行がアジェンダから削除されます。

たとえば、以下の DRL ルールのサンプルは、指定されたアクティベーショングループに属し、以下に示す順序でデシジョンエンジンのスタックにリスト化されています。

銀行取引における DRL ルールのサンプル

```
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
```

この例では、**"report"** アクティベーショングループの最初のルールが実行されると、グループの 2 番目のルールとアジェンダ上の他のすべての実行可能なルールがアジェンダから削除されます。

4.4. デシジョンエンジンにおけるルール実行モードおよびスレッドの安全性

デシジョンエンジンは、デシジョンエンジンがルールを実行する方法とタイミングを決定する以下のルール実行モードをサポートします。

- **パッシブモード**: (デフォルト) デシジョンエンジンは、ユーザーまたはアプリケーションが **fireAllRules()** を明示的に呼び出す場合、ルールを評価します。デシジョンエンジンのパッシブモードは、ルールの評価および実行を直接管理する必要があるアプリケーションにとって、またはデシジョンエンジンで擬似クロック実装を使用する複合イベント処理 (CEP) のアプリケーションにとって最適です。

  パッシブモードのデシジョンエンジンを使用した CEP アプリケーションのコード例

  ```
  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();
  ```

- **アクティブモード**: ユーザーまたはアプリケーションが **fireUntilHalt()** を呼び出す場合、デシジョンエンジンはアクティブモードで開始し、ユーザーまたはアプリケーションが明示的に **halt()** を呼び出すまで、継続的にルールを評価します。デシジョンエンジンのアクティブモードは、デシジョンエンジンにルールの評価と実行の管理を委譲するアプリケーションにとって、またはデシジョンエンジンでリアルタイムクロックの実装を使用する複合イベント処理 (CEP) アプリケーションにとって最適です。

  アクティブモードのデシジョンエンジンを使用した CEP アプリケーションのコード例

  ```
  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();
```

この例では、デシジョンエンジンがルールの評価を継続している間に、現在のスレッドが永久にブロックされないように、専用の実行スレッドから **fireUntilHalt()** を呼び出します。また、専用のスレッドにより、アプリケーションコードの後の段階で **halt()** を呼び出すこともできます。

fireAllRules() および **fireUntilHalt()** の呼び出しは両方とも使用は避けるべきですが (特に異なるスレッドからの場合)、デシジョンエンジンは、スレッドの安全性の論理および内部状態マシンを使ってこのような状況を安全に処理できます。**fireAllRules()** の呼び出し中に **fireUntilHalt()** を呼び出す場合、デシジョンエンジンは、**fireAllRules()** の操作が完了し、**fireUntilHalt()** の呼び出しに対応してアクティブモードで開始するまで、パッシブモードでの実行を続けます。しかし、デシジョンエンジンが **fireUntilHalt()** の呼び出しに続いてアクティブモードで実行中に、**fireAllRules()** を呼び出すと、**fireAllRules()** の呼び出しは無視され、デシジョンエンジンは **halt()** が呼び出されるまで、アクティブモードでの実行を続けます。

アクティブモードでのスレッドの安全性を高めるため、デシジョンエンジンは、**submit()** メソッドをサポートします。このメソッドは、スレッドセーフでアトミックなアクションの KIE セッションで、操作をグループ化および実行するために使用できます。

アクティブモードでアトミック操作を実行するための submit() メソッドを使用したアプリケーションのコード例

```
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();
```

スレッドの安全性とアトミック操作は、クライアントサイドパースペクティブからも役に立ちます。たとえば、複数のファクトを指定の時間に挿入する必要があるものの、デシジョンエンジンが挿入をアトミック操作とみなし、すべての挿入が完了してからルールの再評価を始める必要がある場合などです。

4.5. デシジョンエンジンにおけるファクトの伝播モード

デシジョンエンジンは、以下のファクト伝播モードをサポートします。このモードは、ルール実行の準備としてエンジンネットワークを介して挿入されたファクトを、デシジョンエンジンが進める方法を決定します。

- **Lazy**: (デフォルト) ファクトはルール実行時にバッチコレクションで伝播されますが、ユーザーまたはアプリケーションによってファクトは個別に挿入されるので、リアルタイムでは実行されません。その結果、ファクトが最終的にデシジョンエンジンを介して伝播される順序は、ファクトが個別に挿入された順序とは異なる可能性があります。

- **Immediate**: ファクトは、ユーザーまたはアプリケーションが挿入する順序で即座に伝播されます。

- **Eager**: ファクトは (バッチコレクション内に) 遅延して伝播されますが、ルールの実行前には伝播されます。デシジョンエンジンは、**no-loop** または **lock-on-active** の属性を持つルールに対してこの伝播動作を使用します。

デフォルトでは、デシジョンエンジンの Phreak ルールアルゴリズムは、改善されたルール評価全体に対して Lazy (遅延) ファクト伝播を使用します。ただし、場合によっては、この Lazy 伝播動作は、Immediate または Eager 伝播を必要とする特定のルール実行の想定される結果を変更する可能性があります。

たとえば、以下のルールは接頭辞に **?** を指定したクエリーを使用して、プルオンリーまたはパッシブ方式でクエリーを呼び出します。

パッシブクエリーを使用したルールの例

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

rule "Rule"
  when
    $i : Integer()
```

```
    ?Q( $i; )
  then
    System.out.println( $i );
end
```

この例では、クエリーを満たす **String** が **Integer** の前に挿入される場合にのみ、ルールを実行する必要があります。たとえば、以下のコマンド例のようになります。

ルールの実行をトリガーするコマンドの例

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

ただし、Phreak におけるデフォルトの Lazy (遅延) 伝播動作が原因で、デシジョンエンジンはこの場合における 2 つのファクトの挿入シーケンスを検出しません。そのため、**String** および **Integer** の挿入順序に関係なく、このルールは実行されます。この例では、想定されるルール評価には Immediate 伝播が必要です。

デシジョンエンジンの伝播モードを変更して、この場合に想定されるルール評価を達成するには、ルールに **@Propagation(<type>)** タグを追加し、**<type>** を **LAZY**、**IMMEDIATE**、または **EAGER** に設定します。

同じルールの例では、想定どおりに、Immediate 伝播アノテーションによって、クエリーを満たす **String** が **Integer** の前に挿入される場合にのみ、ルールが評価されます。

パッシブクエリーと指定された伝播モードを使用したルールの例

```
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. アジェンダ評価フィルター

デシジョンエンジンは、アジェンダを評価している間に指定されたルールの評価を許可または拒否するために使用できるフィルターインターフェースの **AgendaFilter** オブジェクトをサポートします。**fireAllRules()** 呼び出しの一部として、アジェンダフィルターを指定することができます。

以下のコード例では、文字列 **"Test"** で終わるルールのみの評価および実行を許可します。他のルールはすべてデシジョンエンジンのアジェンダから除外されます。

アジェンダフィルター定義の例

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

4.7. DRL ルールセットのルールユニット

ルールユニットは、データソース、グローバル変数、および DRL ルールのグループで、特定の目的に向けて互いに機能し合います。ルールユニットを使用して、ルールセットを小さなユニットに分割し、それらのユニットにさまざまなデータソースをバインドしてから、個別のユニットを実行します。ルールユニットは、実行制御用のルールアジェンダグループまたはアクティブ化グループなどの、ルールをグループ化する DRL 属性に代わるものとして、強化されています。

ルールユニットは、ルールの実行を調整することで、あるルールユニットが完全に実行されると別のルールユニットの開始をトリガーする場合などに便利です。たとえば、データ強化用の一連のルール、そのデータを処理する別の一連のルール、および処理されたデータを抽出して出力する別の一連のルールがあるとします。これらのルールセットを 3 つの異なるルールユニットに追加する場合、これらのルールユニットを調整することで、1 つ目のユニットが完全に実行されると 2 つ目のユニットの開始をトリガーし、2 つ目のユニットが完全に実行されると 3 つ目のユニットの開始をトリガーすることができます。

ルールユニットを定義するには、以下の例に示すように **RuleUnit** インターフェースを実装します。

ルールユニットクラスの例

```
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.");
    }
}
```

この例では、**persons** はタイプ **Person** のファクトのソースです。ルールユニットのデータソースは、指定のルールユニットで処理されるデータのソースで、デシジョンエンジンがルールユニットの評価に使用するエントリーポイントを表します。**adultAge** グローバル変数は、このルールユニットに属するすべてのルールからアクセスできます。最後の 2 つのメソッドは、ルールユニットのライフサイクルの一部で、デシジョンエンジンによって呼び出されます。

デシジョンエンジンは、以下のようなルールユニットのオプションのライフサイクルメソッドをサポートします。

表4.1 ルールユニットのライフサイクルメソッド

メソッド	呼び出されるタイミング
onStart()	ルールユニット実行開始時
onEnd()	ルールユニット実行終了時
onSuspend()	ルールユニット実行の一時停止時 (**runUntilHalt()** でのみを使用される)
onResume()	ルールユニット実行の再開時 (**runUntilHalt()** でのみ使用される)
onYield(RuleUnit other)	ルールユニットにおけるルールの結果が異なるルールユニットの実行をトリガー

ルールユニットに、ルールを 1 つ以上追加することができます。デフォルトでは、DRL ファイルのすべてのルールは、DRL ファイル名の命名規則に従うルールユニットに自動的に関連付けられます。DRL ファイルが同じパッケージにあり、**RuleUnit** インターフェースを実装するクラスと同じ名前を持つ場合、その DRL ファイルのすべてのルールは、そのルールユニットに暗黙的に属します。たとえば、**org.mypackage.myunit** パッケージの **AdultUnit.drl** ファイルにあるすべてのルールは、自動的にルールユニット **org.mypackage.myunit.AdultUnit** の一部となります。

この命名規則をオーバーライドし、DRL ファイル内のルールが属するルールユニットを明示的に宣言するには、DRL ファイル内でキーワード **unit** を使用します。**unit** 宣言は、すぐに package 宣言に従い、DRL ファイルのルールが一部となっているパッケージ内のクラス名を含む必要があります。

DRL ファイルのルールユニット宣言の例

```
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
```

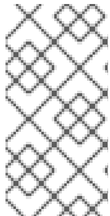
警告

同じ KIE ベースで、ルールユニットありのルールとルールユニットなしのルールを混在させないでください。KIE ベースで 2 つのルールのパラダイムを混在させると、コンパイルエラーが発生します。

以下の例のように OOPath 表記を使用して、より便利な方法で同じパターンを書き換えることもできます。

OOPath 表記を使用する DRL ファイルのルールユニット宣言の例

```
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
```



注記

OOPath は、DRL ルールの条件の制約でオブジェクトのグラフを参照するために設計された XPath のオブジェクト指向構文の拡張です。OOPath は、コレクションおよびフィルター制約を処理する間に XPath からのコンパクト表記を使用して関連要素を移動します。また、OOPath はとくにオブジェクトグラフの場合に役に立ちます。

この例では、ルール条件で一致するファクトはすべて、ルールユニットクラスの **DataSource** 定義で定義される **persons** のデータソースから取得されます。ルール条件およびアクションは、グローバル変数が DRL ファイルレベルで定義されるのと同じ方法で **adultAge** 変数を使用します。

KIE ベースに定義されたルールユニットを 1 つ以上実行するには、KIE ベースにバインドされている新規の **RuleUnitExecutor** クラスを作成し、関連するデータソースからルールユニットを作成して、ルールユニットエグゼキューターを実行します。

ルールユニット実行の例

```
// 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 );
```

ルールは **RuleUnitExecutor** クラスによって実行されます。**RuleUnitExecutor** クラスは KIE セッションを作成し、必要な **DataSource** オブジェクトをこれらのセッションに追加してから、**run()** メソッドにパラメーターとして渡される **RuleUnit** に基づいてルールを実行します。

例の実行コードは、関連する **Person** ファクトが **persons** データソースに挿入されると、以下の出力を生成します。

ルールユニット実行出力の例

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

ルールユニットインスタンスを明示的に作成するのではなく、エグゼキューターにルールユニット変数を登録し、実行するルールユニットクラスをエグゼキューターに渡すと、エグゼキューターがルールユニットのインスタンスを作成します。続いて、ルールユニットを実行する前に **DataSource** 定義および他の変数を設定できます。

登録変数を含む別のルールユニット実行オプション

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

RuleUnitExecutor.bindVariable() メソッドに渡す名前は、実行時に、同じ名前のルールユニットクラスのフィールドに変数をバインドするために使用されます。前述の例では、**RuleUnitExecutor** は、新しいルールユニットに **"persons"** の名前にバインドされているデータソースを挿入します。また、**AdultUnit** クラス内の対応する名前のフィールドに、文字列 **"adultAge"** にバインドされている値 **18** を挿入します。

このデフォルトの変数バインディング動作をオーバーライドするには、**@UnitVar** アノテーションを使用してルールユニットクラスの各フィールドに対して論理バインディング名を明示的に定義します。たとえば、以下のクラスのフィールドバインディングは、代替名で再度定義されます。

@UnitVar を使用した変数バインディング名を変更するコード例

```
package org.mypackage.myunit;

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

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

次に、これらの代替名を使用して、変数をエグゼキューターにバインドし、ルールユニットを実行できます。

変更した変数名を使用したルールユニット実行の例

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

ルールユニットは、**run()** メソッド (KIE セッションで **fireAllRules()** を呼び出す場合と同じ) を使用して **パッシブモード** で、または **runUntilHalt()** メソッド (KIE セッションで **fireUntilHalt()** を呼び出す場合

と同じ) を使用して **アクティブモード** で実行できます。デフォルトでは、デシジョンエンジンは **パッシブモード** で実行され、ユーザーまたはアプリケーションが明示的に **run()** (標準ルールでは **fireAllRules()**) を呼び出す場合にのみルールユニットを評価します。ユーザーまたはアプリケーションがルールユニットに **runUntilHalt()** (標準ルールでは **fireAllRules()**) を呼び出す場合、デシジョンエンジンは **アクティブモード** で開始し、ユーザーまたはアプリケーションが明示的に **halt()** を呼び出すまで、継続的にルールユニットを評価します。

runUntilHalt() メソッドを使用する場合は、メインスレッドをブロックしないように、別の実行スレッド上でメソッドを呼び出します。

別のスレッド上の runUntilHalt() を使用したルールユニットの実行例

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

4.7.1. ルールユニットのデータソース

ルールユニットのデータソースは、指定のルールユニットが処理したデータのソースで、デシジョンエンジンがルールユニットの評価に使用するエントリーポイントを表します。ルールユニットは、ゼロまたは複数のデータソースを持つことができ、ルールユニット内で宣言された各 **DataSource** の定義は、ルールユニットエグゼキューターへの異なるエントリーポイントに対応することができます。複数のルールユニットは、単一データソースを共有できます。ただし、各ルールユニットは、別々のエントリーポイントを使用しなければなりません。このエントリーポイントを介して同じオブジェクトが挿入されます。

以下の例で示すように、ルールユニットクラスの固定されたデータセットを使用して **DataSource** 定義を作成できます。

データソース定義の例

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

データソースはルールユニットのエントリーポイントを表すため、ルールユニットでファクトを挿入、更新、または削除できます。

ルールユニットでファクトを挿入、更新、削除するコード例

```
// 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. ルールユニットの実行制御

一方のルールユニットの実行により、もう一方のルールユニットの開始がトリガーされるようにルールの実行を調整する必要がある場合に、ルールユニットは役に立ちます。

ルールユニットの実行制御を容易にするために、デシジョンエンジンは以下のルールユニットメソッドをサポートします。このメソッドは、DRL ルールアクションで使用して、ルールユニットの実行を調整することができます。

- **drools.run()**: 指定されたルールユニットクラスの実行をトリガーします。このメソッドでは、ルールユニットの実行を命令的に中断し、他の指定されたルールユニットを有効化します。
- **drools.guard()**: 関連付けられたルール条件が満たされるまで、指定されたルールユニットクラスが実行されないようにします (保護します)。このメソッドは、他の指定されたルールユニットの実行を宣言的にスケジュールします。デシジョンエンジンが、保護ルールの条件に対して少なくとも 1 つの一致をもたらす場合、保護されたルールユニットは有効とみなされます。ルールユニットには、複数の保護ルールを含めることができます。

drools.run() メソッドの例として、それぞれが指定されたルールユニットに属す以下の DRL ルールを検討してください。**NotAdult** ルールは **drools.run(AdultUnit.class)** メソッドを使用して **AdultUnit** ルールユニットの実行をトリガーします。

drools.run() を使用した制御された実行を含む DRL ルールの例

```
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
```

この例では、これらのルールからビルドされた KIE ベースから作成された **RuleUnitExecutor** クラスと、これにバインドされている **persons** の **DataSource** 定義も使用します。

ルールエグゼキューターとデータソース定義の例

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

この例では、**RuleUnitExecutor** クラスから **DataSource** 定義を直接作成し、これを単一ステートメントで **"persons"** 変数にバインドします。

例の実行コードは、関連する **Person** ファクトが **persons** データソースに挿入されると、以下の出力を生成します。

ルールユニット実行出力の例

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

NotAdult ルールは、**"Sally"** という人物の評価時に一致を検出します。この人物は 18 歳未満です。続いてこのルールは、この人物の年齢を **18** に変更し、**drools.run(AdultUnit.class)** メソッドを使用して **AdultUnit** ルールユニットの実行をトリガーします。**AdultUnit** ルールユニットには、**DataSource** 定義の 3 人の **persons** 全員に対して実行可能となったルールが含まれています。

drools.guard() メソッドの例として、以下の **BoxOffice** クラスと **BoxOfficeUnit** ルールユニットクラスを検討してください。

BoxOffice クラスの例

```
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;
    }
}
```

BoxOfficeUnit ルールユニットクラスの例

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

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

また、この例では、以下の **TicketIssuerUnit** ルールユニットクラスを使用して、少なくとも 1 つのボックスオフィス (チケット売り場) が営業中である限り、ボックスオフィスでのイベントチケットの販売を続行します。このルールユニットは **persons** および **tickets** の **DataSource** 定義を使用します。

TicketIssuerUnit ルールユニットクラスの例

```
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;
    }
}
```

BoxOfficeUnit ルールユニットには、**BoxOfficeIsOpen** DRL ルールが含まれます。これは、**drools.guard(TicketIssuerUnit.class)** メソッドを使用して、イベントチケットを配布する **TicketIssuerUnit** ルールユニットの実行を保護します。以下に DRL ルールの例を示します。

drools.guard() を使用した制御された実行を含む DRL ルールの例

```
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
```

この例では、少なくとも 1 つのボックスオフィスが **open** である限り、保護された **TicketIssuerUnit** ルールユニットが有効なため、イベントチケットは配布されます。**open** 状態のボックスオフィスがなくなると、保護された **TicketIssuerUnit** ルールユニットは実行されなくなります。

以下のクラスの例は、より完全なボックスオフィスのシナリオを説明しています。

ボックスオフィスシナリオのクラスの例

```
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. ルールユニットのアイデンティティーの競合

保護されたルールユニットを使用したルール実行のシナリオでは、1 つのルールが複数のルールユニットを保護することができます。同時に、複数のルールが 1 つのルールユニットを保護してから有効化することもできます。このような 2 通りの保護シナリオでは、ルールユニットには、アイデンティティーの競合を避けるための明確に定義されたアイデンティティーが必要です。

デフォルトでは、ルールユニットのアイデンティティーはルールユニットクラス名で、**RuleUnitExecutor** によりシングルトンクラスとして処理されます。この識別動作は、**RuleUnit** インターフェースの **getUnitIdentity()** のデフォルトメソッドにエンコードされています。

RuleUnit インターフェースのデフォルトのアイデンティティーメソッド

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

場合によっては、ルールユニット間のアイデンティティーの競合を避けるために、このデフォルトの識別動作をオーバーライドする必要があります。

たとえば、以下の **RuleUnit** クラスには、あらゆる種類のオブジェクトを許可する **DataSource** 定義が含まれています。

Unit0 ルールユニットクラスの例

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

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

このルールユニットには、2 つの条件 (OOPath 表記)に基づいて別のルールユニットを保護する、以下の DRL ルールが含まれています。

ルールユニットの GuardAgeCheck DRL ルールの例

```
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
```

保護された **AgeCheckUnit** ルールユニットは、一連の **persons**の年齢を検証します。**AgeCheckUnit** には、確認用の **persons** の **DataSource** の定義、検証用の **minAge** 変数、および結果を集計する **List** が含まれます。

AgeCheckUnit ルールユニットの例

```
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;
    }
}
```

AgeCheckUnit ルールユニットには、データソースの **persons** の検証を実行する以下の DRL ルールが含まれます。

ルールユニットの CheckAge DRL ルールの例

```
package org.mypackage.myunit
unit AgeCheckUnit

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

この例では、**RuleUnitExecutor** クラスを作成し、これらの 2 つのルールユニットが含まれる KIE ベースにクラスをバインドして、同じルールユニットの **DataSource** 定義を 2 つ作成します。

executor 定義とデータソース定義の例

```
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 );
```

一部のオブジェクトを入力データソースに挿入し、**Unit0** ルールユニットを実行できるようになりました。

挿入されたオブジェクトを使用したルールユニット実行の例

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

実行結果一覧の例

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

この例では、**AgeCheckUnit** という名前のルールユニットはシングルトンクラスと見なされ、1 回のみ実行されます。この時、**minAge** 変数は **3** に設定されます。入力データソースに挿入された文字列 **"test"** および整数 **4** の両方は、**minAge** 変数が **4** に設定された 2 回目の実行をトリガーする可能性もあります。しかし、同じアイデンティティーを持つ別のルールユニットがすでに評価されているため、2 回目の実行はありません。

このルールユニットのアイデンティティーの競合を解決するには、**AgeCheckUnit** クラスの **getUnitIdentity()** メソッドをオーバーライドして、ルールユニットアイデンティティーに **minAge** 変数も含めます。

getUnitIdentity() メソッドをオーバーライドする変更された AgeCheckUnit ルールユニット

```
public class AgeCheckUnit implements RuleUnit {

    ...

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

このオーバーライドにより、以前のルールユニットの実行例は、以下の出力を生成します。

変更したルールユニットの実行結果一覧の例

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

minAge が **3** と **4** に設定されたルールユニットは、2 つの異なるルールユニットとみなされるようになり、両方とも実行されます。

第5章 デシジョンエンジンにおける PHREAK ルールアルゴリズム

Red Hat Process Automation Manager のデシジョンエンジンは、ルール評価に Phreak アルゴリズムを使用します。Phreak は、Rete アルゴリズムから発展したもので、これには強化された Rete アルゴリズムの ReteOO も含まれます。ReteOO は、オブジェクト指向システム向けに Red Hat Process Automation Manager の以前のバージョンに導入されました。全体として Phreak は、Rete および ReteOO よりスケーラブルで、大規模なシステムでより迅速に対応します。

Rete は Eager (即時ルール評価) でデータ指向と考えられていますが、Phreak は Lazy (遅延ルール評価) で目標指向と考えられています。Rete アルゴリズムは、すべてのルールに対して部分的な一致を見つけ出すために insert、update、および delete の各アクションを実行中に、多数のアクションを実行します。ルールの一致において Rete アルゴリズムは活発なため、最終的にルールを実行するまでに長い時間がかかります。これは大規模なシステムにおいて特に顕著です。Phreak の場合は、ルールの部分的な一致を意図的に遅延させ、大量のデータをより効率的に処理します。

Phreak アルゴリズムでは、これまでの Rete アルゴリズムに、以下に示す一連の拡張機能を追加しています:

- コンテキストメモリーの 3 つのレイヤー: ノード、セグメント、およびルールのメモリータイプ
- ルールベース、セグメントベース、およびノードベースのリンク
- Lazy (遅延) ルール評価
- 一時停止と再開を使用したスタックベースの評価
- 孤立したルール評価
- セット指向の伝播

5.1. PHREAK でのルール評価

デシジョンエンジンが開始すると、すべてのルールは、ルールのトリガーが可能なパターン一致データに **リンクされていない** と見なされます。この段階で、デシジョンエンジンの Phreak アルゴリズムはルールを評価しません。**insert**、**update**、および **delete** の各アクションはキューに入れられ、Phreak は、実行する可能性が最も高いルールに基づいてヒューリスティックを使用し、次に評価するルールを計算して選択します。すべての必要な入力値がルールに生成されると、ルールは関連するパターン一致データに **リンクされている** と見なされます。続いて Phreak は、このルールを表す目標を作成し、ルールの顕著性によって順序付けられた優先度キューにこの目標を置きます。目標が作成されたルールのみが評価され、その他の潜在的なルール評価は遅延されます。個別のルールは評価されますが、ノードの共有は引き続きセグメンテーション処理を通じて行われます。

タプル指向の Rete とは異なり、Phreak 伝播はコレクション指向です。評価されているルールの場合、デシジョンエンジンは最初のノードにアクセスし、キューに入れられたすべての insert、update、および delete の各アクションを処理します。結果は 1 つのセットに追加され、そのセットは子ノードに伝播されます。子ノードでは、キューに入れられたすべての insert、update、および delete の各アクションが処理され、その結果を同じセットに追加します。続いてそのセットは次の子ノードに伝播され、同じ処理が終了ノードに達するまで繰り返されます。このサイクルは、特定のルール構成にパフォーマンス上の利点を提供できるバッチ処理効果をもたらします。

ルールのリンクやリンク解除は、ネットワークセグメンテーションに基づいて、レイヤー化されたビットマスクシステムを使用して行われます。ルールネットワークが構築されると、同じ一連のルールで共有されるルールネットワークノードに対して、複数のセグメントが作成されます。ルールはセグメントのパスで構成されます。ルールが他のどのルールともノードを一切共有しない場合は、このルールは単一のセグメントになります。

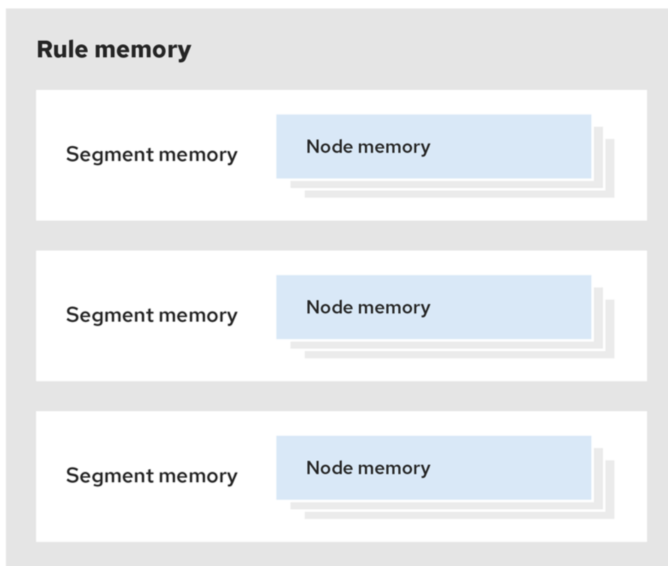
セグメントの各ノードにビットマスクの補正値が割り当てられます。別のビットマスクは、以下の要件に従って、ルールのパスの各セグメントに割り当てられます:

- ノードの入力が少なくとも 1 つ存在する場合、ノードビットは **on** 状態に設定されます。
- セグメントの各ノードのビットが **on** 状態に設定されている場合、セグメントビットも **on** 状態に設定されます。
- いずれかのノードビットが **off** 状態に設定されている場合、セグメントも **off** 状態に設定されます。
- ルールのパスの各セグメントが **on** 状態に設定されている場合、ルールはリンクされていると見なされ、目標が作成されてルールを評価するためのスケジュールが組まれます。

変更されたノード、セグメント、およびルールを追跡する際に、同じビットマスクの手法が使用されます。この追跡機能により、すでにリンクされたルールの評価目標が作成後に変更された場合、このルールの評価をスケジュールから外すことができます。その結果、部分的な一致を評価できるルールはありません。

Rete におけるメモリーの単一ユニットとは対照的に、Phreak には、ノード、セグメント、およびルールのメモリータイプから成る 3 つのレイヤーのコンテキストメモリーがあります。そのため、このルール評価の処理は Phreak で可能となります。このようなレイヤーがあることで、ルールを評価する際は、より深くコンテキストを解釈することが可能になります。

図5.1 Phreak の 3 つのレイヤーから成るメモリーシステム

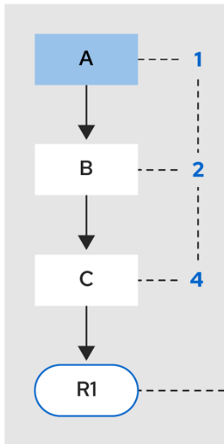


RHDM_29_0319

以下の例では、Phreak の 3 つのレイヤーから成るメモリーシステムで、ルールがどのように組織化および評価されているかについて説明しています。

例 1: 3 つのパターンがある単一のルール (R1): A、B、および C。ルールは、ノード用にビット 1、2、および 4 を持つ単一のセグメントを形成します。単一のセグメントのビットの補正値は 1 です。

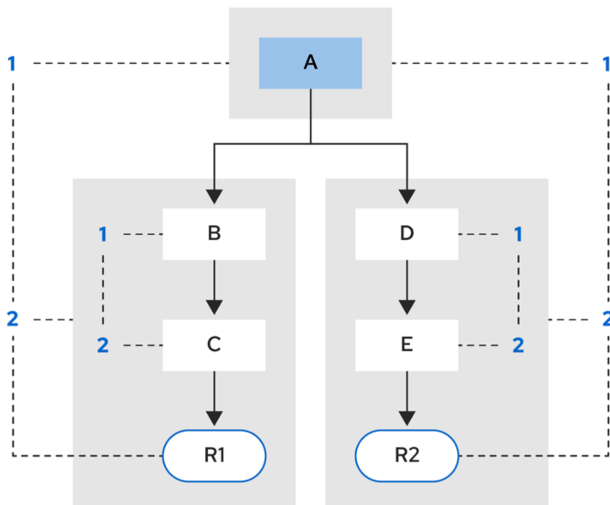
図5.2 例 1: 単一ルール



RHDM_29_0319

例 2: ルール R2 が追加され、パターン A を共有します。

図5.3 例 2: パターン共有の 2 つのルール



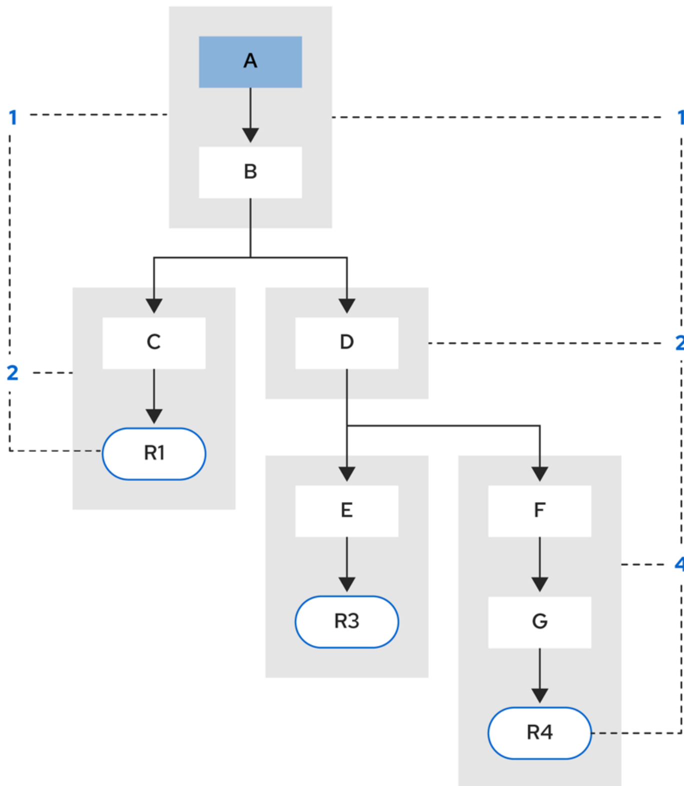
RHDM_29_0319

パターン A は独自のセグメントに置かれ、その結果、各ルールには 2 つのセグメントが作成されます。これら 2 つのセグメントは、各ルールのパスを作成します。1 つ目のセグメントは、両方のパスで共有されます。パターン A がリンクされると、セグメントはリンクされます。続いてこのセグメントは、共有される各パスでこれを繰り返します。この時ビット 1 は **on** に設定されます。パターン B および C がその後オンになると、パス R1 の 2 つ目のセグメントがリンクされ、これによりビット 2 が R1 に対してオンになります。ビット 1 およびビット 2 が R1 に対してオンになったことで、ルールがリンクされるようになり、目標が作成され、ルールを今後評価して実行するためのスケジュールが立てられます。

ルールが評価されると、セグメントによって一致の結果が共有できるようになります。各セグメントには、そのセグメントのすべての insert、update、および delete をキューに入れるステージングメモリーがあります。R1 が評価されるとルールはパターン A を処理し、これにより一連のタプルが作成されます。アルゴリズムがセグメンテーションの分割を検出し、セット内の各 insert、update、および delete にピアタプルを作成します。そして、これらを R2 のステージングメモリーに追加します。次にこれらのタプルは、既存のステージタプルとマージされ、最終的に R2 が評価されると実行されます。

例 3: ルール R3 とルール R4 が追加され、パターン A とパターン B を共有します。

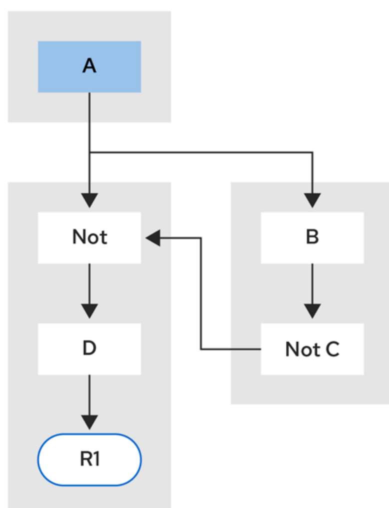
図5.4 例 3: パターン共有の 3 つのルール



ルール R3 と ルール R4 にはセグメントが 3 つあり、R1 にはセグメントが 2 つあります。R1、R3、および R4 がパターン A とパターン B を共有し、R3 と R4 がパターン D を共有しています。

例 4: パターンの共有なしでサブネットワークがある単一ルール (R1)

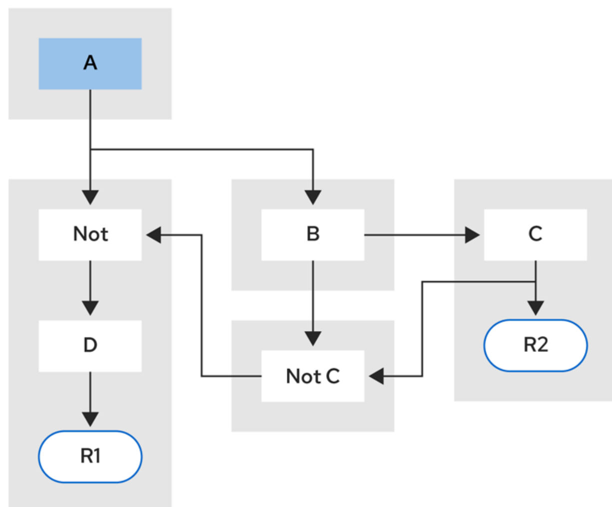
図5.5 例 4: パターンの共有なしで、サブネットワークがある単一のルール



Not、**Exists**、または **Accumulate** ノードに 1 つ以上の要素がある場合、サブネットワークが形成されます。この例では、要素 **B not(C)** がサブネットワークを形成します。要素 **not(C)** は、サブネットワークを必要としない単一要素のため、**Not** ノード内にマージされます。サブネットワークは専用のセグメントを使用します。ルール R1 には、依然として 2 つのセグメントのパスがあり、サブネットワークは別の内部のパスを形成します。サブネットワークがリンクされると、これは外部のセグメントにもリンクされます。

例 5: ルール R2 と共有するサブネットワークのあるルール R1

図5.6 例 5: 2 つのルールのうち、1 つはサブネットワークとパターンを共有



ルールのサブネットワークノードは、サブネットワークのない別のルールと共有することができます。このように共有することで、サブネットワークのセグメントが 2 つのセグメントに分割されることになります。

制約のある **Not** ノードおよび **Accumulate** ノードは、セグメントのリンクを外すことは一切できず、ビットがオンになっていると常に考えられています。

Phreak 評価のアルゴリズムは、メソッド再帰ベースではなく、スタックベースです。**StackEntry** を使用して現在評価中のノードを表す場合は、いつでもルール評価を一時的に停止したり、再開したりすることができます。

ルール評価がサブネットワークに到達すると、**StackEntry** オブジェクトが、外部パスのセグメントおよびサブネットワークセグメント用に作成されます。最初にサブネットワークのセグメントが評価され、セットがサブネットワークパスの終わりに到達すると、そのセグメントがフィードする外部ノードのステージングリストにマージされます。次に、以前の **StackEntry** オブジェクトが再開し、サブネットワークの結果を処理できるようになります。この処理には、子ノードに伝播される前にすべての作業がバッチで完了するという付加的な利点があります。これは、**Accumulate** ノードにとって特に有利です。

同じスタックシステムは、効率的な後向き連鎖に使用されます。ルール評価がクエリーノードに到達すると、評価は一時的に停止し、クエリーがスタックに追加されます。続いてクエリーは、結果セットを生成するために評価されます。結果セットは、再開した **StackEntry** オブジェクトのメモリーロケーションに保存され、回収されて子ノードに伝播されます。クエリー自体が他のクエリーを呼び出した場合はこの処理は繰り返され、その一方で、現在のクエリーは一時的に停止し、現在のクエリーノード用に新しい評価が設定されます。

5.1.1. 前向き連鎖と後向き連鎖を使用したルール評価

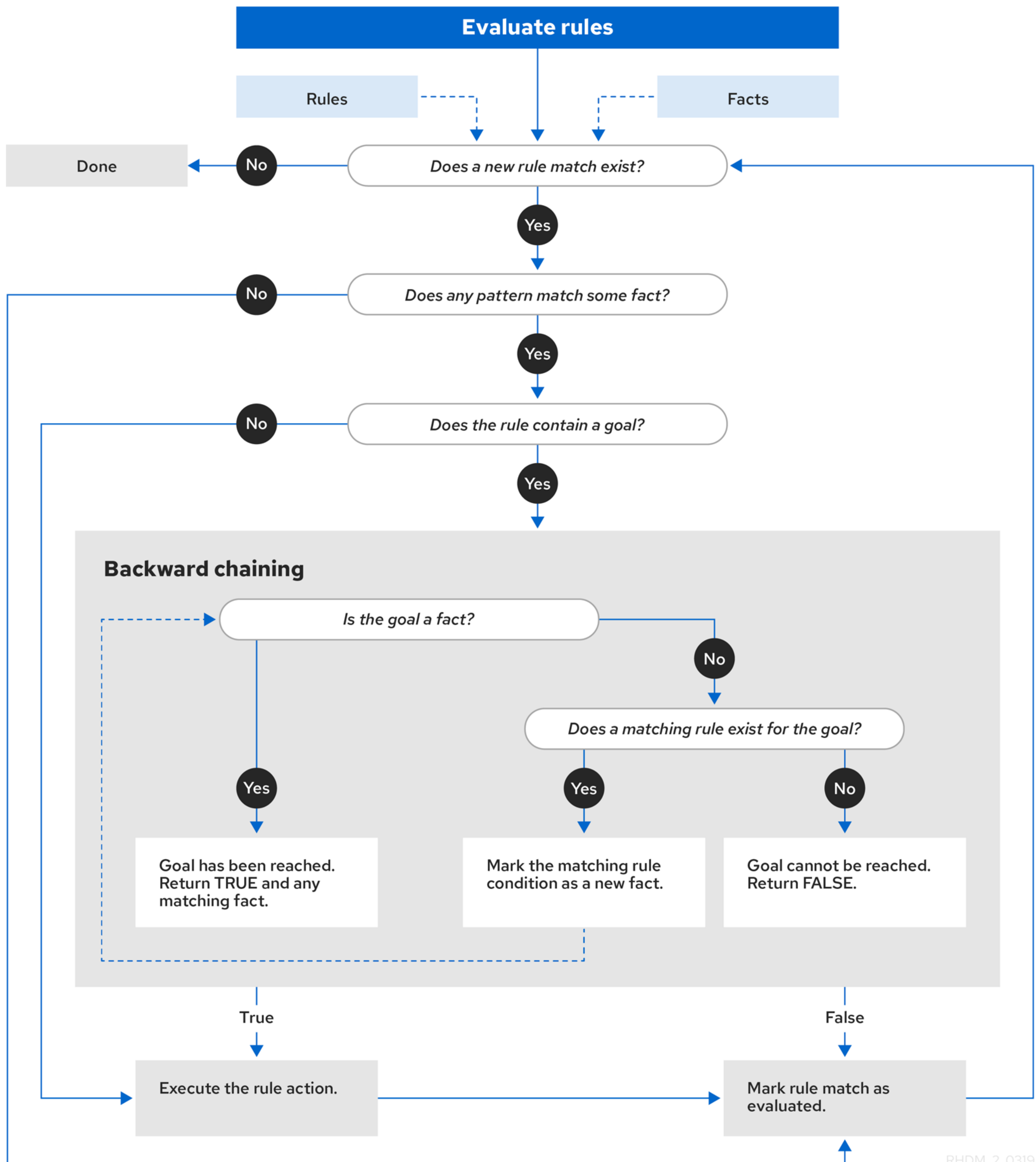
Red Hat Process Automation Manager のデシジョンエンジンは、前向き連鎖と後向き連鎖の両方を使用してルールを評価する、ハイブリッドの理由付けシステムです。前向き連鎖のルールシステムは、デシジョンエンジンのワーキングメモリーにあるファクトで開始して、そのファクトへの変更に反応するデータ駆動型のシステムです。オブジェクトがワーキングメモリーに挿入されると、その変更の結果として True となったルールの条件は、アジェンダにより実行がスケジュールされます。

反対に、後向き連鎖のルールシステムは、通常再帰を使用して、デシジョンエンジンが満たそうとする結論から開始する目的駆動型のシステムです。システムが結論または目的に到達できない場合には、サ

ブとなる目的、つまり、現在の目的の一部を完了する結論を検索します。システムは、最初の結論が満たされるか、すべてのサブとなる目的が満たされるまで続行されます。

以下の図は、デシジョンエンジンが、ロジックフローで後向き連鎖のセグメントと、前向き連鎖全体とを使用してルールを評価する方法を例示します。

図5.7 前向き連鎖と後向き連鎖を使用したルール評価のロジック



5.2. ルールベースの設定

Red Hat Process Automation Manager には、**RuleBaseConfiguration.java** オブジェクトが含まれます。これを使用して、デシジョンエンジンで例外ハンドラーの設定、マルチスレッドの実行、および順次モードを設定することができます。

ルールベースの設定オプションに関しては、Red Hat カスタマーポータル から **Red Hat Process Automation Manager 7.8.0 Source Distribution** の ZIP ファイルをダウンロードし、 **~/rhpam-7.8.0-sources/src/drools-$VERSION/drools-core/src/main/java/org/drools/core/RuleBaseConfiguration.java** に移動してください。

以下のルールベースの設定オプションは、デシジョンエンジンで利用可能です。

drools.consequenceExceptionHandler

設定されると、このシステムプロパティーは、ルールの結果によって例外のスローを管理するクラスを定義します。このプロパティーを使用して、デシジョンエンジンのルール評価にカスタムの例外ハンドラーを指定できます。
デフォルト値: **org.drools.core.runtime.rule.impl.DefaultConsequenceExceptionHandler**

以下のオプションのいずれかを使用して、カスタムの例外ハンドラーを指定できます。

- システムプロパティーで例外ハンドラーを指定:

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

- プログラムを用いて KIE ベースを作成中に例外ハンドラーを指定:

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

drools.multithreadEvaluation

有効化されると、このシステムプロパティーは、Phreak のルールネットワークを個々のパーティションに分割することで、デシジョンエンジンが並列してルールを評価できるようにします。
デフォルト値: **false**

以下のオプションのいずれかを使用してマルチスレッド評価を有効化できます。

- マルチスレッド評価のシステムプロパティーを有効化:

```
drools.multithreadEvaluation=true
```

- プログラムを用いて KIE ベースを作成中にマルチスレッド評価を有効化:

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

警告

クエリー、顕著性、またはアジェンダグループを使用するルールは現在、並列のデシジョンエンジンではサポートされていません。これらのルールの要素が KIE ベースに存在する場合、コンパイラーは警告を発し、自動的にシングルスレッドの評価に切り替えます。しかし、ケースによっては、デシジョンエンジンはサポートされていないルールの要素を検出できず、ルールが間違って評価される可能性があります。たとえば、ルールが DRL ファイル内のルールの順序によって与えられた暗黙の顕著性に依存する場合、デシジョンエンジンは検出できない可能性があり、その結果、サポートされていない顕著性の属性により、間違った評価となります。

drools.sequential

有効化されると、このシステムプロパティーは、デシジョンエンジンの順次モードを有効化します。順次モードでは、デシジョンエンジンは、ワーキングメモリーでの変更に関係なく、デシジョンエンジンアジェンダにリスト化された順番でルールを一度評価します。これは、デシジョンエンジンがルールの **insert**、**modify**、または **update** ステートメントをすべて無視し、ルールを単一シーケンスで実行することを意味します。その結果、ルールの実行は順次モードの方が速くなりますが、重要な更新はルールに適用されない可能性があります。ステートレスな KIE セッションを使用し、アジェンダ内の後続のルールに対してルールの実行による影響を与えないようにする場合に、このプロパティーを使用できます。順次モードは、ステートレスな KIE セッションにのみ適用されます。
デフォルト値: **false**

以下のオプションのいずれかを使用して、順次モードを有効化できます。

- 順次モードのシステムプロパティーの有効化:

```
drools.sequential=true
```

- プログラムを用いて KIE ベースを作成中に順次モードを有効化:

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

- 特定の Red Hat Process Automation Manager プロジェクトの KIE モジュール記述子ファイル (**kmodule.xml**) で順次モードを有効化

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

5.3. PHREAK における順次モード

順次モードは、デシジョンエンジンにおける高度なルールベースの設定で、Phreak がサポートしています。順次モードでは、デシジョンエンジンは、ワーキングメモリーでの変更に関係なく、デシジョンエンジンアジェンダにリスト化された順番でルールを一度評価します。順次モードでは、デシジョンエンジンがルールの **insert**、**modify**、または **update** ステートメントをすべて無視し、ルールを単一シーケンスで実行します。その結果、ルールの実行は順次モードの方が速くなりますが、重要な更新はルールに適用されない可能性があります。

ステートフルな KIE セッションは本来、以前呼び出された KIE セッションのデータを使用するため、順次モードが適用されるのはステートレスな KIE セッションのみとなります。ステートレスな KIE セッションを使用し、ルールを実行して、アジェンダ内の後続のルールを決定するには、順次モードを有効化しないでください。デシジョンエンジンでは、デフォルトで順次モードは無効となっています。

以下のオプションのいずれかを使用して、順次モードを有効化します。

- システムプロパティー **drools.sequential** を **true** に設定:
- プログラムを用いて KIE ベースを作成中に順次モードを有効化:

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

- 特定の Red Hat Process Automation Manager プロジェクトの KIE モジュール記述子ファイル (**kmodule.xml**) で順次モードを有効化

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

以下のオプションのいずれかを使用して、順次モードが動的アジェンダを使用するように設定します。

- システムプロパティー **drools.sequential.agenda** を **dynamic** に設定:
- プログラムを用いて KIE ベースを作成中に順次アジェンダオプションを設定:

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

順次モードを有効化すると、以下の方法でデシジョンエンジンがルールを評価します。

1. ルールは、ルールセットの顕著性および位置によって順序付けられます。
2. 考えられるルールの一致ごとの要素が作成されました。要素の位置が実行の順番を示しています。

3. right-input オブジェクトメモリーを除いて、ノードメモリーは無効化されました。
4. left-input アダプターノードの伝播は切断され、ノードを持つオブジェクトは **Command** オブジェクトで参照されます。**Command** オブジェクトは、後で実行するためにワーキングメモリーのリストに追加されます。
5. すべてのオブジェクトがアサートされると、**Command** オブジェクトのリストが確認され、実行されます。
6. リストの実行によって生じるすべての一致は、ルールのシーケンス番号に基づいて要素に追加されます。
7. 一致を含む要素は、順次実行されます。ルール実行の最大数を設定している場合、デシジョンエンジンがアジェンダで実行するルールの数は、この最大数を超えることはありません。

順次モードでは、**LeftInputAdapterNode** ノードが **Command** オブジェクトを作成し、これをデシジョンエンジンのワーキングメモリーリストに追加します。この **Command** オブジェクトには、**LeftInputAdapterNode** ノードおよび伝播されたオブジェクトへの参照が含まれます。これらの参照は、right-input 伝播が left-input の結合を試みる必要が一切ないように、挿入時にあらゆる left-input 伝播を停止します。また、参照により、left-input メモリーが不要となります。

すべてのノードのメモリーはオフになっています。これには、left-input のタプルメモリーも含まれますが、right-input オブジェクトメモリーは除外されます。すべてのアサーションが終了し、すべてのオブジェクトの right-input メモリーが生成されると、デシジョンエンジンは **LeftInputAdatperNode** と **Command** オブジェクトのリストを繰り返します。オブジェクトはネットワークを伝播し、right-input オブジェクトに結合しようと試みますが、left input では保持されません。

タプルをスケジュールするための優先度キューがあるアジェンダは、各ルールの要素に置き換えます。**RuleTerminalNode** ノードの順番で、要素に対して、一致をどこに配置するか指定します。**Command** オブジェクトすべてが終了すると、要素が確認され、既存の一致が実行されます。要素内で最初と最後に生成されたセルを保持してパフォーマンスを向上します。

ネットワークが構築されると、各 **RuleTerminalNode** ノードは、顕著性の番号とネットワークに追加された順序をベースとするシーケンス番号を受け取ります。

right-input ノードメモリーは通常、オブジェクトをすばやく削除するためのハッシュマップです。オブジェクトの削除はサポートされていないので、オブジェクトの値がインデックス化されていない場合、Phreak はオブジェクトリストを使用します。大量のオブジェクトに対しては、インデックス化されたハッシュマップがパフォーマンスを向上させます。オブジェクトのインスタンスが少しだけの場合、Phreak はインデックスの代わりにオブジェクトリストを使用します。

第6章 複合イベント処理 (CEP)

Red Hat Process Automation Manager では、イベントとは、ある時点でのアプリケーションドメインの状態の大幅な変化の記録です。ドメインのモデル化方法に応じて、状態の変化は単一のイベント、複数のアトミックイベント、または相関イベントの階層によって表される場合があります。複合イベント処理 (CEP) の観点から見ると、イベントは特定の時点で発生するファクトまたはオブジェクトのタイプであり、ビジネスルールはそのファクトまたはオブジェクトからのデータにどのように反応するかを定義したものです。たとえば、株式ブローカーアプリケーションでは、株価の変動、売り手から買い手への所有権の変更、またはアカウント所有者の残高の変更はすべて、所定の時間にアプリケーションドメインの状態で変更が発生したため、イベントと見なされます。

Red Hat Process Automation Manager のデシジョンエンジンは、複合イベント処理 (CEP) を使用して、イベントのコレクション内の複数のイベントを検出および処理し、イベント間に存在する関係を明らかにするほか、イベントとイベント同士の関係から新しいデータを推論します。

CEP のユースケースは、複数の要件と目標をビジネスルールのユースケースと共有しています。

ビジネスの観点から見ると、ビジネスルールの定義は多くの場合、イベントによってトリガーされるシナリオの発生に基づいて定義されます。以下の例では、イベントがビジネスルールの基礎を形成しています。

- アルゴリズム取引アプリケーションでは、株価が始値を X パーセント上回った場合、ルールがアクションを実行します。価格の上昇は、株式取引アプリケーションのイベントによって示されます。
- 監視アプリケーションでは、サーバールームの温度が Y 分で X 度上昇すると、ルールがアクションを実行します。センサーの測定値はイベントによって示されます。

技術的な観点から見ると、ビジネスルールの評価と CEP には、以下の重要な類似点があります。

- ビジネスルールの評価と CEP の両方で、エンタープライズインフラストラクチャーとアプリケーションとのシームレスな統合が必要です。これは、ライフサイクル管理、監査、およびセキュリティーにおいて特に重要です。
- ビジネスルールの評価と CEP の両方には、パターン一致などの機能要件と、応答時間の制限やクエリールールの説明などの非機能要件があります。

CEP シナリオには、以下の重要な特徴があります。

- シナリオは通常、大量のイベントを処理しますが、関連するイベントはごく一部です。
- 通常、イベントは不変であり、状態の変化の記録を表します。
- ルールとクエリーはイベントに対して実行され、検出されたイベントパターンに対応する必要があります。
- 通常、関連するイベントには強い一時的な関係があります。
- 個々のイベントは優先されません。CEP システムは、関連するイベントのパターンとイベント間の関係に優先順位を付けます。
- 通常、イベントは構成および集約を行う必要があります。

これらの一般的な CEP シナリオの特徴を前提として、Red Hat Process Automation Manager の CEP システムは、イベント処理を最適化するために以下の機能をサポートしています。

- 適切なセマンティクスによるイベント処理

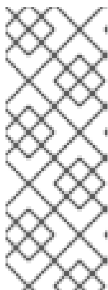
- イベントの検出、相関、集約、および構成
- イベントストリーム処理
- イベント間の一時的な関係をモデル化する一時的な制約
- 重要なイベントのスライディングウィンドウ
- セッションスコープの統合クロック
- CEP ユースケースに必要なイベントのボリューム
- リアクティブルール
- デシジョンエンジンへのイベント入力アダプター (パイプライン)

6.1. 複合イベント処理 (CEP) におけるイベント

Red Hat Process Automation Manager では、イベントとは、ある時点でのアプリケーションドメインの状態の大幅な変化の記録です。ドメインのモデル化方法に応じて、状態の変化は単一のイベント、複数のアトミックイベント、または相関イベントの階層によって表される場合があります。複合イベント処理 (CEP) の観点から見ると、イベントは特定の時点で発生するファクトまたはオブジェクトのタイプであり、ビジネスルールはそのファクトまたはオブジェクトからのデータにどのように反応するかを定義したものです。たとえば、株式ブローカーアプリケーションでは、株価の変動、売り手から買い手への所有権の変更、またはアカウント所有者の残高の変更はすべて、所定の時間にアプリケーションドメインの状態で変更が発生したため、イベントと見なされます。

イベントには、以下の重要な特徴があります。

- **不変性:** イベントは、過去のある時点で発生した変更の記録であり、変更することはできません。



注記

デシジョンエンジンは、イベントを表す Java オブジェクトに不変性を強制しません。この動作により、イベントデータの強化が可能になります。アプリケーションは、未入力のイベント属性を入力できる必要があります。そしてこれらの属性は、推論データでイベントを強化するためにデシジョンエンジンによって使用されます。ただし、すでに入力されているイベント属性は変更しないでください。

- **強力な一時的な制約:** 通常、イベントに関係するルールは、相互に関連する異なる時点で発生する複数のイベントの相関を必要とします。
- **管理されたライフサイクル:** イベントは不変であり、一時的な制約があるため、通常は特定の期間にのみ関連します。これは、デシジョンエンジンがイベントのライフサイクルを自動的に管理できることを意味します。
- **スライディングウィンドウが使用可能:** イベントで時間または長さのスライディングウィンドウを定義できます。スライディングタイムウィンドウは、イベントを処理できる特定の期間です。スライディングレングスウィンドウは、処理可能な指定されたイベントの数です。

6.2. ファクトのイベントとしての宣言

Java クラスまたは DRL ルールファイルでファクトをイベントとして宣言すると、デシジョンエンジン

が複雑なイベント処理中にファクトをイベントとして処理できます。ファクトは、interval-based イベントまたは point-in-time イベントとして宣言できます。interval-based のイベントには持続期間があり、その持続期間が経過するまでデシジョンエンジンのワーキングメモリーで持続します。point-in-time イベントには持続期間はなく、基本的には期間がゼロの interval-based イベントになります。

手順

Java クラスまたは DRL ルールファイルの関連するファクトタイプについては、**@role(event)** メタデータタグとパラメーターを入力します。**@role** メタデータタグは、以下の 2 つの値を受け入れます。

- **fact**: (デフォルト) タイプを通常のファクトとして宣言
- **event**: タイプをイベントとして宣言

たとえば、以下のスニペットは、株式ブローカーアプリケーションの **StockPoint** ファクトタイプをイベントとして処理する必要があることを宣言しています。

ファクトタイプをイベントとして宣言

```
import some.package.StockPoint

declare StockPoint
  @role( event )
end
```

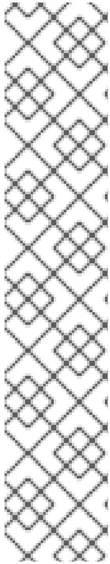
StockPoint が、既存のクラスではなく DRL ルールファイルで宣言されたファクトタイプである場合、アプリケーションコードでイベントをインラインで宣言できます。

ファクトタイプをインラインで宣言し、イベントロールに割り当てる

```
declare StockPoint
  @role( event )

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

6.3. イベントのメタデータタグ

デシジョンエンジンは、デシジョンエンジンのワーキングメモリーに挿入されるイベントに以下のメタデータタグを使用します。必要に応じて、Java クラスまたは DRL ルールファイルでデフォルトのメタデータタグ値を変更できます。



注記

VoiceCall クラスを参照する本セクションの例では、サンプルアプリケーションドメインモデルに以下のクラスの詳細が含まれていることを前提としています。

Telecom ドメインモデルの例における VoiceCall ファクトクラス

```
public class VoiceCall {
  private String  originNumber;
  private String  destinationNumber;
  private Date    callDateTime;
  private long    callDuration;  // in milliseconds

  // Constructors, getters, and setters
}
```

@role

このタグは、指定のファクトタイプが複雑なイベントの処理時にデシジョンエンジンにて通常のファクトまたはイベントとして処理されるかどうかを決定します。
デフォルトパラメーター: **fact**

サポート対象のパラメーター: **fact**、**event**

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

例: イベントタイプとして VoiceCall の宣言

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

@timestamp

このタグは、デシジョンエンジンのすべてのイベントに自動的に割り当てられます。デフォルトでは、タイムはセッションクロックにより提供され、デシジョンエンジンのワーキングメモリーへの挿入時にイベントに割り当てられます。セッションクロックが追加するデフォルトのタイムスタンプの代わりに、カスタムのタイムスタンプ属性を指定することができます。
デフォルトパラメーター: デシジョンエンジンのセッションクロックが追加する時間

サポート対象のパラメーター: セッションクロックタイムまたはカスタムのタイムスタンプ属性

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

例: VoiceCall のタイムスタンプ属性の宣言

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

@duration

このタグは、デシジョンエンジンのイベントの持続期間を決定します。イベントは、interval-based

イベントまたは point-in-time イベントのいずれかになります。interval-based イベントには持続期間があり、この持続期間が経過するまでデシジョンエンジンのワーキングメモリーで持続します。point-in-time イベントには持続期間はなく、基本的には期間の時間単位が 0 の interval-based イベントと同じです。デフォルトでは、デシジョンエンジンのすべてのイベントの持続期間は 0 です。デフォルトの代わりに、カスタムの持続期間属性を指定することができます。
デフォルトパラメーター: null (ゼロ)

サポート対象のパラメーター: カスタムの持続期間属性

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

例: VoiceCall の持続期間属性の宣言

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

@expires

このタグは、デシジョンエンジンのワーキングメモリーでイベントの有効期限が切れるまでの時間を決定します。デフォルトでは、イベントは現在のルールのいずれにも一致せず、それらのいずれもアクティベートできなくなった時点で失効します。イベント失効後の期間を定義できます。また、このタグの定義は、KIE ベースの一時的な制約やスライディングウィンドウから算出した暗黙的な有効期限のオフセットもオーバーライドします。デシジョンエンジンがストリームモードで実行中の場合にのみ、このタグを使用できます。
デフォルトパラメーター: null (イベントがルールに一致せず、ルールをアクティブにできなくなるとイベントの有効期限が切れる)

サポート対象のパラメーター: **[#d][#h][#m][#s][[ms]]** 形式のカスタムの **timeOffset** 属性

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

例: VoiceCall イベントに対する有効期限のオフセットの宣言

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

6.4. デシジョンエンジンのイベント処理モード

デシジョンエンジンは、クラウドモードまたはストリームモードで実行されます。クラウドモードでは、デシジョンエンジンは、ファクトを一時的な制約がなく、時間に依存せず、順不同のファクトとして処理します。ストリームモードでは、デシジョンエンジンは、ファクトをリアルタイムまたはほぼリアルタイムで、強力な一時的な制約のあるイベントとして処理します。ストリームモードは同期を使用して、Red Hat Process Automation Manager でのイベント処理を可能にします。

クラウドモード

クラウドモードは、デシジョンエンジンのデフォルトの動作モードです。クラウドモードでは、デシジョンエンジンはイベントを順不同のクラウドとして扱います。イベントには依然としてタイムスタンプがありますが、クラウドモードでは現在の時刻が無視されるため、クラウドモードで実行されているデシジョンエンジンは、タイムスタンプから関連性を引き出すことができません。このモードでは、ルール制約を使用して一致するタプルを検索し、ルールを有効化して実行します。
クラウドモードでは、ファクトに対して追加要件を課すことは一切ありません。ただし、このモードのデシジョンエンジンには時間の概念がないため、スライディングウィンドウや自動ライフサイクル管理などの一時的な機能を使用できません。クラウドモードでは、イベントが必要なくなると、明示的にイベントを取り消す必要があります。

クラウドモードでは、以下の要件が課されることはありません。

- デシジョンエンジンには時間の概念がないため、クロックの同期はありません
- デシジョンエンジンは、イベントを順不同のクラウドとして処理するため、イベントの順序付けはありませんが、デシジョンエンジンは順不同のクラウドに対してルールを一致させます。

関連する設定ファイルでシステムプロパティーを設定するか、または Java クライアント API を使用して、クラウドモードを指定できます。

システムプロパティーを使用してクラウドモードを設定

```
drools.eventProcessingMode=cloud
```

Java クライアント API を使用してクラウドモードを設定

```
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);
```

特定の Red Hat Process Automation Manager プロジェクトの KIE モジュール記述子ファイル (**kmodule.xml**) の **eventProcessingMode="<mode>"** KIE ベース属性を使用して、クラウドモードを指定することもできます。

プロジェクト kmodule.xml ファイルを使用してクラウドモードを設定

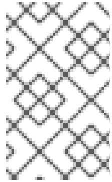
```
<kmodule>
  ...
  <kbase name="KBase2" default="false" eventProcessingMode="cloud"
packages="org.domain.pkg2, org.domain.pkg3" includes="KBase1">
    ...
  </kbase>
  ...
</kmodule>
```

ストリームモード

イベントがデシジョンエンジンに挿入されると、デシジョンエンジンは、ストリームモードを使用することで、イベントを時系列およびリアルタイムに処理できます。ストリームモードでは、(異なるストリームのイベントを時系列で処理できるように) デシジョンエンジンはイベントのストリーム

を同期し、時間または長さのスライディングウィンドウを実装し、自動ライフサイクル管理を可能にします。
以下の要件がストリームモードに適用されます。

- 各ストリームのイベントは、時系列に並べる必要があります。
- イベントストリームを同期するには、セッションクロックが必要です。



注記

お使いのアプリケーションが、ストリーム間でイベントの順序付けを強制する必要はありませんが、同期されていないイベントストリームを使用すると、予期しない結果が生じる可能性があります。

関連する設定ファイルでシステムプロパティーを設定するか、または Java クライアント API を使用して、ストリームモードを指定できます。

システムプロパティーを使用してストリームモードを設定

```
drools.eventProcessingMode=stream
```

Java クライアント API を使用してストリームモードを設定

```
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);
```

特定の Red Hat Process Automation Manager プロジェクトの KIE モジュール記述子ファイル (**kmodule.xml**) の **eventProcessingMode="<mode>"** KIE ベース属性を使用して、ストリームモードを指定することもできます。

プロジェクト kmodule.xml ファイルを使用してストリームモードを設定

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

6.4.1. デシジョンエンジンのストリームモードにおける負のパターン

負のパターンは、条件が一致しない場合のパターンです。たとえば、以下の DRL ルールは、火災が検知されてもスプリンクラーが有効になっていない場合に、火災報知機を有効にします。

負のパターンでの火災報知機ルール

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

クラウドモードでは、デシジョンエンジンはすべてのファクト (通常のファクトとイベント) が前もって知らされているものと想定し、負のパターンをすぐに評価します。ストリームモードでは、デシジョンエンジンは、ファクトの一時的な制約をサポートして、ルールを有効化する前に一定期間、待機することができます。

ストリームモードでは、同じ例のルールは、通常どおりに火災報知機を有効にしますが、10 秒間の遅延が適用されます。

負のパターンと遅延時間のある火災報知機ルール (ストリームモードのみ)

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

以下の修正された火災報知器ルールは、10 秒ごとに 1 つの **Heartbeat** イベントが発生することを想定しています。想定されるイベントが発生しない場合、ルールが実行されます。このルールは、最初のパターンと負のパターンの両方で同じタイプのオブジェクトを使用します。負のパターンには、実行前に 0 秒から10 秒待機する一時的な制約があり、**$h** にバインドされている **Heartbeat** イベントを除外して、ルールを実行できるようにします。ルールを実行するには、バインドされたイベント **$h** を明示的に除外する必要があります。なぜなら、一時的な制約 **[0s, …]** は、このイベントが再び一致することを本質的に除外しないからです。

負のパターンでバインドされたイベントを除外する火災報知器ルール (ストリームモードのみ)

```
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. ファクトタイプに対するプロパティー変更の設定およびリスナー

デフォルトでは、デシジョンエンジンは、ルールがトリガーされるたびに、ファクトタイプに対するすべてのファクトパターンを再評価しません。代わりに、指定のパターン内に制約またはバインドされている変更されたプロパティーのみに対応します。たとえば、ルールが、ルールアクションの一環として **modify()** を呼び出すものの、アクションが KIE ベースで新しいデータを生成しない場合、データが変更されないため、デシジョンエンジンはすべてのファクトパターンを自動的に再評価しません。このプロパティーのリアクティビティー動作は、KIE ベースでの不要な再帰を阻止し、より効率的なルール評価をもたらします。また、この動作は無限再帰を回避するために **no-loop** ルール属性を必ずしも使用する必要がないことを意味します。

以下の **KnowledgeBuilderConfiguration** オプションを使用して、このプロパティーリアクティビティー動作を変更または無効にできます。次に、Java クラスまたは DRL ファイルでプロパティー変更設定を使用し、必要に応じてプロパティーリアクティビティーを調整します。

- **ALWAYS**: (デフォルト) すべてのタイプはプロパティーリアクティブです。ただし、**@classReactive** プロパティー変更設定を使用して、特定タイプのプロパティーリアクティビティーを無効にできます。
- **ALLOWED**: すべてのタイプはプロパティーリアクティブではありません。ただし、**@propertyReactive** プロパティー変更設定を使用して、特定タイプのプロパティーリアクティビティーを有効にできます。
- **DISABLED**: すべてのタイプはプロパティーリアクティブではありません。すべてのプロパティー変更リスナーは無視されます。

KnowledgeBuilderConfiguration におけるプロパティーリアクティビティー設定の例

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

または、Red Hat Process Automation Manager ディストリビューションにおける **standalone.xml** ファイルの **drools.propertySpecific** システムプロパティーを更新できます。

システムプロパティーにおけるプロパティーリアクティビティー設定の例

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

デシジョンエンジンは、ファクトクラスまたは宣言された DRL ファクトタイプに対して、以下のプロパティー変更の設定およびリスナーをサポートします。

@classReactive

デシジョンエンジンでプロパティーリアクティビティーが **ALWAYS** に設定されている場合 (すべてのタイプはプロパティーリアクティブ)、このタグは特定の Java クラスまたは宣言された DRL ファクトタイプに対してデフォルトのプロパティーリアクティビティー動作を無効にします。このタグは、特定パターン内に制約またはバインドされる変更されたプロパティーのみに対応するのではなく、ルールがトリガーされるたびに指定されたファクトタイプのすべてのファクトパターンをデシジョンエンジンが再評価する必要がある場合に使用できます。

例: DRL タイプの宣言におけるデフォルトのプロパティーリアクティビティーの無効化

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

例: Java クラスにおけるデフォルトのプロパティーリアクティビティーの無効化

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

@propertyReactive

プロパティーリアクティビティーがデシジョンエンジンで **ALLOWED** に設定されている場合 (指定されていない場合、すべてのタイプはプロパティーリアクティブではない)、このタグは特定の Java クラスまたは宣言された DRL ファクトタイプに対してプロパティーリアクティビティーを有効にします。デシジョンエンジンが指定されたファクトタイプに対して指定のパターン内に制約またはバインドされている変更されたプロパティーのみに対応するようにする場合に、このタグを使用できます。

例: DRL タイプの宣言におけるプロパティーリアクティビティーの有効化 (リアクティビティーがグローバルに無効にされる場合)

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

例: Java クラスでのプロパティーのリアクティビティーの有効化 (リアクティビティーがグローバルに無効にされる場合)

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

@watch

このタグは、DRL ルールのファクトパターンで、インラインで指定する追加のプロパティーに対するプロパティーリアクティビティーを有効化します。このタグがサポートされるのは、デシジョンエンジンでプロパティーリアクティビティーが **ALWAYS** に設定されている場合か、またはプロパティーリアクティビティーが **ALLOWED** に設定され、関連するファクトタイプが **@propertyReactive** タグを使用する場合に限られます。DRL ルールでこのタグを使用して、ファクトプロパティーリアクティビティー論理の指定されたプロパティーを追加または除外できます。
デフォルトパラメーター: なし

サポートされているパラメーター: プロパティー名、***** (all)、**!** (not)、**!*** (no properties)

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

例: ファクトパターンにおけるプロパティーリアクティビティーの有効化または無効化

```
// 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( !* )
```

デシジョンエンジンは、**@classReactive** タグ (プロパティーリアクティビティーを無効にする) を使用するファクトタイプのプロパティーに対して **@watch** タグを使用する場合に、またはデシジョンエンジンでプロパティーリアクティビティーが **ALLOWED** に設定され、関連するファクトタイプが **@propertyReactive** タグを使用しない場合に、コンパイルエラーを生成します。また、**@watch(firstName, ! firstName)** などのリスナーアノテーションでプロパティーを複製する場合でも、コンパイルエラーが生じます。

@propertyChangeSupport

JavaBeans Specification で定義されたプロパティー変更のサポートを実装するファクトの場合、このタグによりデシジョンエンジンがファクトプロパティーの変更を監視できるようになります。

例: JavaBeans オブジェクトでのプロパティー変更のサポートの宣言

```
declare Person
    @propertyChangeSupport
end
```

6.6. イベントの一時オペレーター

ストリームモードでは、デシジョンエンジンは、デシジョンエンジンのワーキングメモリーに挿入されるイベントに対して以下の一時オペレーターをサポートします。これらのオペレーターを使用して、Java クラスまたは DRL ルールファイルで宣言するイベントの一時的な理由付け動作を定義できます。デシジョンエンジンがクラウドモードで実行されている場合は、一時オペレーターはサポートされません。

- **after**
- **before**
- **coincides**
- **during**
- **includes**
- **finishes**
- **finished by**
- **meets**
- **met by**

- **overlaps**
- **overlapped by**
- **starts**
- **started by**

after

このオペレーターは、相関イベントの後に現在のイベントが発生するかどうかを指定します。また、このオペレーターは時間を定義でき、この時間の後に、現在のイベントは相関イベントを追跡することができます。または、現在のイベントが相関イベントを追跡できる区切られた時間範囲を定義することもできます。
たとえば、以下のパターンは、**$eventA** が **$eventB** の終了後 3 分 30 秒から 4 分の間に開始する場合に一致します。 **$eventA** が **$eventB** の終了後 3 分 30 秒よりも前に開始する場合、または **$eventB** の終了後 4 分より後に開始する場合は、パターンは一致しません。

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

以下の方法で、このオペレーターを表すこともできます。

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

after オペレーターは、パラメーター値を 2 つまでサポートします。

- 2 つの値が定義されると、間隔は 1 番目の値 (例では 3 分 30 秒) で開始し、2 番目の値 (例では 4 分) で終了します。
- 1 つの値のみ定義すると、間隔は提示した値で開始し、終了時間なしで無期限に実行されます。
- 値が定義されない場合は、間隔は 1 ミリ秒から開始し、終了時間なしで無期限に実行されます。

after オペレーターは、負の時間範囲もサポートしています。

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

1 番目の値が 2 番目の値より大きい場合、デシジョンエンジンは順番を自動的に入れ替えます。たとえば、デシジョンエンジンは以下の 2 つのパターンを同じものと解釈します。

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

before

このオペレーターは、相関イベントの前に現在のイベントが発生するかどうかを指定します。また、このオペレーターは時間を定義でき、この時間の前に、現在のイベントは相関イベントに先行することができます。または、現在のイベントが相関イベントに先行できる区切られた時間範囲を定義することもできます。
たとえば、以下のパターンは、**$eventA** が **$eventB** の開始前 3 分 30 秒から 4 分の間に終了する場合に一致します。 **$eventA** が **$eventB** の開始前 3 分 30 秒よりも前に終了する場合、または **$eventB** の開始前 4 分より後に終了する場合は、パターンは一致しません。

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

以下の方法で、このオペレーターを表すこともできます。

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

before オペレーターは、パラメーター値を 2 つまでサポートします。

- 2 つの値が定義されると、間隔は 1 番目の値 (例では 3 分 30 秒) で開始し、2 番目の値 (例では 4 分) で終了します。
- 1 つの値のみ定義すると、間隔は提示した値で開始し、終了時間なしで無期限に実行されます。
- 値が定義されない場合は、間隔は 1 ミリ秒から開始し、終了時間なしで無期限に実行されます。

before オペレーターは、負の時間範囲もサポートしています。

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

1 番目の値が 2 番目の値より大きい場合、デシジョンエンジンは順番を自動的に入れ替えます。たとえば、デシジョンエンジンは以下の 2 つのパターンを同じものと解釈します。

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

coincides

このオペレーターは、2 つのイベントが同じ開始時刻と終了時刻で同時に発生するかどうかを指定します。
たとえば、**$eventA** と **$eventB** の開始タイムスタンプと終了タイムスタンプの両方が同一の場合、以下のパターンは一致します。

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

coincides オペレーターは、イベントの開始時間と終了時間の間隔が同じではない場合、最大 2 つのパラメーター値をサポートします。

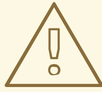
- パラメーターが 1 つだけ指定されている場合、このパラメーターを使用して、両方のイベントの開始時間と終了時間のしきい値が設定されます。
- パラメーターが 2 つ指定されている場合、1 番目のパラメーターは開始時間のしきい値として使用され、2 番目のパラメーターは終了時間のしきい値として使用されます。

以下のパターンでは、開始時間と終了時間のしきい値を使用しています。

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

以下の条件が一致する場合、パターンは一致します。

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



警告

デシジョンエンジンは、**coincides** オペレーターの負の間隔をサポートしていません。負の間隔を使用すると、デシジョンエンジンはエラーを生成します。

during

このオペレーターは、相関イベントが開始および終了する時間枠内で現在のイベントが発生するかどうかを指定します。現在のイベントは、相関イベントの開始後に開始し、相関イベントの終了前に終了する必要があります。(**coincides** オペレーターを使用すると、開始時間と終了時間は同じか、ほぼ同じになります。)
たとえば、以下のパターンは、**$eventA** が **$eventB** の開始後に開始し、**$eventB** の終了前に終了する場合に一致します。

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

以下の方法で、このオペレーターを表すこともできます。

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

during オペレーターは、1、2、または 4 つのオプションのパラメーターをサポートします。

- 1 つの値が定義されている場合、この値は 2 つのイベントのそれぞれの開始時間の間隔が最大であるほか、2 つのイベントのそれぞれの終了時間の間隔が最大であることを示しています。
- 2 つの値が定義されている場合、これらの値はしきい値で、これらのしきい値の間では、現在のイベントの開始時間と終了時間が、相関イベントの開始時間と終了時間に関連して発生する必要があります。
  たとえば、値が **5s** と **10s** である場合、現在のイベントは相関イベントの開始後 5 秒から10 秒の間に開始し、相関イベント終了の 5 秒から10 秒前に終了する必要があります。
- 4 つの値が定義されている場合、1 番目と 2 番目の値は、各イベントの開始時間の最小間隔と最大間隔を表しています。また、3 番目と 4 番目の値は、2 つのイベントの終了時間の最小間隔と最大間隔を表しています。

includes

このオペレーターは、相関イベントが、現在のイベントが発生する時間枠内で発生するかどうかを指定します。相関イベントは、現在のイベントの開始後に開始し、現在のイベントの終了前に終了する必要があります。(このオペレーターの動作は、**during** オペレーターの動作の反対になります。)
たとえば、以下のパターンは、**$eventB** が **$eventA** の開始後に開始し、**$eventA** の終了前に終了する場合に一致します。

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

以下の方法で、このオペレーターを表すこともできます。

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

includes オペレーターは、1、2、または 4 つのオプションのパラメーターをサポートします。

- 1 つの値が定義されている場合、この値は 2 つのイベントのそれぞれの開始時間の間隔が最大であるほか、2 つのイベントのそれぞれの終了時間の間隔が最大であることを示しています。
- 2 つの値が定義されている場合、これらの値はしきい値で、これらのしきい値の間では、相関イベントの開始時間と終了時間が、現在のイベントの開始時間と終了時間に関連して発生する必要があります。
  たとえば、値が **5s** と **10s** である場合、相関イベントは現在のイベントの開始後 5 秒から10 秒の間に開始し、現在のイベント終了の 5 秒から10 秒前に終了する必要があります。
- 4 つの値が定義されている場合、1 番目と 2 番目の値は、各イベントの開始時間の最小間隔と最大間隔を表しています。また、3 番目と 4 番目の値は、2 つのイベントの終了時間の最小間隔と最大間隔を表しています。

finishes

このオペレーターは、現在のイベントが相関イベントの後に開始するが、両方のイベントが同時に終了するかどうかを指定します。
たとえば、以下のパターンは、**$eventA** が **$eventB** の開始後に開始し、**$eventB** と同時に終了する場合に一致します。

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

以下の方法で、このオペレーターを表すこともできます。

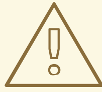
```
$eventB.startTimestamp < $eventA.startTimestamp
&&
$eventA.endTimestamp == $eventB.endTimestamp
```

finishes オペレーターは、2 つのイベントのそれぞれの終了時間の間隔に最大許容時間を設定する 1 つのオプションパラメーターをサポートします。

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

これらの条件が一致する場合、パターンは一致します。

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



警告

デシジョンエンジンは、**finishes** オペレーターに対して負の間隔をサポートしていません。負の間隔を使用すると、デシジョンエンジンはエラーを生成します。

finished by

このオペレーターは、相関イベントが現在のイベントの後に開始するが、両方のイベントが同時に終了するかどうかを指定します。(このオペレーターの動作は、**finishes** オペレーターの動作の逆になります。)
たとえば、以下のパターンは、**$eventB** が **$eventA** の開始後に開始し、**$eventA** と同時に終了する場合に一致します。

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

以下の方法で、このオペレーターを表すこともできます。

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

finished by オペレーターは、2 つのイベントのそれぞれの終了時間の間隔に最大許容時間を設定する 1 つのオプションパラメーターをサポートします。

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

これらの条件が一致する場合、パターンは一致します。

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

警告

デシジョンエンジンは、**finished by** オペレーターに対して負の間隔をサポートしていません。負の間隔を使用すると、デシジョンエンジンはエラーを生成します。

meets

このオペレーターは、現在のイベントが相関イベントの開始と同時に終了するかどうかを指定します。
たとえば、以下のパターンは、**$eventA** が **$eventB** の開始と同時に終了する場合に一致します。

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

以下の方法で、このオペレーターを表すこともできます。

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

meets オペレーターは、現在のイベントの終了時間と相関イベントの開始時間との間隔に最大許容時間を設定する 1 つのオプションパラメーターをサポートします。

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

これらの条件が一致する場合、パターンは一致します。

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

警告

デシジョンエンジンは、**meets** オペレーターに対して負の間隔をサポートしていません。負の間隔を使用すると、デシジョンエンジンはエラーを生成します。

met by

このオペレーターは、相関イベントが現在のイベントの開始と同時に終了するかどうかを指定します。(このオペレーターの動作は、**meets** オペレーターの動作の逆になります。) たとえば、以下のパターンは、**$eventB** が **$eventA** の開始と同時に終了する場合に一致します。

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

以下の方法で、このオペレーターを表すこともできます。

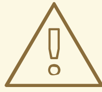
```
abs($eventA.startTimestamp - $eventB.endTimestamp) == 0
```

met by オペレーターは、相関イベントの終了時間と現在のイベントの開始時間との間に最大距離を設定する 1 つのオプションパラメーターをサポートします。

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

これらの条件が一致する場合、パターンは一致します。

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



警告

デシジョンエンジンは、**met by** オペレーターに対して負の間隔をサポートしていません。負の間隔を使用すると、デシジョンエンジンはエラーを生成します。

overlaps

このオペレーターは、現在のイベントが相関イベントの開始前に開始し、相関イベントが発生する時間枠内で終了するかどうかを指定します。現在のイベントは、相関イベントの開始時間と終了時間の間に終了する必要があります。
たとえば、以下のパターンは、**$eventA** が **$eventB** の開始前に開始し、**$eventB** の終了前に **$eventB** が発生する間に終了する場合に一致します。

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

overlaps オペレーターは、パラメーター値を 2 つまでサポートします。

- 1 つのパラメーターが定義されている場合、値は相関イベントの開始時間と現在のイベントの終了時間との間の最大距離になります。
- 2 つのパラメーターが定義されている場合、値は相関イベントの開始時間と現在のイベントの終了時間との間の最短距離 (1 番目の値) と最大距離 (2 番目の値) になります。

overlapped by

このオペレーターは、相関イベントが、現在のイベントの開始前に開始し、現在のイベントが発生する時間枠内で終了するかどうかを指定します。相関イベントは、現在のイベントの開始時間と終了時間の間に終了する必要があります。(このオペレーターの動作は、**overlaps** オペレーターの動作の反対になります。)
たとえば、以下のパターンは、**$eventB** が **$eventA** の開始前に開始し、**$eventA** の終了前に **$eventA** が発生する間に終了する場合に一致します。

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

overlapped by オペレーターは、パラメーター値を 2 つまでサポートします。

- 1 つのパラメーターが定義されている場合、値は現在のイベントの開始時間と相関イベントの終了時間との間の最大距離になります。
- 2 つのパラメーターが定義されている場合、値は現在のイベントの開始時間と相関イベントの終了時間との間の最短距離 (1 番目の値) と最大距離 (2 番目の値) になります。

starts

このオペレーターは、2 つのイベントが同時に開始するが、現在のイベントが相関イベントの終了前に終了するかどうかを指定します。
たとえば、以下のパターンは、**$eventA** と **$eventB** が同時に開始し、**$eventA** が **$eventB** の終了前に終了する場合に一致します。

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

以下の方法で、このオペレーターを表すこともできます。

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

starts オペレーターは、2 つのイベントのそれぞれの開始時間の間の最大距離を設定する 1 つのオプションパラメーターをサポートします。

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

これらの条件が一致する場合、パターンは一致します。

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

警告

デシジョンエンジンは、**starts** オペレーターに対して負の間隔をサポートしていません。負の間隔を使用すると、デシジョンエンジンはエラーを生成します。

started by

このオペレーターは、2 つのイベントが同時に開始するが、現在のイベントの終了前に相関イベントが終了するかどうかを指定します。(このオペレーターの動作は、**starts** オペレーターの動作の逆になります。)
たとえば、以下のパターンは、**$eventA** と **$eventB** が同時に開始し、**$eventB** が **$eventA** の終了前に終了する場合に一致します。

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

以下の方法で、このオペレーターを表すこともできます。

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

started by オペレーターは、2 つのイベントのそれぞれの開始時間の間の最大距離を設定する 1 つのオプションパラメーターをサポートします。

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

これらの条件が一致する場合、パターンは一致します。

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

警告

デシジョンエンジンは、**started by** オペレーターに対して負の間隔をサポートしていません。負の間隔を使用すると、デシジョンエンジンはエラーを生成します。

6.7. デシジョンエンジンにおけるセッションクロックの実装

複雑なイベントの処理中、デシジョンエンジンのイベントには一時的な制約があるかもしれないため、現在の時刻を提供するセッションクロックが必要になります。たとえば、ルールが過去 60 分間の指定の株式の平均価格を決定する必要がある場合、デシジョンエンジンは株価イベントのタイムスタンプとセッションクロックの現在の時刻とを比較できなければなりません。

デシジョンエンジンは、リアルタイムクロックと擬似クロックをサポートしています。シナリオに応じて、1 つまたは両方のクロックタイプを使用できます。

- **ルールのテスト:** テストには管理された環境が必要です。テストに一時的制約を持つルールが含まれる場合、入力ルール、ファクト、および時間のフローを制御できる必要があります。
- **通常の実行:** デシジョンエンジンは、リアルタイムでイベントに反応するので、リアルタイムクロックが必要です。
- **特別な環境:** 特定の環境には、特定の時間制御要件がある場合があります。たとえば、クラスター環境ではクロックの同期が必要な場合があり、Java Enterprise Edition (JEE) 環境ではアプリケーションサーバーが提供するクロックが必要な場合があります。
- **ルールの再生またはシミュレーション:** シナリオを再生またはシミュレートするには、アプリケーションが時間のフローを制御できる必要があります。

デシジョンエンジンで、リアルタイムクロックを使用するか、または擬似クロックを使用するかを決定する際には、環境要件を考慮してください。

リアルタイムクロック

リアルタイムクロックは、デシジョンエンジンのデフォルトのクロック実装であり、システムクロックを使用してタイムスタンプの現在の時刻を決定します。リアルタイムクロックを使用するようにデシジョンエンジンを設定するには、KIE セッション設定パラメーターを **realtime** に設定します。

KIE セッションでのリアルタイムクロックの設定

```
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"));
```

擬似クロック

デシジョンエンジンでの擬似クロックの実装は、一時的なルールのテストに役立ち、アプリケーションによって制御できます。疑似クロックを使用するようにデシジョンエンジンを設定するには、KIE セッション設定パラメーターを **pseudo** に設定します。

KIE セッションでの擬似クロックの設定

```
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"));
```

追加の設定とファクトハンドラーを使用して、擬似クロックを制御することもできます。

KIE セッションで擬似クロックの動作を制御

```
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. イベントストリームとエントリーポイント

デシジョンエンジンは、大量のイベントをイベントストリームの形式で処理できます。 DRL ルール宣言では、ストリームは **エントリーポイント** としても知られています。 DRL ルールまたは Java アプリ

ケーションでエントリーポイントを宣言すると、デシジョンエンジンはコンパイル時に適切な内部構造を特定して作成し、そのエントリーポイントのみからのデータを使用してそのルールを評価します。

1 つのエントリーポイント、またはストリームからのファクトは、デシジョンエンジンのワーキングメモリーにすでにあるファクトに加えて、他のエントリーポイントからのファクトに参加できます。ファクトは、常にデシジョンエンジンに入ったエントリーポイントに関連付けられたままとなっています。同じタイプのファクトは、複数のエントリーポイントからデシジョンエンジンに入ることができますが、エントリーポイント A からデシジョンエンジンに入るファクトは、エントリーポイント B からのパターンと一致することはありません。

イベントストリームには、以下の特徴があります。

- ストリーム内のイベントは、タイムスタンプ別に並べられます。タイムスタンプは、ストリームごとにさまざまなセマンティクスを持つ場合もありますが、常に内部で並べ替えが行われます。
- 通常、イベントストリームには大量のイベントがあります。
- 通常、ストリームに含まれるアトミックなイベントは、それだけでは実用的ではなく、ストリームで集合的な場合にのみ実用的です。
- イベントストリームは、同種 (単一タイプのイベントを含む) の場合も、異種 (異るタイプのイベントを含む) の場合もあります。

6.8.1. ルールデータのエントリーポイントの宣言

イベントのエントリーポイント (イベントストリーム) を宣言して、デシジョンエンジンがそのエントリーポイントのみからのデータを使用してルールを評価することが可能です。エントリーポイントは、DRL ルールで参照することで暗黙的に宣言することも、Java アプリケーションで明示的に宣言することもできます。

手順

以下のいずれかの方法を使用して、エントリーポイントを宣言します。

- DRL ルールファイルで、挿入されたファクトの **from entry-point "<name>"** を指定します。

  "ATM Stream" エントリーポイントで取り消しルールを認可

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

  "Branch Stream" エントリーポイントで手数料ルールを適用

```
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
```

–

銀行取引アプリケーションにおける両方の DRL ルールのサンプルは、イベント **WithdrawalRequest** にファクト **CheckingAccount** を挿入しますが、エントリーポイントは異なります。実行時にデシジョンエンジンは、**"ATM Stream"** エントリーポイントのみからのデータを使用して **Authorize withdrawal** ルールを評価し、**"Branch Stream"** エントリーポイントのみからのデータを使用して **Apply fee** ルールを評価します。**"ATM Stream"** に挿入されたイベントは、**"Apply fee"** ルールのパターンと一致することはありません。また、**"Branch Stream"** に挿入されたイベントは、**"Authorize withdrawal rule"** のパターンと一致することはありません。

- Java アプリケーションコードで、**getEntryPoint()** メソッドを使用して **EntryPoint** オブジェクトを指定および取得し、以下に従ってファクトをそのエントリーポイントに挿入します。

EntryPoint オブジェクトと挿入されたファクトを持つ Java アプリケーションコード

```
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);
```

from entry-point "ATM Stream" を指定する DRL ルールはすべて、このエントリーポイントのデータのみに基づいて評価されます。

6.9. 時間または長さのスライディングウィンドウ

ストリームモードでは、デシジョンエンジンは指定された時間または長さのスライディングウィンドウからのイベントを処理できます。スライディングタイムウィンドウは、イベントを処理できる特定の期間です。スライディングレングスウィンドウは、処理可能なイベントの指定された数です。DRL ルールまたは Java アプリケーションでスライディングウィンドウを宣言すると、デシジョンエンジンは、コンパイル時に適切な内部構造を特定して作成し、そのスライディングウィンドウのみからのデータを使用してそのルールを評価します。

たとえば、以下の DRL ルールスニペットは、最後の 2 分間のストックポイントのみを処理する (スライディングタイムウィンドウ) か、最後の 10 のストックポイントのみを処理する (スライディングレングスウィンドウ) かをデシジョンエンジンに指示します。

過去 2 分間のストックポイントを処理 (スライディングタイムウィンドウ)

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

最後の 10 のストックポイントを処理 (スライディングレングスウィンドウ)

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

6.9.1. ルールデータのスライディングタイムウィンドウを宣言

イベントの時間 (時間のフロー)または長さ (発生回数) のスライディングウィンドウを宣言して、デシジョンエンジンがそのウィンドウのみのデータを使用してルールを評価することが可能です。

手順

DRL ルールファイルで、挿入されたファクトに **over window:<time_or_length>(<value>)** を指定します。

たとえば、以下の 2 つの DRL ルールは、平均温度に基づいて火災報知器を有効にします。ただし、1 番目のルールはスライディングタイムウィンドウを使用して最後の10 分間の平均を計算し、2 番目のルールはスライディングレングスウィンドウを使用して最後の 100 の温度測定値の平均を計算します。

スライディングタイムウィンドウにおける平均温度

```
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
```

スライディングレングスウィンドウにおける平均温度

```
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
```

デシジョンエンジンは、**SensorReading** イベントのうち、10 分を経過したもの、または最後の 100 の読み取り値以外のものをすべて破棄し、時間または読み取り値がリアルタイムで先へと「スライド」していくなか、平均を再計算し続けます。

デシジョンエンジンは、KIE セッションから古いイベントを自動的に削除しません。これは、スライディングウィンドウの宣言がない他のルールが、古いイベントに依存する可能性があるからです。デシジョンエンジンは、明示的なルール宣言または KIE ベースの推論データに基づくデシジョンエンジン内の暗黙的な理由付けのいずれかによって、イベントの有効期限が切れるまで、KIE セッションにイベントを保存します。

6.10. イベントのメモリー管理

ストリームモードでは、デシジョンエンジンは自動メモリー管理を使用して、KIE セッションに保存されているイベントを維持します。デシジョンエンジンは、一時的な制約が原因でルールと一致しなくなったイベントを KIE セッションから取り除き、取り除かれたイベントが使っていたリソースを解放します。

デシジョンエンジンは、明示的な有効期限または推論された有効期限のいずれかを使用して、古いイベントを取り消します。

- **明示的な有効期限:** デシジョンエンジンは、**@expires** タグを宣言するルールで明示的に有効期限が切れるよう設定されているイベントを削除します。

  有効期限が明示的な DRL ルールスニペット

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

  この例のルールは、**StockPoint** イベントが 30 分後に有効期限となるよう設定し、他のルールがイベントを使用しない場合は KIE セッションから削除するよう設定します。

- **推論された有効期限:** デシジョンエンジンは、ルールの一時的な制約を解析することで、特定のイベントに関する有効期限の補正値を暗黙的に算出できます。

  一時制約のある DRL ルール

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

  この例のルールでは、デシジョンエンジンは、**BuyOrder** イベントが発生するたびに、デシジョンエンジンがイベントを最大 10 秒間保存し、一致する **AckOrder** イベントを待つ必要があることを自動的に計算します。10 秒後に、デシジョンエンジンは有効期限を推論し、KIE セッションからイベントを削除します。**AckOrder** イベントは既存の **BuyOrder** イベントとのみ一致するため、一致が発生しない場合はデシジョンエンジンが有効期限を推論し、イベントをすぐに削除します。

  デシジョンエンジンは、KIE ベース全体を分析して、すべてのイベントタイプの補正値を見つけ、他のルールが削除を保留しているイベントを使用しないようにします。暗黙的な有効期限が明示的な有効期限の値と衝突するたびに、デシジョンエンジンはこの 2 つのうちの大きい方の時間枠を使用して、イベントをより長く保存します。

第7章 デシジョンエンジンクエリーおよびライブクエリー

デシジョンエンジンでクエリーを使用して、ルールで使用されるファクトパターンに基づいてファクトセットを取得できます。また、パターンはオプションのパラメーターを使用することもできます。

デシジョンエンジンでクエリーを使用するには、DRL ファイルにクエリー定義を追加し、アプリケーションコードで一致する結果を取得します。クエリーは結果コレクション上で反復しますが、クエリーにバインドされている識別子を使用して、バインディング変数名を引数として使用する **get()** メソッドを呼び出すことで、対応するファクトまたはファクトフィールドにアクセスできます。バインディングがファクトオブジェクトを参照する場合、変数名をパラメーターとして使用する **getFactHandle()** を呼び出すことで、ファクトハンドルを取得できます。

DRL ファイルにおけるクエリー定義の例

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

クエリー結果を取得および反復するアプリケーションのコード例

```
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" );
}
```

経時的な変化をモニタリングする場合には、クエリーを呼び出して返された値のセットを反復して結果を処理することが困難な場合があります。進行中のクエリーでこの困難な作業を軽減するために、Red Hat Process Automation Manager は **ライブクエリー** を提供しています。これは、反復可能な結果セットを返すのではなく、変更イベントに対してアタッチされたリスナーを使用します。ライブクエリーは、ビューを作成し、このビューのコンテンツ向けに変更イベントを公開することで、オープンの状態を保ちます。

ライブクエリーをアクティブ化するには、パラメーターを使用してクエリーを開始し、結果ビューの変更を監視します。**dispose()** メソッドを使用してクエリーを終了し、このリアクティブシナリオを中断できます。

DRL ファイルにおけるクエリー定義の例

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

イベントリスナーとライブクエリーを使用したアプリケーションのコード例

```
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()
```

第8章 デシジョンエンジンのイベントリスナーおよびデバッグロギング

Red Hat Process Automation Manager では、ファクトの挿入やルールの実行など、デシジョンエンジンのイベントリスナーを追加または削除できます。デシジョンエンジンのイベントリスナーを使用すると、デシジョンエンジンのアクティビティーの通知を受けることができ、ロギングと監査をアプリケーションのコアから分けることができます。

デシジョンエンジンは、アジェンダおよびワーキングメモリーに対して、以下のデフォルトのイベントリスナーをサポートします。

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

各イベントリスナーについて、デシジョンエンジンは、監視するように指定できる以下の特定のイベントもサポートします。

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

たとえば、以下のコードは、KIE セッションにアタッチされた **DefaultAgendaEventListener** リスナーを使用して、**AfterMatchFiredEvent** イベントが監視されるように指定します。コードは、ルールの実行後にパターンの一致を出力します。

アジェンダの AfterMatchFiredEvent イベントを監視および出力するコード例

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

デシジョンエンジンは、デバッグロギングに対して以下のアジェンダおよびワーキングメモリーイベントリスナーもサポートします。

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

これらのイベントリスナーは、同様のサポートされているイベントリスナーメソッドを実装し、デフォルトでデバッグ出力ステートメントを含んでいます。特定のサポートされるイベントを追加して監視およびドキュメント化するか、すべてのアジェンダまたはワーキングメモリーアクティビティーを監視することができます。

たとえば、以下のコードは **DebugRuleRuntimeEventListener** イベントリスナーを使用して、すべてのワーキングメモリーイベントを監視および出力します。

すべてのワーキングメモリーイベントを監視および出力するコード例

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

8.1. デシジョンエンジンでのロギングユーティリティーの設定

デシジョンエンジンは、システムロギングに Java ロギング API SLF4J を使用します。以下のロギングユーティリティーのいずれかをデシジョンエンジンで使用して、トラブルシューティングまたはデータ収集などのデシジョンエンジンのアクティビティーを調査できます。

- Logback
- Apache Commons Logging
- Apache Log4j
- **java.util.logging** パッケージ

手順

使用するロギングユーティリティーについては、Maven プロジェクトに関連する依存関係を追加するか、Red Hat Process Automation Manager ディストリビューションの **org.drools** パッケージに関連する XML 設定ファイルを保存します。

Logback の Maven 依存関係の例

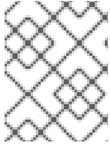
```
<dependency>
  <groupId>ch.qos.logback</groupId>
  <artifactId>logback-classic</artifactId>
  <version>${logback.version}</version>
</dependency>
```

org.drools パッケージにおける logback.xml 設定ファイルの例

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

org.drools パッケージにおける log4j.xml 設定ファイルの例

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



注記

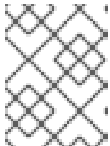
超軽量環境向けに開発している場合、**slf4j-nop** または **slf4j-simple** ロガーを使用します。

第9章 RED HAT PROCESS AUTOMATION MANAGER の IDE 向けのデシジョン例

Red Hat Process Automation Manager は、統合開発環境 (IDE: integrated development environment) にインポートできるように Java クラスとして配信される、デシジョン例を提供します。これらの例は、デシジョンエンジン機能をさらに理解するために使用する目的か、Red Hat Process Automation Manager プロジェクトに定義するデシジョンの参考として使用してください。

以下のデシジョンセットの例は、Red Hat Process Automation Manager で利用可能な例の一部です。

- **Hello World の例**: 基本的なルール実行や、デバッグ出力の使用方法を例示します。
- **状態の例**: ルールの顕著性やアジェンダグループを使用した前向き連鎖や競合解決を例示します。
- **フィボナッチの例**: ルールの顕著性を使用した再帰や競合解決を例示します。
- **銀行の例**: パターン一致、基本的なソート、計算を例示します。
- **ペットショップの例**: ルールアジェンダグループ、グローバル変数、コールバック、GUI 統合を例示します。
- **数独の例**: 複雑なパターン一致、問題解決、コールバック、GUI 統合を例示します。
- **House of Doom の例**: 後向き連鎖と再帰を例示します。



注記

Red Hat Business Optimizer で提供される最適化の例については、『Red Hat Business Optimizer のスタートガイド』を参照してください。

9.1. IDE での RED HAT PROCESS AUTOMATION MANAGER のデシジョン例のインポートと実行

Red Hat Process Automation Manager のデシジョン例を統合開発環境 (IDE) にインポートして実行し、ルールとコードがどのように機能するかチェックできます。これらの例は、デシジョンエンジン機能をさらに理解するために使用するか、Red Hat Process Automation Manager プロジェクトに定義するデシジョンの参考として使用してください。

前提条件

- Java 8 以降をインストールしている。
- Maven 3.5.x 以降をインストールしていること
- Red Hat CodeReady Studio などの IDE がインストールされている。

手順

1. Red Hat カスタマーポータル から **Red Hat Process Automation Manager 7.8.0 Source Distribution** を **/rhpam-7.8.0-sources** などの一時的なディレクトリーにダウンロードして展開します。

2. IDE を開き、**File → Import → Maven → Existing Maven Projects** を選択するか、同等のオプションを選択して、Maven プロジェクトをインポートします。

3. **Browse** をクリックして、**~/rhpam-7.8.0-sources/src/drools-$VERSION/drools-examples** (または、Conway の Game of Life の例の場合は、**~/rhpam-7.8.0-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples**) に移動して、プロジェクトをインポートします。

4. 実行するパッケージ例に移動して、**main** メソッドが含まれる Java クラスを検索します。

5. Java クラスを右クリックし、**Run As → Java Application** を選択して例を実行します。
   基本的なユーザーインターフェースですべての例を実行するには、**org.drools.examples** Main クラスの **DroolsExamplesApp.java** クラス (または Conway の Game of Life の場合は **DroolsJbpmIntegrationExamplesApp.java** クラス) を実行します。

図9.1 drools-examples (DroolsExamplesApp.java) 内のすべての例のインターフェース

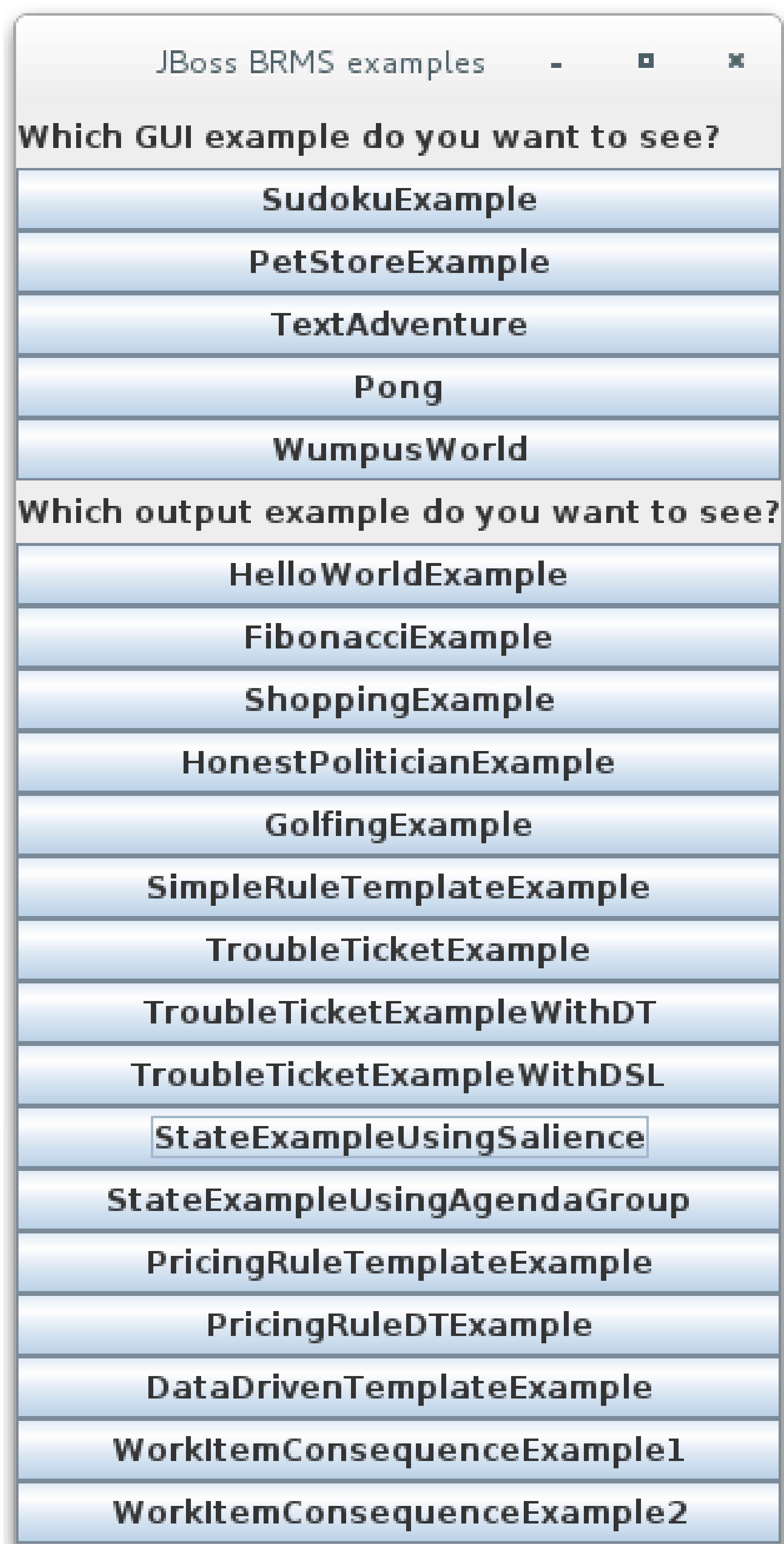
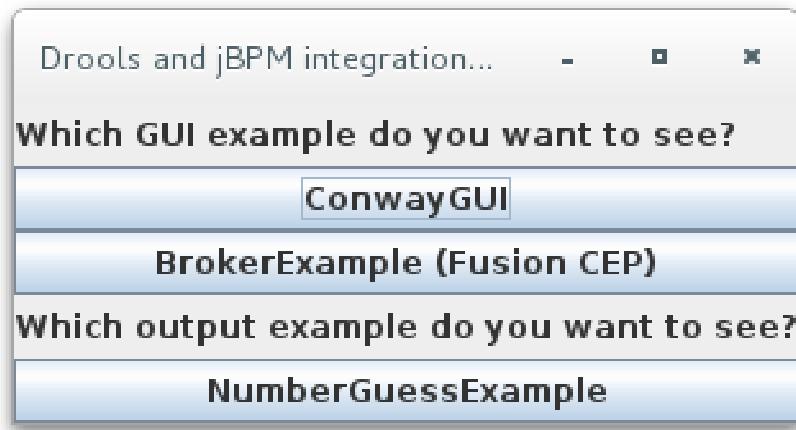


図9.2 droolsjbpm-integration-examples (DroolsJbpmIntegrationExamplesApp.java) のすべての例のインターフェース



9.2. HELLO WORLD のデシジョン例 (基本ルールおよびデバッグ)

Hello World のデシジョンセットの例では、オブジェクトをデシジョンエンジンのワーキングメモリーに挿入する方法、ルールを使用してオブジェクトを照合する方法、エンジンの内部アクティビティーを追跡するロギングの設定方法を例示します。

以下は、Hello World の例の概要です。

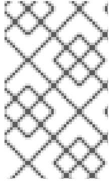
- **名前**: **helloworld**
- Main class: **org.drools.examples.helloworld.HelloWorldExample** (in **src/main/java**)
- **モジュール**: **drools-examples**
- **タイプ**: Java アプリケーション
- Rule file: **org.drools.examples.helloworld.HelloWorld.drl** (in **src/main/resources**)
- **目的**: 基本的なルール実行とデバッグ出力の使用方法を例示します。

Hello World の例では、KIE セッションが生成されて、ルールの実行が可能になります。すべてのルールは、実行できるように KIE セッションが必要です。

ルール実行の KIE セッション

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

1. **KieServices** ファクトリーを取得します。これは、アプリケーションがデシジョンエンジンとの対話に使用する主なインターフェースです。
2. プロジェクトクラスパスから **KieContainer** を作成します。これで、**/META-INF/kmodule.xml** ファイルを検出し、このファイルをもとに設定して **KieModule** で **KieContainer** をインスタンス化します。
3. **/META-INF/kmodule.xml** ファイルに定義された **"HelloWorldKS"** KIE セッション設定をもとに **KieSession** を作成します。



注記

Red Hat Process Automation Manager プロジェクトのパッケージ化に関する詳細は、『Red Hat Process Automation Manager プロジェクトのパッケージ化およびデプロイ』を参照してください。

Red Hat Process Automation Manager には、内部エンジンアクティビティーを公開するイベントモデルがあります。**DebugAgendaEventListener** と **DebugRuleRuntimeEventListener** のデフォルトのデバッグリスナー 2 つにより、デバッグイベント情報が **System.err** の出力に表示されます。**KieRuntimeLogger** では実行監査が提供され、その結果はグラフィックビューワーで確認できます。

リスナーと監査ロガーのデバッグ

```
// 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 );
```

ロガーは、**Agenda** と **RuleRuntime** リスナーにビルドされる特別な実装です。デシジョンエンジンが実行を終えると、**logger.close()** が呼び出されます。

この例では、**"Hello World"** というメッセージを含む **Message** オブジェクトを作成し、ステータス **HELLO** を **KieSession** に挿入して、**fireAllRules()** でルールを実行します。

データの挿入および実行

```
// 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();
```

ルール実行は、データモデルを使用して、**KieSession** への出入力としてデータを渡します。この例のデータモデルには **message** (**String**) と **status** (**HELLO** または **GOODBYE**) の2 つのフィールドが含まれます。

データモデルクラス

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

    private String        message;
```

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

この 2 つのルールは、**src/main/resources/org/drools/examples/helloworld/HelloWorld.drl** ファイルに配置されます。

"Hello World" ルールの **when** 条件では、ステータスが **Message.HELLO** の KIE セッションに、**Message** オブジェクトが挿入されるたびに、このルールを有効化すると記述しています。さらに、変数のバインドが 2 つ作成されます (**message** 変数を **message** 属性に、**m** 変数を一致する **Message** オブジェクト自体にバインド)。

ルールの **then** アクションは、バインドされた変数 **message** のコンテンツを **System.out** に出力するよう指定し、続いて **m** にバインドされている **Message** オブジェクトの **message** と **status** 属性値を変更します。このルールは **modify** ステートメントを使用して、1 つのステートメントに割り当てブロックを適用し、ブロックの最後にデシジョンエンジンにこの変更について通知します。

"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
```

"Good Bye" ルールは、ステータスが **Message.GOODBYE** の **Message** オブジェクトと一致する点を除き、**"Hello World"** ルールによく似ています。

"Good Bye" rule

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

この例を実行するには、**org.drools.examples.helloworld.HelloWorldExample** クラスを IDE で Java アプリケーションとして実行します。このルールは **System.out** に、デバッグリスナーは **System.err** に書き込み、監査ロガーは **target/helloworld.log** のログファイルを作成します。

IDE コンソールの System.out 出力

```
Hello World
Goodbye cruel world
```

IDE コンソールでの System.err の出力

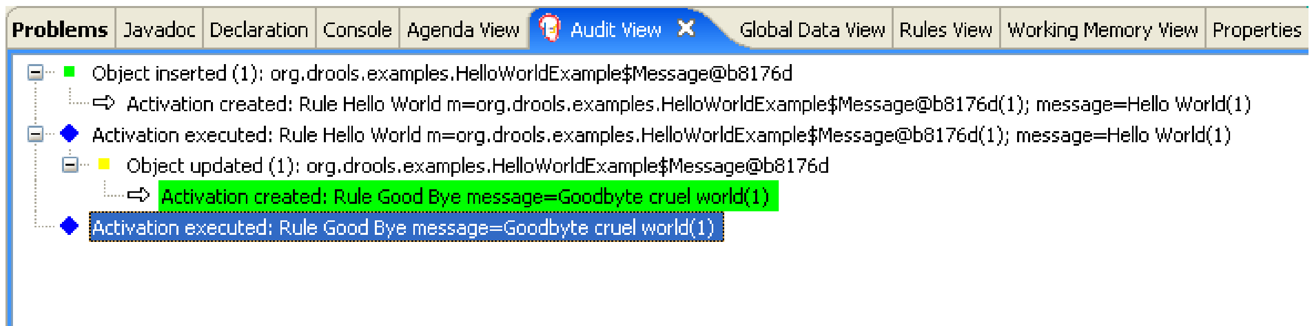
```
==>[ActivationCreated(0): rule=Hello World;
                   tuple=[fid:1:1:org.drools.examples.helloworld.HelloWorldExample$Message@17cec96]]
[ObjectInserted: handle=
```

```
[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]
```

この例の実行フローをさらに理解するには、**target/helloworld.log** からの監査ログファイルを IDE デバッグビューまたは **Audit View** が利用できる場合は Audit View (例: IDE の **Window → Show View**) にロードします。

この例では、**Audit view** で、オブジェクトが挿入され、**"Hello World"** ルールのアクティベーションが作成されます。次に、このアクティベーションが実行され、**Message** オブジェクトを更新して、**"Good Bye"** ルールのアクティベーションをトリガーします。最後に、**"Good Bye"** ルールが実行されます。**Audit View** でインベントが選択されると、この例の **"Activation created"** イベントである元のイベントが緑色にハイライトされます。

図9.3 Hello World の例の監査ビュー



9.3. 状態のデシジョン例 (前向き連鎖および競合解決)

状態のデシジョンセットの例では、デシジョンエンジンが、前向き連鎖およびワーキングメモリー内のファクトへの変更を使用して、ルールの実行競合を順番に解決していく方法を説明しています。この例では、ルールで定義可能な顕著性の値またはアジェンダグループを使用して競合を解決することにフォーカスしています。

以下は、状態の例の概要です。

- **名前**: **state**
- Main クラス: (**src/main/java** 内の) **org.drools.examples.state.StateExampleUsingSalience**、**org.drools.examples.state.StateExampleUsingAgendaGroup**
- **モジュール**: **drools-examples**
- **タイプ**: Java アプリケーション

- **ルールファイル**: (**src/main/resources** 内の) **org.drools.examples.state.*.drl**
- **目的**: ルールの顕著性やアジェンダグループを使用した前向き連鎖や競合解決を例示します。

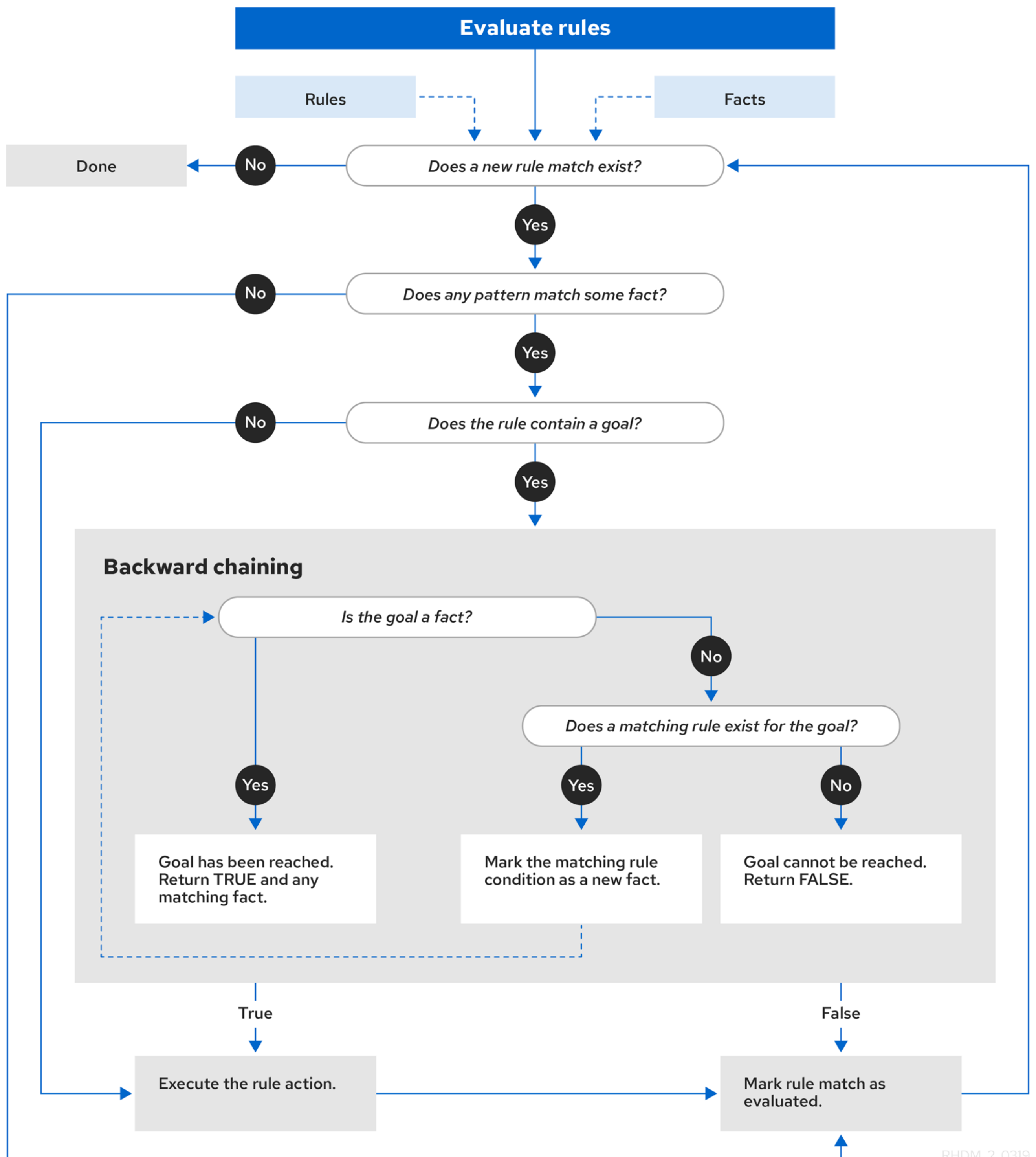
前向き連鎖のルールシステムは、デシジョンエンジンのワーキングメモリーにあるファクトで開始して、そのファクトへの変更に反応するデータ駆動型のシステムです。オブジェクトがワーキングメモリーに挿入されると、その変更の結果として True となってルールの条件が、アジェンダにより実行がスケジュールされます。

反対に、後向き連鎖のルールシステムは、通常再帰を使用して、デシジョンエンジンが満たそうとする結論から開始する目的駆動型のシステムです。システムが結論または目的に到達できない場合には、サブとなる目的、つまり、現在の目的の一部を完了する結論を検索します。システムは、最初の結論が満たされるか、すべてのサブとなる目的が満たされるまで続行されます。

Red Hat Process Automation Manager のデシジョンエンジンは、前向き連鎖と後向き連鎖の両方を使用してルールを評価します。

以下の図は、デシジョンエンジンが、ロジックフローで後向き連鎖のセグメントと、前向き連鎖全体とを使用してルールを評価する方法を例示します。

図9.4 前向き連鎖と後向き連鎖を使用したルール評価のロジック



状態の例では、**State** クラスごとに、名前や現在の状態のフィールドが含まれます (**org.drools.examples.state.State** のクラス参照)。以下の状態は、各プロジェクトで考えられる状態 2 つです。

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

State クラス

```
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...
}
```

状態の例には、同じ例が 2 つのバージョンとして提供されており、それぞれルール実行の競合を解決します。

- ルールの顕著性を使用して競合を解決する **StateExampleUsingSalience** バージョン
- ルールアジェンダグループを使用して競合を解決する **StateExampleUsingAgendaGroups** バージョン

状態の例のバージョンはいずれも、**A**、**B**、**C**、**D** の 4 つの **State** オブジェクトを使用します。最初に、それぞれの状態は、**NOTRUN** に設定されます。NOTRUN は、例が使用するコンストラクターのデフォルト値です。

顕著性を使用した状態の例

状態の例の **StateExampleUsingSalience** バージョンでは、ルールで顕著性の値を使用し、ルール実行の競合を解決します。顕著性の値が高いルールは、アクティベーションキューの順番で、優先度が高くなります。

この例では、各 **State** インスタンスを KIE セッションに挿入して、**fireAllRules()** を呼び出します。

顕著性の状態例の実行

```
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();
```

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.state.StateExampleUsingSalience** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます。

IDE コンソールでの顕著性の状態例の出力

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

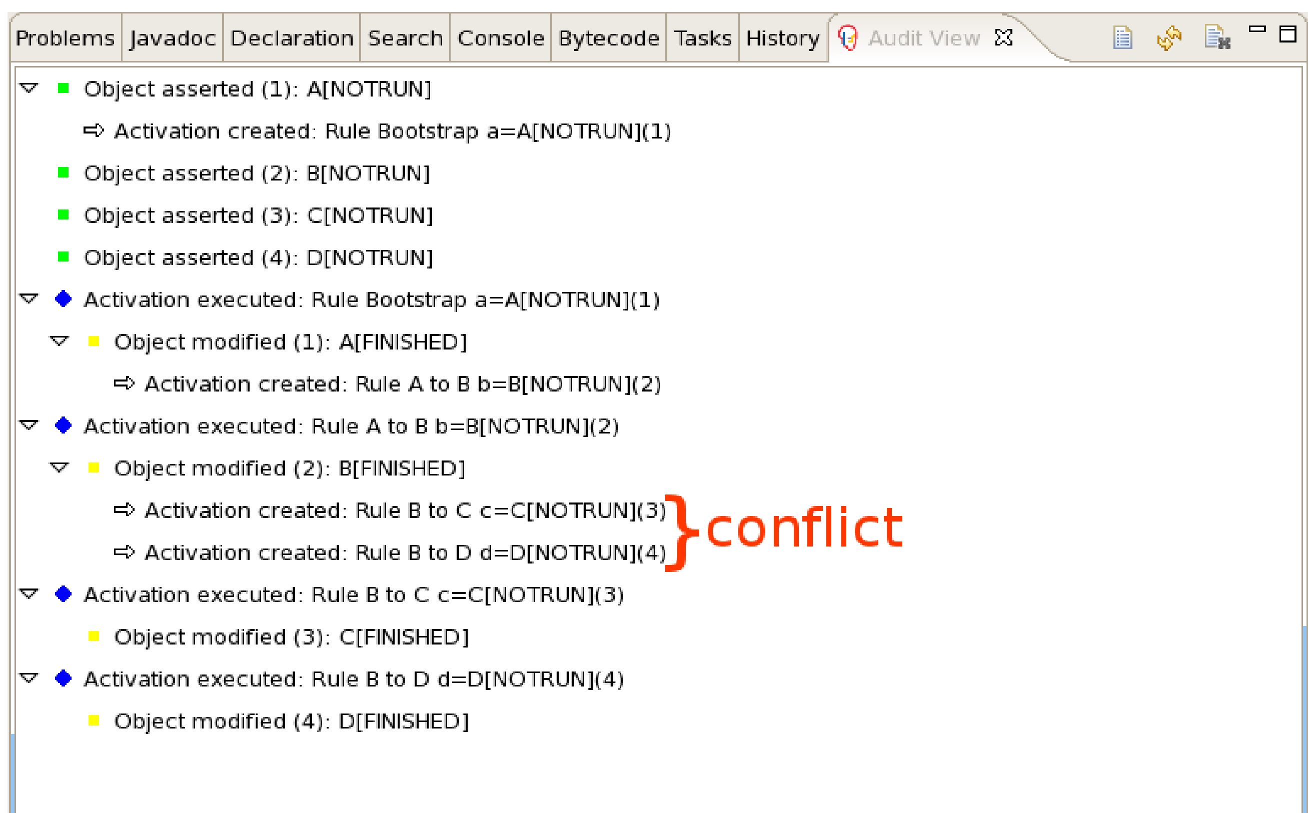
4 つのルールが存在します。

まず、**"Bootstrap"** ルールが実行され、**A** の状態が **FINISHED** に設定されます。次に、**B** の状態が **FINISHED** に変更され、オブジェクト **C** と **D** はいずれも **B** に依存するため、競合が発生しますが、顕著性の値で解決されます。

この例の実行フローをさらに理解するには、**target/state.log** からの監査ログファイルを IDE デバッグビューまたは **Audit View** が利用できる場合は Audit View (例: IDE の **Window → Show View**) にロードします。

この例では、**Audit View** は、状態が **NOTRUN** のオブジェクト **A** のアサーションが **"Bootstrap"** ルールをアクティベートしますが、他のオブジェクトのアサーションはすぐに有効になりません。

図9.5 顕著性の状態例の監査ビュー



顕著性の状態例の "Bootstrap" ルール

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

"Bootstrap" ルールを実行すると、**A** の状態が **FINISHED** に変わり、ルール **"A to B"** をアクティベートします。

顕著性の状態例の "A to B" ルール

```
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
```

"A to B" ルールを実行すると、**B** の状態を **FINISHED** に変更し、**"B to C"** と **"B to D"** 両方のルールをアクティベートして、これらのアクティベーションをデシジョンエンジンアジェンダに配置します。

顕著性の状態例の "B to C" および "B to D" ルール

```
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
```

この時点から、両方のルールが実行される可能性があるため、これらのルールは競合しています。競合解決ストラテジーを使用すると、デシジョンエンジンアジェンダがどのルールを実行するかを決定できます。**"B to C"** は、顕著性の値が高い (**10** と、デフォルトの顕著性の値 **0**) ので、先に実行され、オブジェクト **C** の状態が **FINISHED** に変更されます。

IDE の **Audit View** では、ルール **"A to B"** の **State** オブジェクトが変更され、2 つのアクティベーションが競合する結果になることが分かります。

IDE で **Agenda View** を使用して、デシジョンエンジンアジェンダの状態を調査できます。この例では **Agenda View** で、ルール **"A to B"** のブレークポイントと、2 つの競合するルールを持つアジェンダの状態が分かります。最後にルール **"B to D"** が実行され、オブジェクト **D** の状態が **FINISHED** に変更されます。

図9.6 顕著性の状態例のアジェンダビュー

StateExampleUsingSalience.java　StateExampleUsingSalience.drl

```
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

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
```

Text Editor　Rete Tree

Console　Tasks　Agenda View　Audit View　Global Data View　Rules View　Working Memory View

```
MAIN[focus]= BinaryHeapQueueAgendaGroup (id=1392)
  [0]= Activation
    ruleName= "B to C"
    c= State (id=1406)
      FINISHED= 1
      NOTRUN= 0
      changes= PropertyChangeSupport (id=1433)
      name= "C"
      state= 0
  [1]= Activation
    ruleName= "B to D"
    c= State (id=1406)
      FINISHED= 1
      NOTRUN= 0
      changes= PropertyChangeSupport (id=1433)
      name= "C"
      state= 0
```

アジェンダグループを使用した状態の例

状態の例の **StateExampleUsingAgendaGroups** バージョンでは、ルールでアジェンダグループを使用し、ルール実行における競合を解決します。アジェンダグループを使用すると、デシジョンエンジンアジェンダが分割され、ルールのグループの実行に対してこれまで以上に制御ができるようになります。デフォルトでは、ルールはすべてアジェンダグループ **MAIN** に含まれています。**agenda-group** 属性を使用して、ルールに異なるアジェンダグループを指定できます。

最初は、ワーキングメモリーは、アジェンダグループ **MAIN** にフォーカスを当てます。アジェンダグループのルールは、グループがこのフォーカスを受けた場合のみ実行されます。**setFocus()** のメソッドか、**auto-focus** のルール属性を使用してフォーカスを設定できます。**auto-focus** 属性を使用すると、ルールが一致してアクティベートされた場合のみ、ルールにアジェンダグループのフォーカスが自動的に当てられます。

この例では、**auto-focus** 属性を使用すると **"B to D"** の前に **"B to C"** ルールを実行できます。

アジェンダグループの状態例のルール "B to C"

```
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
```

ルール **"B to C"** は、アジェンダグループ **"B to D"** の **setFocus()** を呼び出し、有効なルールを実行できるようにします。その後にルール **"B to D"** が実行できるようになります。

アジェンダグループの状態例のルール "B to D"

```
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
```

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.state.StateExampleUsingAgendaGroups** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます (状態の例の顕著性バージョンと同じ)。

IDE コンソールでのアジェンダグループの状態例の出力

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

状態の例に含まれる動的なファクト

状態の例に含まれる主なコンセプトとして他には、**PropertyChangeListener** オブジェクトを実装するオブジェクトをもとに **動的ファクト** を使用するというものがあります。デシジョンエンジンがファクトプロパティーへの変更を確認し、対応するためには、アプリケーションがデシジョンエンジンに対し

て、変更があったことを通知する必要があります。**modify** ステートメントを使用して、このコミュニケーションをルールで明示的に設定するか、JavaBeans 使用で定義されているようにファクトが **PropertyChangeSupport** インターフェースを実装するように指定することで暗黙的に設定できます。

この例は、ルールで **modify** ステートメントを明示的に指定しなくても良いように **PropertyChangeSupport** インターフェースを使用する方法が示されています。このインターフェースを使用するには、**org.drools.example.State** クラスと同じ方法で、ファクトに **PropertyChangeSupport** が実装されていることを確認し、DRL ルールファイルで以下のコードを使用して、これらのファクトでプロパティー変更がないかをリッスンするようにデシジョンエンジンを設定してください。

動的ファクトの宣言

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

PropertyChangeListener オブジェクトを使用する場合に、各セッターは通知用に追加のコードを実装する必要があります。たとえば、**state** の以下のセッターは **org.drools.examples** のクラスに含まれます。

PropertyChangeSupport のセッター例

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

9.4. フィボナッチのデシジョン例 (再帰および競合解決)

フィボナッチのデシジョンセットの例では、デシジョンエンジンが再帰を使用してルールの実行競合を順番に解決していく方法を説明しています。この例では、ルールで定義可能な顕著性の値を使用して競合を解決することにフォーカスしています。

以下は、フィボナッチの例の概要です。

- **名前**: **フィボナッチ**
- Main **クラス**: (**src/main/java** 内の) **org.drools.examples.fibonacci.FibonacciExample**
- **モジュール**: **drools-examples**
- **タイプ**: Java アプリケーション
- **ルールファイル**: (**src/main/resources** 内の) **org.drools.examples.fibonacci.Fibonacci.drl**
- **目的**: ルールの顕著性を使用した再帰や競合解決を例示します。

フィボナッチ数は、0 または 1 で開始する数列です。0、1、1、2、3、5、8、13、21、34、55、89、144、233、377、610、987、1597、2584、4181、6765、10946 などのように、2 つの先行する数を足すことにより、次にくるフィボナッチ数が求められます。

フィボナッチの例では、**Fibonacci** のファクトクラスを 1 つ使用し、このクラスに以下のフィールド 2 つが含まれています。

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

sequence フィールドは、フィボナッチ数列のオブジェクトの位置を示します。**value** フィールドは、その数列の位置のフィボナッチオブジェクトの値を示します。**-1** は、計算する必要がある値という意味です。

フィボナッチクラス

```
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...
}
```

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.fibonacci.FibonacciExample** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます。

IDE コンソールでのフィボナッチの例の出力

```
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
```

Java でこの動作を実現するには、sequence フィールドに **50** を指定して、**Fibonacci** オブジェクトを挿入します。この例では、次に再帰ルールを使用して、他の 49 個の **Fibonacci** オブジェクトを挿入します。

PropertyChangeSupport インターフェースを実装して動的ファクトを使用する代わりに、このレでは MVEL 方言の **modify** キーワードを使用して、ブロックセッターアクションを有効にしてデシジョンエンジンに変更を通知しています。

フィボナッチの例の実行

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

この例では、以下の 3 つのルールを使用します。

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

"Recurse" ルールは、値が **-1** の、アサートされた各 **Fibonacci** オブジェクトを照合して、現在の値よりも数列が 1 つ小さい **Fibonacci** オブジェクトを新たに作成し、アサートします。数列フィールドが **1** に相当するオブジェクトが存在しない場合に、フィボナッチオブジェクトが追加されると毎回、このルールは再度照合、実行されます。メモリーにフィボナッチオブジェクト 50 個すべてが存在する場合には、**not** 条件要素を使用して、ルールの合致を停止します。また、**"Bootstrap"** ルールを実行する前に **Fibonacci** オブジェクト 50 個すべてをアサートする必要があるので、このルールには **salience** の値も含まれます。

ルール "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
```

この例の実行フローをさらに理解するには、**target/fibonacci.log** からの監査ログファイルを IDE デバッグビューまたは **Audit View** が利用できる場合は Audit View (例: IDE の **Window → Show View**) にロードします。

この例では、**監査ビュー** に、 **sequence** フィールドが **50** に指定された、**Fibonacci** の元のアサーションが表示されます。これは Java コードで実行されています。これ以降、**監査ビュー** で、ルールの再帰が継続して行われ、アサートされた **Fibonacci** オブジェクトにより、**"Recurse"** ルールがアクティベートされて、再度実行されます。

図9.7 監査ビューでのルール "Recurse"

Problems Javadoc Declaration Search Console Error Log History Audit View Properties

Object asserted (1): Fibonacci(50/-1)
Activation created: Rule Recurse f=Fibonacci(50/-1)(1)
Activation executed: Rule Recurse f=Fibonacci(50/-1)(1)
Object asserted (2): Fibonacci(49/-1)
Activation created: Rule Recurse f=Fibonacci(49/-1)(2)
Activation executed: Rule Recurse f=Fibonacci(49/-1)(2)
Object asserted (3): Fibonacci(48/-1)
Activation created: Rule Recurse f=Fibonacci(48/-1)(3)
Activation executed: Rule Recurse f=Fibonacci(48/-1)(3)
Object asserted (4): Fibonacci(47/-1)
Activation created: Rule Recurse f=Fibonacci(47/-1)(4)
Activation executed: Rule Recurse f=Fibonacci(47/-1)(4)
Object asserted (5): Fibonacci(46/-1)
Activation created: Rule Recurse f=Fibonacci(46/-1)(5)
Activation executed: Rule Recurse f=Fibonacci(46/-1)(5)
Object asserted (6): Fibonacci(45/-1)
Activation created: Rule Recurse f=Fibonacci(45/-1)(6)
Activation executed: Rule Recurse f=Fibonacci(45/-1)(6)
Object asserted (7): Fibonacci(44/-1)
Activation created: Rule Recurse f=Fibonacci(44/-1)(7)

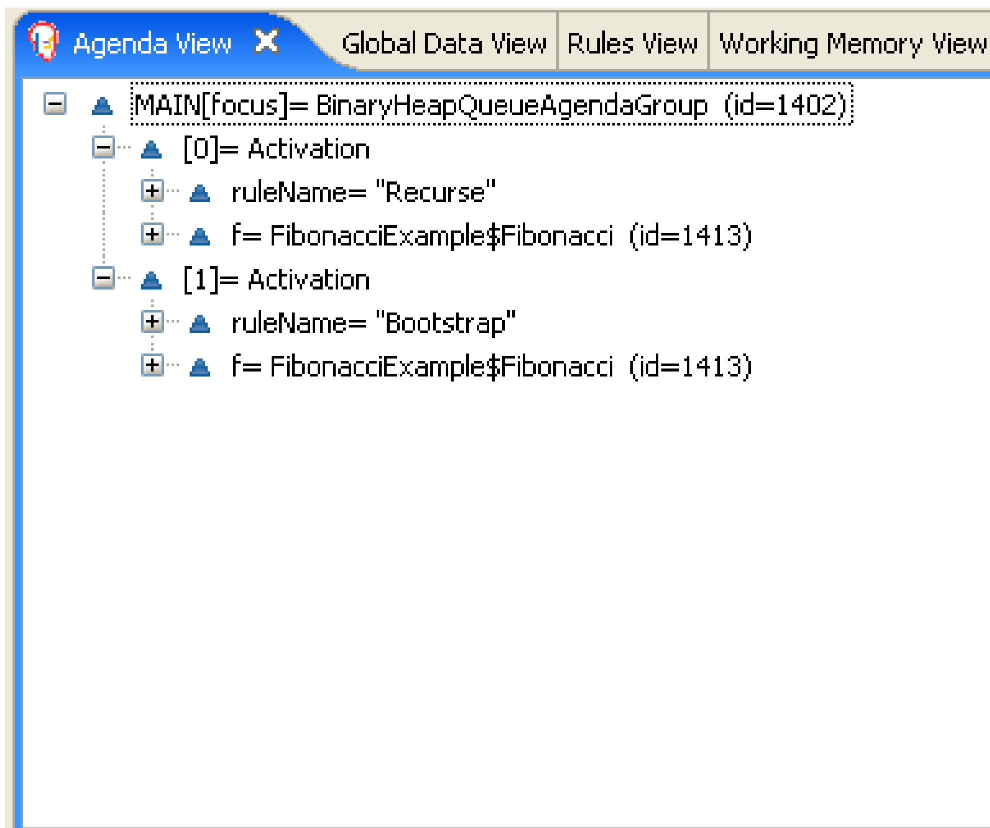
sequence フィールドが **2** の **Fibonacci** オブジェクトがアサートされると、**"Bootstrap"** ルールが一致し、**"Recurse"** ルールとともにアクティベートされます。フィールド**sequence** には、複数の制約があり、**1** または **2** と同等かをテストしている点に注目してください。

ルール "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
```

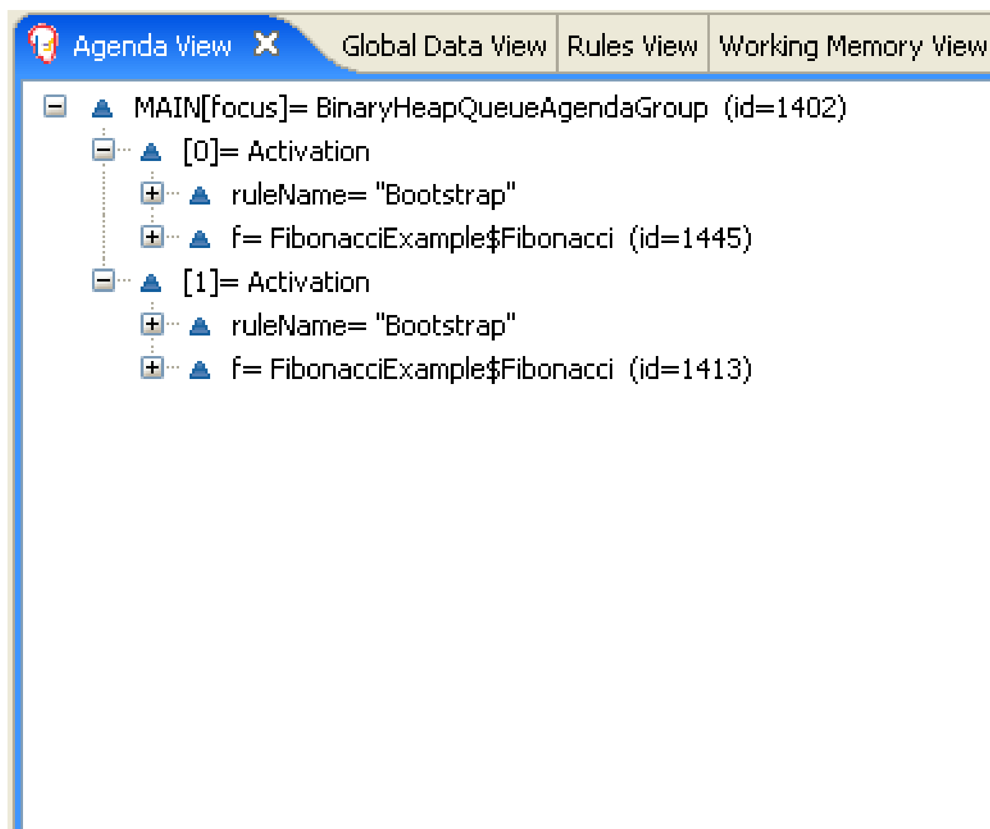
IDE で **Agenda View** を使用して、デシジョンエンジンアジェンダの状態を調査できます。**"Recurse"** の顕著性の値のほうが高いので、**"Bootstrap"** ルールはまだ実行されません。

図9.8 アジェンダビュー 1 でのルール "Recurse" および "Bootstrap"



sequence が **1** の **Fibonacci** オブジェクトがアサートされると、**"Bootstrap"** ルールが再度一致し、このルールに含まれる 2 つのルールがアクティベートされます。**sequence** が **1** の **Fibonacci** オブジェクトが存在すると、すぐに **not** 条件要素で、ルールが一致しなくなるので、**"Recurse"** ルールの照合やアクティベーションはされません。

図9.9 アジェンダビュー 2 でのルール "Recurse" および "Bootstrap"



"Bootstrap" ルールは、**sequence** が **1** と **2** のオブジェクトの値を **1** に設定します。値が **-1** でない **Fibonacci** オブジェクトが 2 つあるので、**"Calculate"** ルールの照合が可能になります。

この例のある時点で、ワーキングメモリーに 50 近くの **Fibonacci** オブジェクトが存在します。3 つ選択してそれぞれを乗算し、順番に各値を計算する必要があります。フィールドの制約なしに、ルールで Fibonacci パターン 3 つを使用してクラス積候補を絞り込む場合に、考えられる組み合わせとして 50x49x48 通りあり、約 12 万 5000 のルールを実行できるにもかかわらず、その大半が誤っていることになります。

"Calculate" ルールは、フィールドの制約を使用して正しい順番にフィボナッチパターン 3 つを評価します。この手法は **cross-product matching** と呼ばれます。

最初のパターンでは、値が **!= -1** の **Fibonacci** オブジェクトを検索して、このパターンとフィールド両方をバインドします。2 番目の **Fibonacci** オブジェクトの実行する内容は同じですが、別のフィールド制約を追加して、シーケンスが **f1** にバインドされている **Fibonacci** オブジェクトより 1 つ大きくなるようにします。このルールが初めて実行されると、シーケンスが **1** と **2** にだけ、値 **1** が割り当てられていることが分かります。また、この 2 つの制約で、**f1** がシーケンス **1** を、**f2** がシーケンス **2** を参照するようにします。

最後のパターンでは、値が **-1** と等しく、シーケンスが **f2**よりも大きい **Fibonacci** オブジェクトを検索します。

フィボナッチの例のこの時点で、3 つの **Fibonacci** オブジェクトが利用可能なクロス積から正しく選択され、**f3** にバインドされている 3 番目の **Fibonacci** オブジェクトの値を計算できます。

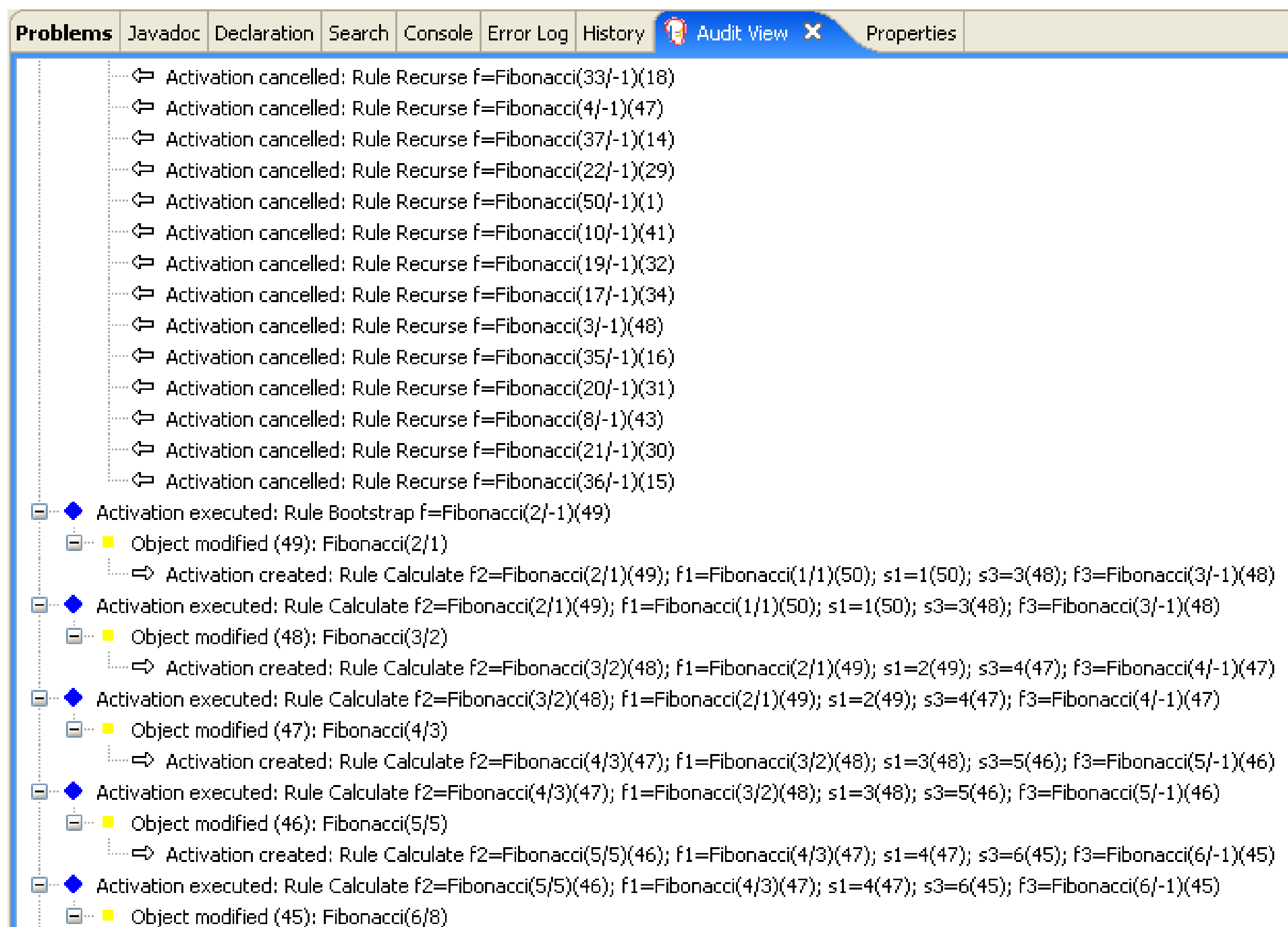
ルール "Calculate"

```
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
```

modify ステートメントにより、**f3** にバインドされた **Fibonacci** オブジェクトの値が更新されます。つまり、値が **-1** 以外の **Fibonacci** オブジェクトが新たに存在するということで、**"Calculate"** ルールにより、再度合致があるか検索して次のフィボナッチ番号を算出することができます。

IDE のデバッグビューまたは **監査ビュー** では、最後の **"Bootstrap"** ルールが実行されることで **Fibonacci** オブジェクトが変更され、**"Calculate"** ルールに合致し、次に、別の **Fibonacci** オブジェクトが変更され、この **"Calculate"** ルールに再度合致できていることが分かります。このプロセスは、すべての **Fibonacci** オブジェクトに値が設定されるまで継続されます。

図9.10 監査ビューのルール



9.5. 価格設定のデシジョン例 (デシジョンテーブル)

価格設定のデシジョン例では、スプレッドシートのデシジョンテーブルを使用して、DRL ファイルに直接ではなく、表形式で保険料金の価格を計算する方法が説明されます。

以下は価格設定の例の概要です。

- **名前**: **decisiontable**
- Main クラス: **org.drools.examples.decisiontable.PricingRuleDTExample** (in **src/main/java**)
- **モジュール**: **drools-examples**
- **タイプ**: Java アプリケーション
- **ルールファイル**: **org.drools.examples.decisiontable.ExamplePolicyPricing.xls** (**src/main/resources** 内)
- **目的**: スプレッドシートのデシジョンテーブルを使用してルールを定義する方法を示します。

スプレッドシートのデシジョンテーブルは、表形式でビジネスルールを定義する XLS または XLSX 形式のスプレッドシートです。スプレッドシートのデシジョンテーブルは、スタンドアロンの Red Hat Process Automation Manager プロジェクトに追加したり、Business Central のプロジェクトにアップロードしたりできます。スプレッドシートの各行がルールになり、各列が条件、アクション、または別のルール属性になります。最初に Red Hat Process Automation Manager でデシジョンテーブルを作成してアップロードします。次に、その他のすべてのルールアセットと同じように、定義したルールを Drools Rule Language (DRL) ルールにコンパイルします。

この価格設定の例では、特定タイプの保険を申請するドライバーに対して基本価格と割引を計算するビジネスルールセットを提供します。保険の種類とともにドライバーの年齢と履歴が、基本保険料の算定で考慮され、別のルールで適用可能な割引が計算されます。

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.decisiontable.PricingRuleDTExample** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます。

```
Cheapest possible
BASE PRICE IS: 120
DISCOUNT IS: 20
```

この例を実行するコードは、標準の実行パターンに準拠しています。ルールが読み込まれ、ファクトが挿入されて、ステートレスな KIE セッションが作成されます。この例における違いは、DRL ファイルや他のソースではなく、**ExamplePolicyPricing.xls** ファイルでルールが定義されるという点です。このスプレッドシートファイルは、テンプレートと DRL ルールを使ってデシジョンエンジンに読み込まれます。

スプレッドシートのデシジョンテーブルの設定

ExamplePolicyPricing.xls スプレッドシートには、以下の 2 つのデシジョンテーブルが含まれています。

- **Base pricing rules**
- **Promotional discount rules**

この例のスプレッドシートで分かるように、デシジョンテーブルの作成にはスプレッドシートの最初のタブしか使用できませんが、単一のタブ内に複数のテーブルが作成できます。デシジョンテーブルは必ずしもトップダウンの論理に従うものではなく、ルールとなるデータを補足する手段です。ルールの評価は、デシジョンエンジンのすべての通常のメカニックが適用されるため、必ずしも特定の順序で行われるわけではありません。このために、スプレッドシートの同一タブ内に複数のデシジョンテーブルが作成可能となります。

デシジョンテーブルは、対応するルールテンプレートファイルである **BasePricing.drt** と **PromotionalPricing.drt** で実行されます。これらのテンプレートファイルはテンプレートパラメーターによってデシジョンテーブルを参照し、デシジョンテーブルの条件およびアクションの各種ヘッダーを直接参照します。

BasePricing.drt ルールテンプレートファイル

```
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 ルールテンプレートファイル

```
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
```

ルールは、KIE セッション **DTableWithTemplateKB** の **kmodule.xml** 参照によって実行されます。これは **ExamplePolicyPricing.xls** スプレッドシートを特定して参照するもので、ルールの実行の成功には必要なものです。この実行方法により、ルールをスタンドアロンユニットとして実行したり (ここでの例)パッケージ化されたナレッジ JAR (KJAR) ファイルにルールを含めたりすることができるので、スプレッドシートはルール実行とともにパッケージ化されます。

kmodule.xml ファイルの以下のセクションは、ルールが正常に実行され、スプレッドシートが機能するために必要になります。

```
    <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>
```

ルールテンプレートファイルを使ったデシジョンテーブルの実行方法とは別に、**DecisionTableConfiguration** オブジェクトを使用して、**DecisionTableInputType.xls** のような入力スプレッドシートを入力タイプとして指定することもできます。

```
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 );
```

価格設定の例では以下の 2 つのファクトタイプを使用します。

- **Driver**
- **Policy**.

この例では、これらのファクトのデフォルト値をそれぞれのの Java クラス **Driver.java** と **Policy.java** に設定します。**Driver** は 30 歳で、これまでに保険の請求をしたことがなく、現在のリスクプロファイル **LOW** となっています。申請している **Policy** は **COMPREHENSIVE** です。

デシジョンテーブルでは、各行は異なるルール。各列は条件またはアクションとみなされます。実行時にアジェンダがクリアされなければ、デシジョンテーブルの各行が評価されます。

デシジョンテーブルのスプレッドシート (XLS または XLSX) には、ルールデータを定義する以下の 2 つの主要なエリアが必要です。

- **RuleSet** エリア
- **RuleTable** エリア

RuleSet エリアでは、ルールセット名、ユニバーサルルール属性など、(このスプレッドシートだけでなく) すべてのルールをパッケージ全体に、グローバルに適用する要素を定義します。**RuleTable** エリアでは、実際のルール (行) と、指定したルールセットのルールテーブルを構成する条件、アクション、その他のルール属性 (列) を定義します。デシジョンテーブルのスプレッドシートには複数の **RuleTable** エリアを追加できますが、**RuleSet** エリアは 1 つのみとなります。

図9.11 デシジョンテーブルの設定

	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

RuleTable エリアでは、ルール属性を適用するオブジェクトも定義します。この例では、**Driver** と **Policy**、さらにオブジェクトの制限です。たとえば、**Driver** オブジェクトの制限では、**Age Bracket** コラムが **age >= $1, age <= $2** と定義されています。ここでのコンマ区切りの範囲は、**18,24** などのテーブルのコラム値で定義されます。

基本価格のルール

価格設定例での **Base pricing rules** デシジョンテーブルでは、ドライバーの年齢、リスクプロファイル、請求数、ポリシータイプを評価し、これらの条件をベースにしたポリシーの基本価格を算定します。

図9.12 基本価格の計算

	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

Driver 属性は、以下のテーブルコラムで定義されます。

- **Age Bracket**: この年齢層には、ドライバー年齢の条件範囲を定義する条件 **age >=$1, age <=$2** の定義があります。この条件コラムでは **$1 and $2** を使用しており、スプレッドシートではコンマ区切りになります。ここでの値入力は **18,24** や **18, 24** の形式となり、両方ともビジネスルールの実行で機能する形式です。

- **Location risk profile**: リスクプロファイルは、この例のプログラムでは常に **LOW** として渡す文字列です。ただし、**MED** または **HIGH** に変更することが可能です。
- **Number of prior claims**: これまでの請求数は整数で定義し、アクションをトリガーするには条件コラムのものと同一である必要があります。この値は範囲ではなく、完全一致のみになります。

デシジョンテーブルの **Policy** は、ルールの条件とアクションの両方で使用され、属性は以下のテーブルコラムで定義されます。

- **Policy type applying for**: ポリシータイプは文字列として渡される条件で、適用される以下のいずれかのポリシータイプを定義します: **COMPREHENSIVE**、**FIRE_THEFT**、または **THIRD_PARTY**。
- **Base $ AUD**: **basePrice** は **ACTION** として定義され、これはこの値に対応するスプレッドシートのセルをベースに制限 **policy.setBasePrice($param);** で価格を設定します。このデシジョンテーブルの対応する DRL ルールを実行する際には、ファクトに合致する true 条件でルールの **then** 部分がこのアクションステートメントを実行し、基本価格を対応する値に設定します。
- **Record Reason**: ルールが正常に実行されると、アクションは出力メッセージを **System.out** コンソールに生成し、どのルールが適用されたかが反映されます。これは後でアプリケーションにキャプチャーされ、プリントされます。

この例では、左側の最初のコラムでルールをカテゴリ分けしています。このコラムは注釈目的で、ルール実行には影響がありません。

プロモーション割引ルール
価格設定例での **Promotional discount rules** デシジョンテーブルでは、ドライバーの年齢、請求数、ポリシータイプを評価し、ポリシー価格の割引を算定します。

図9.13 割引計算

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

このデシジョンテーブルには、ドライバーに適用可能な割引条件が含まれています。基本価格の算定と同様に、このテーブルではドライバーの **Age**、**Number of prior claims**、および **Policy type applying for** を評価して、適用する **Discount %** 率を判定します。たとえば、ドライバーは 30 歳であり、請求履歴がなく、**COMPREHENSIVE** ポリシーを申請している場合、**20** パーセントの割引率が導き出されます。

9.6. ペットショップのデシジョン例 (アジェンダグループ、グローバル変数、コールバック、GUI 統合)

ペットショップのデシジョンセットの例では、ルールでのアジェンダグループとグローバル変数の使用方法および Red Hat Process Automation Manager ルールとグラフィカルユーザーインターフェース (GUI) の統合方法を説明しています。今回は Swing ベースのデスクトップアプリケーションを使用します。また、この例では、コールバックを使用して実行中のデシジョンエンジンと通信し、ランタイム時に加えられたワーキングメモリー内の変更をもとに GUI を更新する方法も説明しています。

以下は、ペットショップの例の概要です。

- **名前**: **petstore**
- Main クラス: (**src/main/java** 内の) **org.drools.examples.petstore.PetStoreExample**
- **モジュール**: **drools-examples**
- **タイプ**: Java アプリケーション
- **ルールファイル**: (**src/main/resources** 内の) **org.drools.examples.petstore.PetStore.drl**
- **目的**: ルールアジェンダグループ、グローバル変数、コールバック、GUI 統合を例示します。

ペットショップの例では、**PetStoreExample.java** クラス例を使用して (Swing イベントを処理する複数のクラスに加え)、以下のクラスを主に定義しています。

- **Petstore** には **main()** メソッドが含まれます。
- **PetStoreUI** は Swing ベースの GUI を作成して表示します。このクラスには複数の小さいクラスが含まれており、マウスボタンのクリックなど、さまざまな GUI イベントに主に対応します。
- **TableModel** には表データが含まれています。このクラスは基本的に **AbstractTableModel** を継承する JavaBean です。
- **CheckoutCallback** により、GUI がルールと対話できるようになります。
- **Ordershow** は購入するアイテムを保持します。
- **Purchase** には、顧客が購入する商品と商品の詳細が保存されます。
- **Product** は、販売可能な商品と価格の詳細を含む JavaBean です。

この例の Java コードはほぼ、プレーンな JavaBean か Swing ベースとなっています。Swing コンポーネントの詳細は、「Creating a GUI with JFC/Swing」の Java チュートリアルを参照してください。

ペットショップの例でのルール実行動作

他のデシジョンセットの例では、ファクトがすぐにアサートされて実行されるのに対し、ペットショップの例では、ユーザーの対話をもとに他のファクトが収集されるまでルールは実行されます。このルールでは、コンストラクターで作成される **PetStoreUI** オブジェクトを使用してルールを実行し、**Vector** オブジェクトの **stock** を受けれ入れて商品を収集します。次に、この例では、以前の読み込まれたルールベースを含む **CheckoutCallback** クラスのインスタンスを使用します。

ペットショップの KIE コンテナーおよびファクト実行の設定

```
// 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();
```

ルールを実行する Java コードは **CheckoutCallBack.checkout()** メソッドに含まれます。このメソッドは、ユーザーが UI で **チェックアウト** をクリックするとトリガーされます。

CheckoutCallBack.checkout() からのルール実行

```
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();
}
```

このコード例では、2 つの要素を **CheckoutCallBack.checkout()** メソッドに渡します。1 つ目の要素は、GUI の一番下にある出力テキストのフレームを囲む **JFrame** Swing コンポーネントのハンドルです。2 つ目の要素は注文アイテムのリストで、GUI の右上のセクションにある **Table** エリアからの情報を保存する **TableModel** から取得します。

for ループは GUI からの注文アイテム一覧を **Order** JavaBean に変換します。これは、**PetStoreExample.java** ファイルにも含まれています。

今回の例では、データはすべて Swing コンポーネントに含まれており、ユーザーが UI の **チェックアウト** をクリックしない限り実行されないため、ルールはステートレスな KIE セッションで実行します。ユーザーが **チェックアウト** をクリックするたびに、リストの内容を Swing **TableModel** から KIE セッションのワーキングメモリーに移動し、**ksession.fireAllRules()** メソッドで実行します。

このコード内には、**KieSession** への呼び出しが 9 個あります。1 つ目は、**KieContainer** から新しい **KieSession** を作成します (この例では、**main()** メソッドの **CheckoutCallBack** クラスから **KieContainer** に渡されます)。次の 2 つの呼び出しは、ルールでグローバル変数として保持されるオブジェクト を 2 つ渡します (メッセージの書き込みに使用する Swing テキストエリアと Swing フレーム)。他に挿入することで、商品の情報を **KieSession** と注文リストに配置します。最後の呼び出しは、標準の **fireAllRules()** です。

ペットショップのルールファイルのインポート、グローバル変数、Java 関数
PetStore.drl ファイルには、さまざまな Java クラスをルールで利用できるように、標準のパッケージとインポートステートメントが含まれています。このルールファイルには、**frame** および **textArea** などのように、ルール内で使用する **グローバル変数** が含まれています。グローバル変数では、Swing コンポーネント **JFrame** と、**setGlobal()** メソッドを呼び出した Java コードにより以前に渡された **JTextArea** コンポーネントへの参照を保持します。ルールが実行されるとすぐに失効するルールの標準変数とは異なり、グローバル変数は KIE セッションの有効期間中この値を保持します。つまり、この後に続く全ルールを評価するのに、これらの変数の内容を使用できます。

PetStore.drl パッケージ、インポートおよびグローバル変数

```
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
```

PetStore.drl ファイルには、このファイル内のルールが使用する関数 2 つも含まれています。

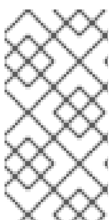
PetStore.drl Java 関数

```
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;
}
```

この 2 つの関数は以下のアクションを実行します。

- **doCheckout()** は、チェックアウトするかどうかユーザーに尋ねるダイアログボックスを表示します。チェックアウトする場合は、フォーカスが **checkout** アジェンダグループに設定され、そのグループのルールが (今後) 実行できるようにします。
- **requireTank()** は、水槽を購入するかどうかを確認するダイアリーを表示します。購入する場合は、新しい水槽の **Product** がワーキングメモリーの注文リストに追加されます。



注記

この例では、効率化を図るため、すべてのルールと関数が同じルールファイルで実行しています。実稼働環境では、通常、ルールと関数を別のファイルに分けるか、静的な Java メソッドを構築して、**import function my.package.name.hello** などのインポート関数を使用し、ファイルをインポートします。

アジェンダグループを使用したペットショップルール

ペットショップの例のルールはほぼ、アジェンダグループを使用してルールの実行を制御しています。アジェンダグループを使用すると、デシジョンエンジンアジェンダを分割し、ルールのグループの実行を、詳細にわたり制御できるようになります。デフォルトでは、全ルールはアジェンダグループ **MAIN** に含まれます。**agenda-group** 属性を使用してルールに異なるアジェンダグループを指定できます。

最初は、ワーキングメモリーは、アジェンダグループ **MAIN** にフォーカスを当てます。アジェンダグループのルールは、グループがこのフォーカスを受けた場合のみ実行されます。**setFocus()** のメソッドか、**auto-focus** のルール属性を使用してフォーカスを設定できます。**auto-focus** 属性を使用すると、ルールが一致してアクティベートされた場合のみ、ルールにアジェンダグループのフォーカスが自動的に当てられます。

ペットショップの例では、ルールに以下のアジェンダグループを使用します。

- **"init"**
- **"evaluate"**
- **"show items"**
- **"checkout"**

たとえば、同じルール **"Explode Cart"** は **"init"** のアジェンダグループを使用して、ショッピングカードのアイテムを実行して、KIE セッションのワーキングメモリーに挿入するオプションが提供されるようにします。

ルール "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
```

このルールは、**grossTotal** がまだ計算されていない全注文に対してマッチングされます。購入アイテムごとに、順番に実行がループされます。

ルールは、アジェンダグループに関連する以下の機能を使用します。

- **agenda-group "init"** はアジェンダグループの名前を定義します。この例では、グループにはルールが 1 つしかありませんが、Java コードもルール結果もこのグループにフォーカスされていないため、**auto-focus** の属性により、ルールが実行されるかが決まります。
- このルールはアジェンダグループで唯一のルールですが、**auto-focus true** を使用して、**fireAllRules()** が Java コードから呼び出されると、必ず実行されるようにします。
- **kcontext….setFocus()** で、フォーカスを **"show items"** と **"evaluate"** のアジェンダグループに設定して、これらのルールが実行されるようにします。実際は、注文に含まれる全アイテムをループでチェックし、メモリーに挿入してから、挿入ごとに他のルールを実行します。

"show items" アジェンダグループには **"Show Items"** というルール 1 つだけが含まれます。KIE セッションのワーキングメモリーに現在含まれる注文で購入があるたびに、このルールを使用して、ルールファイルに定義した **textArea** 変数をもとに、GUI の下の部分にあるテキストエリアに詳細がロギングされます。

ルール "Show Items"

```
rule "Show Items"
    agenda-group "show items"
  when
```

```
    $order : Order()
    $p : Purchase( order == $order )
  then
   textArea.append( $p.product + "\n");
end
```

また、**"evaluate"** アジェンダグループにより、**"Explode Cart"** ルールからフォーカスを取得します。このアジェンダグループには、**"Free Fish Food Sample"** と **"Suggest Tank"** のルールが 2 つ含まれます。"Free Fish Food Sample"、"Suggest Tank" の順番に実行されます。

ルール "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
```

ルール **"Free Fish Food Sample"** は、以下の条件がすべて該当する場合のみ実行されます。

1. アジェンダグループ **"evaluate"** がルール実行で評価されている
2. ユーザーが魚の餌をまだ持っていない
3. ユーザーが無料の魚の餌サンプルをまだ持っていない
4. ユーザーが金魚を注文している

この注文ファクトが上記の要件すべてを満たす場合には、新しい商品 (Fish Food Sample) が作成され、ワーキングメモリーの注文に追加されます。

ルール "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
```

ルール **"Suggest Tank"** は以下の条件がすべて該当する場合のみ実行されます。

1. ユーザーが水槽を注文していない
2. ユーザーが 6 匹以上注文した

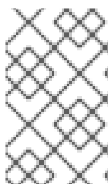
このルールが実行されると、ルールファイルに定義されている **requireTank()** 関数が呼び出されます。この関数により、水槽を購入するかどうかを尋ねるダイアログが表示されます。新しい水槽の **Product** がワーキングメモリーの注文リストに追加されます。ルールが **requireTank()** 関数を呼び出した場合には、このルールを使用して、関数に Swing GUI のハンドルが含まれるように、**frame** のグローバル変数を渡します。

ペットショップの例の **"do checkout"** ルールにはアジェンダルールや **when** 条件がないので、ルールは常に実行されて、デフォルトの **MAIN** のアジェンダグループの一部とみなされます。

ルール "do checkout"

```
rule "do checkout"
  when
  then
    doCheckout(frame, kcontext.getKieRuntime());
end
```

このルールが実行されると、ルールファイルで定義されている **doCheckout()** 関数を呼び出します。この関数により、チェックアウトするかどうかユーザーに尋ねるダイアログボックスが表示されます。チェックアウトする場合は、フォーカスが **checkout** アジェンダグループに設定され、そのグループのルールが (今後) 実行できるようにします。このルールで **doCheckout()** 関数を呼び出し、この変数に Swing GUI のハンドルが含まれるように **frame** グローバル変数を渡します。



注記

この例では、結果が想定どおりに実行されない場合のトラブルシューティングの方法を例示します。ルールの **when** ステートメントから条件を削除して、**then** ステートメントのアクションをテストし、アクションが正しく実行されることを検証します。

"checkout" アジェンダグループには、**"Gross Total"**、**"Apply 5% Discount"** および **"Apply 10% Discount"** の注文のチェックアウト処理、割引の適用のルールが 3 つ含まれています。

ルール "Gross Total"、"Apply 5% Discount" および "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
```

ユーザーがまだ総計を算出していない場合には、**Gross Total** で、商品の価格を累積して合計を出し、この合計を KIE セッションに渡して、**textArea** のグローバル変数を使用し、Swing **JTextArea** で合計を表示します。

総計が **10** から **20** (通貨単位) の場合には、**"Apply 5% Discount"** ルールで割引合計を計算し、KIE セッションに追加して、テキストエリアに表示します。

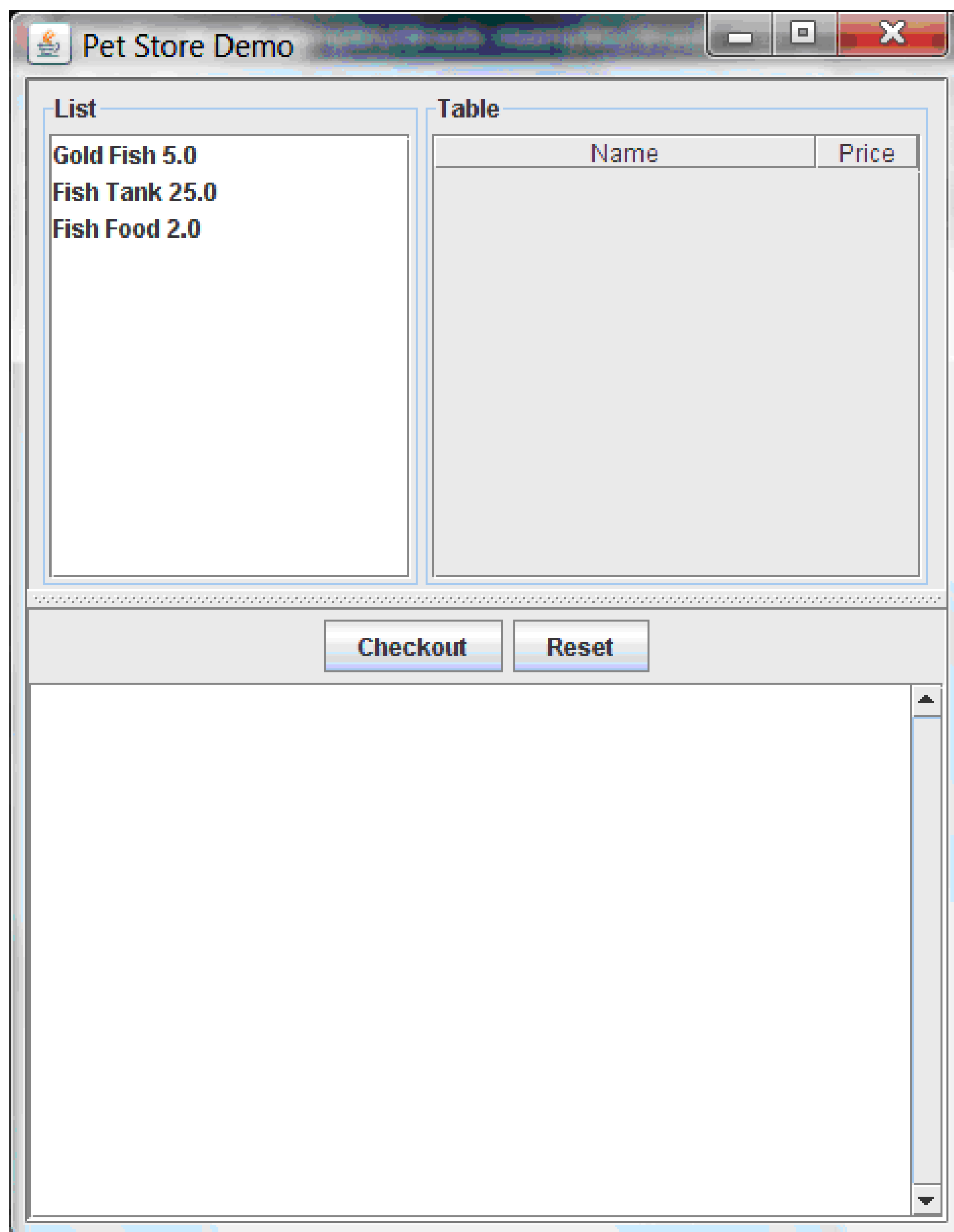
総計が **20** 未満の場合には、**"Apply 10% Discount"** ルールで割引合計を計算し、KIE セッションに追加して、テキストエリアに表示します。

ペットショップ例の実行

他の Red Hat Process Automation Manager のデシジョン例と同じように、お使いの IDE で **org.drools.examples.petstore.PetStoreExample** クラスを Java アプリケーションとして実行し、ペットショップの例を実行します。

ペットショップの例を実行すると、**Pet Store Demo** GUI ウィンドウが表示されます。このウィンドウでは、購入可能な商品 (左上)、選択済み商品の空白のリスト (右上)、**チェックアウト** および **リセット** ボタン (真ん中)、空白のシステムメッセージエリア (下) が表示されます。

図9.14 起動後のペットショップ例の GUI



この例では、以下のイベントが発生して、この実行動作を確立します。

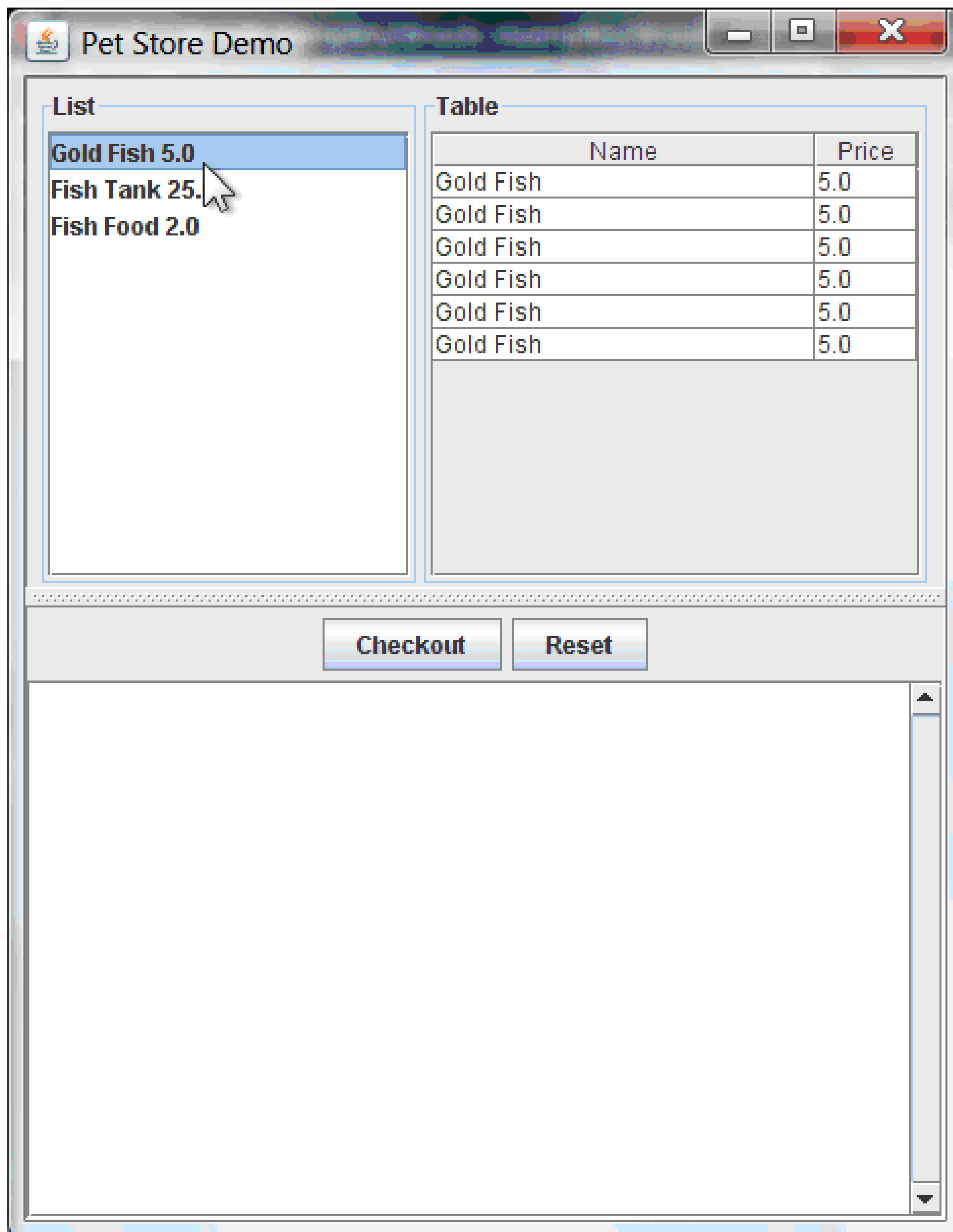
1. **main()** メソッドがルールベースの実行と読み込みを終えているが、ルールが実行されていないこと。今のところ、実行されたルールに関連する唯一のコードがこれです。

2. 新しい **PetStoreUI** オブジェクトが作成され、後で使用できるようにルールベースにハンドルを渡すこと。

3. さまざまな Swing コンポーネントが関数を実行し、最初の UI 画面が表示され、ユーザーの入力を待ちます。

リストからさまざまな商品をクリックして、UI 設定をチェックできます。

図9.15 ペットショップ例の GUI のチェック



ルールコードはまだ実行されていません。UI は Swing コードを使用してユーザーによるマウスクリックを検出し、選択済みの商品を **TableModel** オブジェクトに追加して、UI の右上隅に表示します。この例では、Model-View-Controller 設計パターンを紹介しています。

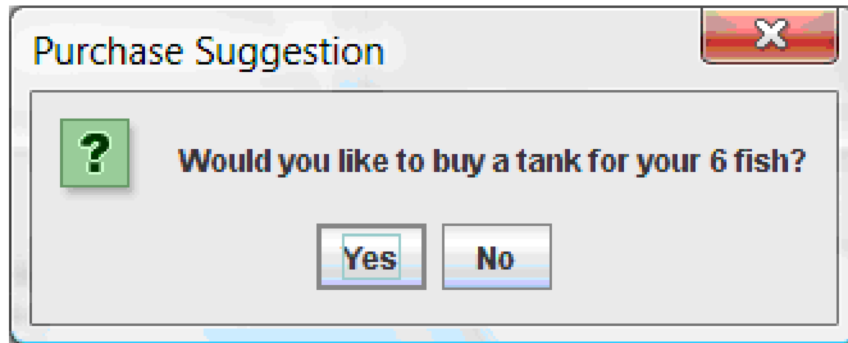
Checkout をクリックすると、ルールが以下の方法で実行されます。

1. Swing クラスは **Checkout** がクリックされるまで待機して、(最終的に) **CheckOutCallBack.checkout()** メソッドを呼び出します。これにより、**TableModel** オブジェ

クト (UI の右上隅) から KIE セッションのワーキングメモリーにデータを挿入します。その後に、メソッドによりルールが実行されます。

2. **"Explode Cart"** ルールは、**auto-focus** 属性を **true** に設定して最初に実行します。このルールは、カートの商品すべてを順にループしていき、商品がワーキングメモリーに含まれていることを確認し、**"show Items"** と **"evaluate"** アジェンダグループに実行するオプションを提供します。このグループのルールは、カートのコンテンツをテキストエリア (UI の下) に追加して、魚の餌を無料で受け取る資格があるかどうかを評価し、また水槽購入の有無を尋ねるかどうかを決定します。

図9.16 水槽の資格



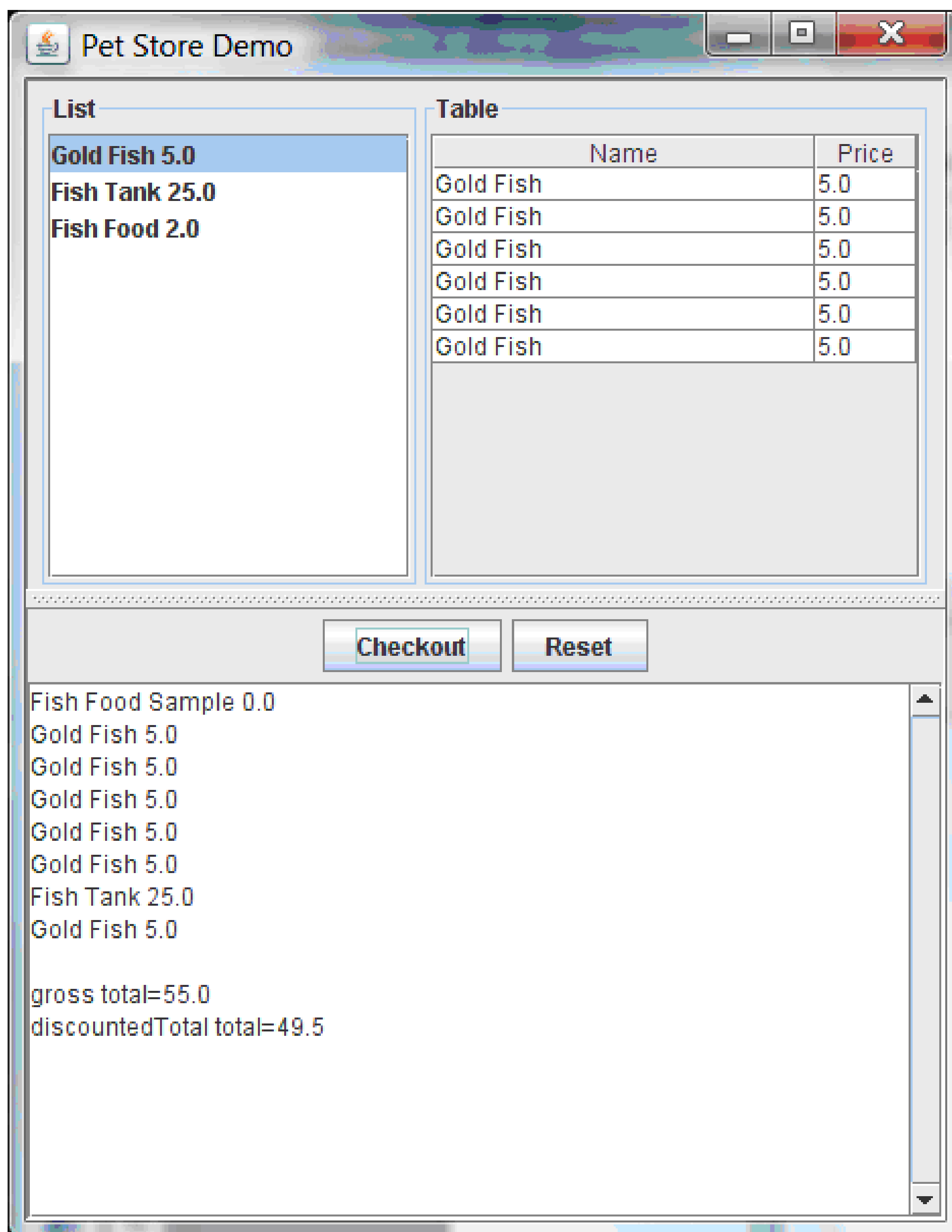
3. 現在、他のアジェンダグループにフォーカスが当たっておらず、**"do checkout"** ルールは、デフォルトの **MAIN** アジェンダグループに含まれているので、次に実行されます。このルールは常に **doCheckout()** 関数を呼び出し、この関数によりチェックアウトをするかどうかが尋ねられます。

4. **doCheckout()** 関数は、フォーカスを **"checkout"** アジェンダグループに設定し、そのグループ内のルールに、実行するオプションを提供します。

5. **"checkout"** アジェンダグループ内のルールは、「カート」内の内容を表示し、適切な割引を適用します。

6. Swing は、別の商品の選択 (およびもう一度ルールを実行させる) または GUI の終了のいずれかのユーザー入力を待ちます。

図9.17 全ルールが実行された後のペットショップ例の GUI



IDE コンソールでイベントのこのフローを例示するには、他の **System.out** 呼び出しを追加します。

IDE コンソールの System.out 出力

```
Adding free Fish Food Sample to cart
SUGGESTION: Would you like to buy a tank for your 6 fish? - Yes
```

9.7. 誠実な政治家のデシジョン例 (真理維持および顕著性)

誠実な政治家のデシジョンセットの例では、論理挿入を使用した真理維持の概念およびルールでの顕著性の使用方法を説明しています。

以下は、誠実な政治家の例の概要です。

- **名前**: **honestpolitician**
- **Main クラス**: (**src/main/java** 内の) **org.drools.examples.honestpolitician.HonestPoliticianExample**
- **モジュール**: **drools-examples**
- **タイプ**: Java アプリケーション
- **ルールファイル**: (**src/main/resources** 内の) **org.drools.examples.honestpolitician.HonestPolitician.drl**
- **目的**: ファクトの論理挿入をもとにした真理維持の概念およびルールでの顕著性の使用方法を紹介します。

誠実な政治家の例の前提として基本的に、ステートメントが True の場合にのみ、オブジェクトが存在できます。**insertLogical()** メソッドを使用して、ルールの結果により、オブジェクトを論理的に挿入します。つまり、論理的に挿入されたルールが True の状態であれば、オブジェクトは KIE セッションのワーキングメモリー内に留まります。ルールが True でなくなると、オブジェクトは自動的に取り消されます。

この例では、ルールを実行することで、企業による政治家の買収が原因で、政治家グループが「誠実」から「不誠実」に変わります。各政治家が評価されるにつれ、最初は honesty 属性を **true** に設定して開始しますが、ルールが実行されると政治家は「誠実」ではなくなります。状態が「誠実」から「不誠実」に切り替わると、ワーキングメモリーから削除されます。ルールの顕著性により、顕著性が定義されているルールをどのように優先付けするかを、デシジョンエンジンに通知します。そうでない場合には、デフォルトの顕著性の値 **0** を使用します。アクティベーションキューで順番待ちをしている場合には、顕著性の高いルールの優先順位が高くなります。

Politician および Hope クラス

この例の **Politician** クラス例は、誠実な政治家として設定されています。**Politician** クラスは、文字列アイテム **name** とブール値アイテム **honest** で構成されています。

Politician クラス

```
public class Politician {
    private String name;
    private boolean honest;
    ...
}
```

Hope クラスは、**Hope** オブジェクトが存在するかどうかを判断します。このクラスには意味を持つメンバーは存在しませんが、社会に希望がある限り、ワーキングメモリーに存在します。

Hope クラス

```
public class Hope {

    public Hope() {
```

```
    }
  }
```

政治家の誠実性に関するルール定義

誠実な政治家の例では、ワーキングメモリーに最低でも 1 名誠実な政治家が存在する場合には、**"We have an honest Politician"** ルールで論理的に新しい **Hope** オブジェクトを挿入します。すべての政治家が不誠実になると、**Hope** オブジェクトは自動的に取り除かれます。このルールでは、**salience** 属性の値が **10** となっており、他のルールより先に実行されます。理由は、この時点では **"Hope is Dead"** ルールが True となっているためです。

ルール "We have an honest politician"

```
rule "We have an honest Politician"
    salience 10
  when
    exists( Politician( honest == true ) )
  then
    insertLogical( new Hope() );
end
```

Hope オブジェクトが存在すると、すぐに **"Hope Lives"** ルールが一致して実行されます。**"Corrupt the Honest"** ルールよりも優先されるように、このルールにも **salience** 値を **10** に指定しています。

ルール "Hope Lives"

```
rule "Hope Lives"
    salience 10
  when
    exists( Hope() )
  then
    System.out.println("Hurrah!!! Democracy Lives");
end
```

最初は、誠実な政治家が 4 人いるので、このルールには 4 つのアクティベーションが存在し、すべてが競合しています。各ルールが順番に実行され、政治家が誠実でなくなるように、企業により各政治家を買収させていきます。政治家 4 人すべてが買収されたら、プロパティーが **honest == true** の政治家はいなくなります。**"We have an honest Politician"** のルールは True でなくなり、論理的に挿入されるオブジェクト (最後に実行された **new Hope()** による) は自動的に取り除かれます。

ルール "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
```

真理維持システムにより **Hope** オブジェクトが自動的に取り除かれると、**Hope** に適用された条件付き要素 **not** は True でなくなり、**"Hope is Dead"** ルールが一致して実行されます。

ルール "Hope is Dead"

```
rule "Hope is Dead"
  when
    not( Hope() )
  then
    System.out.println( "We are all Doomed!!! Democracy is Dead" );
end
```

実行と監査証跡
HonestPoliticianExample.java クラスでは、honest の状態が **true** に設定されている政治家 4 人が挿入され、定義したビジネスルールに対して評価します。

HonestPoliticianExample.java クラスの実行

```
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();
    }
```

この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.honestpolitician.HonestPoliticianExample** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます。

IDE コンソールでの実行出力

```
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
```

この出力では、democracy lives に誠実な政治家が最低でも 1 人いることが分かります。ただし、各政治家は企業に買収されているので、全政治家は不誠実になり、民主性がなくなります。

この例の実行フローをさらに理解するために、**HonestPoliticianExample.java** クラスを変更し、**DebugRuleRuntimeEventListener** リスナーと監査ロガーを追加して実行の詳細を表示することができます。

監査ロガーを含む HonestPoliticianExample.java クラス

```
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. **DebugAgendaEventListener** と **DebugRuleRuntimeEventListener** を処理するインポートパッケージに追加します。

2. この監査ログは **KieContainer** レベルでは利用できないので、**KieServices Factory** および **ks** 要素を作成してログを生成します。

3. **execute** メソッドを変更して **KieServices** と **KieContainer** 両方を使用します。

4. **execute** メソッドを変更して **KieContainer** に加えて **KieServices** で渡します。

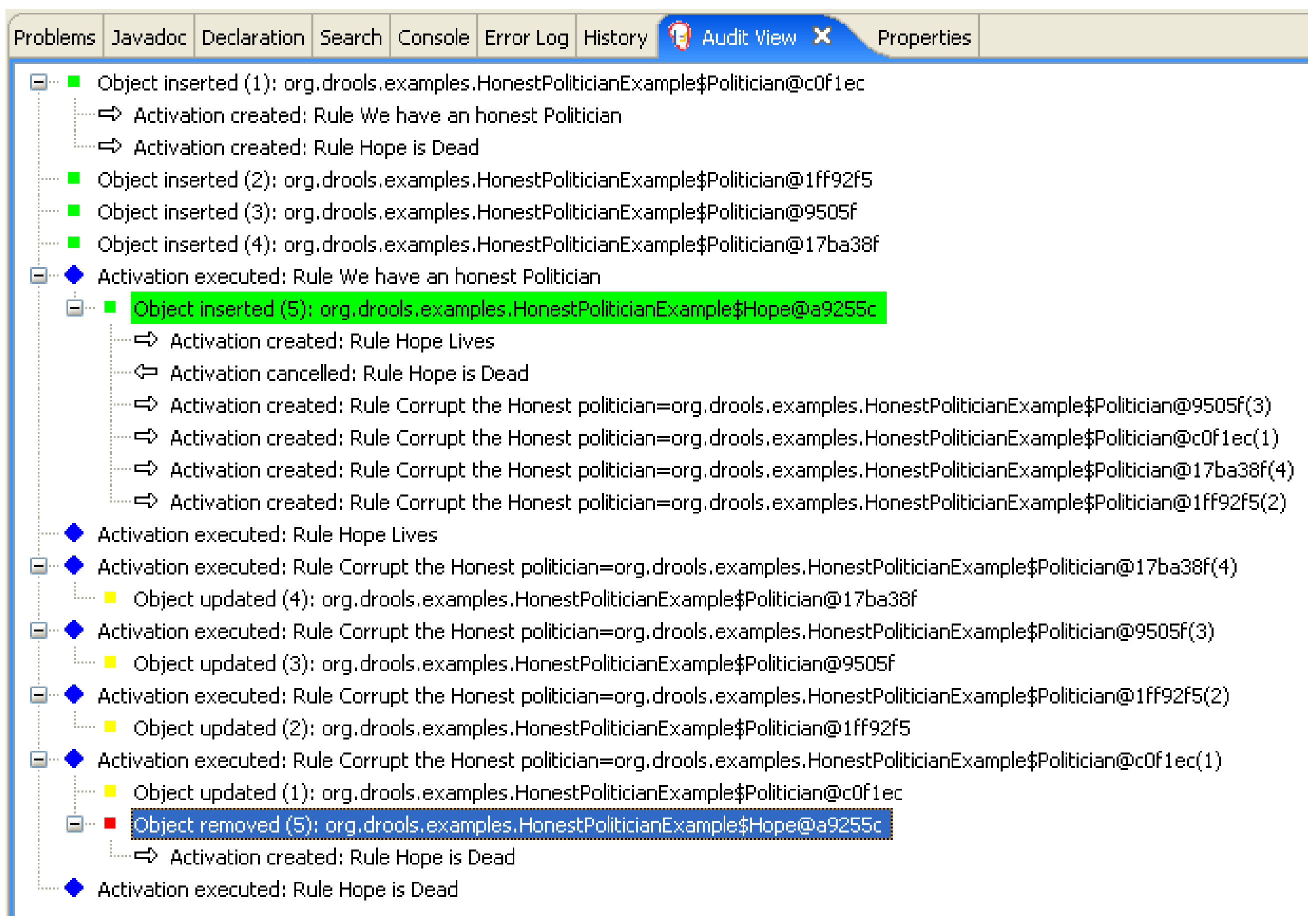
5. リスナーを作成します。

6. ルールの実行後にデバッグビュー、**監査ビュー** または IDE に渡すことが可能なログを構築します。

ロギング機能を変更して、「誠実な政治家」のサンプルを実行すると、**target/honestpolitician.log** から IDE デバッグビューまたは 利用可能な場合には (IDE の一部では **Window → Show View**) **監査ビュー** に、監査ログファイルを読み込むことができます。

この例では、**監査ビュー** では、クラスやルールのサンプルで定義されているように、実行フロー、挿入、取り消しが示されています。

図9.18 誠実な政治家例の監査ビュー



最初の政治家が挿入されると、2 つのアクティベーションが発生します。**"We have an honest Politician"** のルールは、**exists** の条件付き要素を使用するので、最初に挿入された政治家に対してのみ一度だけアクティベートされます。この条件付き要素は、政治家が最低でも 1 人挿入されると一致します。**Hope** オブジェクトがまだ挿入されていないので、ルール **"Hope is Dead"** もこの時点でアクティベートされます。**"We have an honest Politician"** ルールは、**"Hope is Dead"** ルールより、**salience** の値が高いので先に実行され、**Hope** オブジェクト (緑にハイライト) を挿入します。**Hope** オブジェクトを挿入すると、ルール **"Hope Lives"** がアクティベートされ、ルール **"Hope is Dead"** が無効になり

ます。この挿入により、挿入された誠実な各政治家に対して **"Corrupt the Honest"** ルールがアクティベートされます。**"Hope Lives"** のルールが実行されて、**"Hurrah!!! Democracy Lives"** が出力されます。

次に、政治家ごとに **"Corrupt the Honest"** ルールを実行して **"I'm an evil corporation and I have corrupted X"** と出力します。**X** には政治家の名前が入り、政治家の誠実性の値を **false** に変更します。最後の政治家が買収された時点で、真理維持システム (青でハイライト) により **Hope** が自動的に取り消されます。緑でハイライトされたエリアは、現在選択されている青のハイライトエリアの出元です。**Hope** ファクトが取り消されると、**"Hope is dead"** ルールが実行されて **"We are all Doomed!!! Democracy is Dead"** が出力されます。

9.8. 数独のデシジョン例 (複雑なパターン一致、コールバック、GUI 統合)

数独のデシジョンセットの例では、人気の数字パズルゲーム「数独」をもとにしています。このセットでは、Red Hat Process Automation Manager のルールを使用してさまざまな制約をもとに、多数の考えられる回答スペースの中で回答を導き出す方法を説明しています。また、この例では、Red Hat Process Automation Manager ルールとグラフィカルユーザーインターフェース (GUI) の統合方法も説明しています。今回は Swing ベースのデスクトップアプリケーションを使用します。また、この例では、コールバックを使用して実行中のデシジョンエンジンと通信し、ランタイム時に加えられたワーキングメモリー内の変更をもとに GUI を更新する方法も説明しています。

以下は数独の例の概要です。

- **名前**: **sudoku**
- **Main クラス**: (**src/main/java** 内の) **org.drools.examples.sudoku.SudokuExample**
- **モジュール**: **drools-examples**
- **タイプ**: Java アプリケーション
- **ルールファイル**: (**src/main/resources** 内の) **org.drools.examples.sudoku.*.drl**
- **目的**: 複雑なパターン一致、問題解決、コールバック、GUI 統合を例示します。

数独は、ロジックベースの数字配置パズルです。目的は、各列、行、および 3x3 ゾーン に 1 から 9 の数字が一度だけ含まれるように 9x9 のグリッドを埋めることです。パズルセッターでは、グリッド内の一部だけ記入されており、上記の制約ですべての空白を埋めるのがパズルの回答者のタスクです。

問題解決の一般的なストラテジーとして、新しい番号の挿入時に、特定の 3x3 ゾーン、行、列で同じ番号がないことを確認します。この数独のデシジョンセットの例では、Red Hat Process Automation Manager ルールを使用して、さまざまな難易度の数独パズルを解き、無効なエントリーが含まれ、不備のあるパズルの解決を試みます。

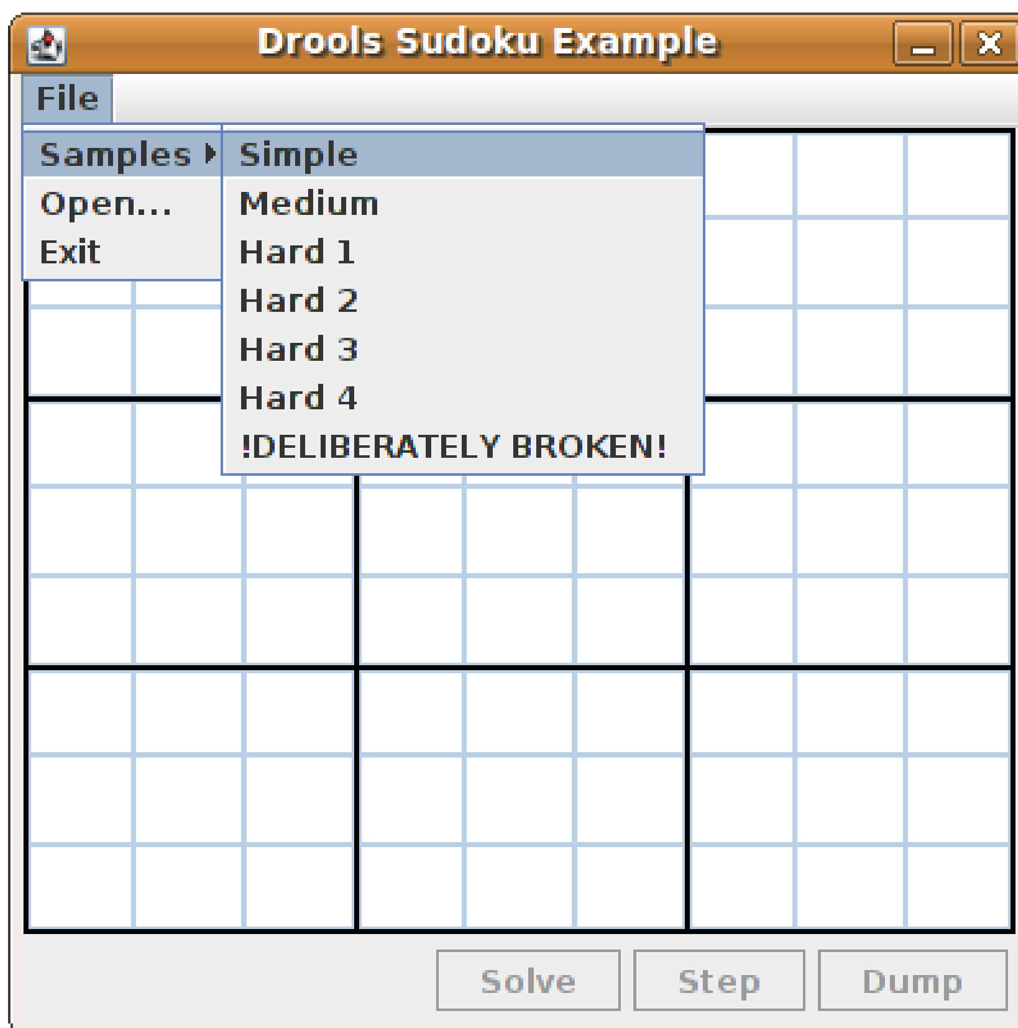
数独例の実行および対話

他の Red Hat Process Automation Manager のデシジョン例と同じように、お使いの IDE で **org.drools.examples.sudoku.SudokuExample** クラスを Java アプリケーションとして実行し、数独の例を実行します。

数独の例を実行すると、**Drools Sudoku Example** GUI ウィンドウが表示されますこのウィンドウには、空白のグリッドが含まれますが、プログラムには、読み込みや解決が可能なグリッドが複数、内部に格納されています。

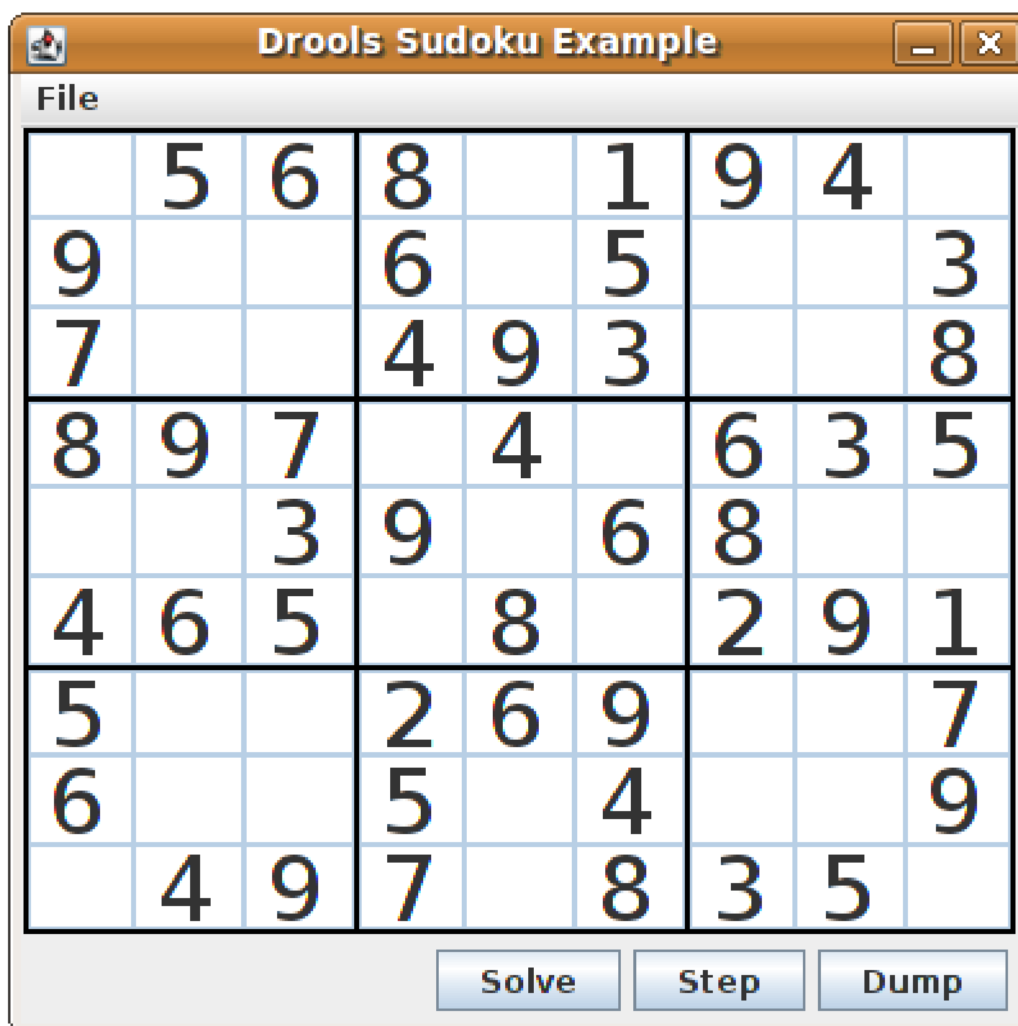
File → Samples → Simple をクリックして、例の 1 つを読み込みます。グリッドが読み込まれるまで、すべてのボタンが無効になっている点に注目してください。

図9.19 起動後の数独例の GUI



Simple サンプルを読み込むと、パズルの最初の状態に合わせて、グリッドが埋められます。

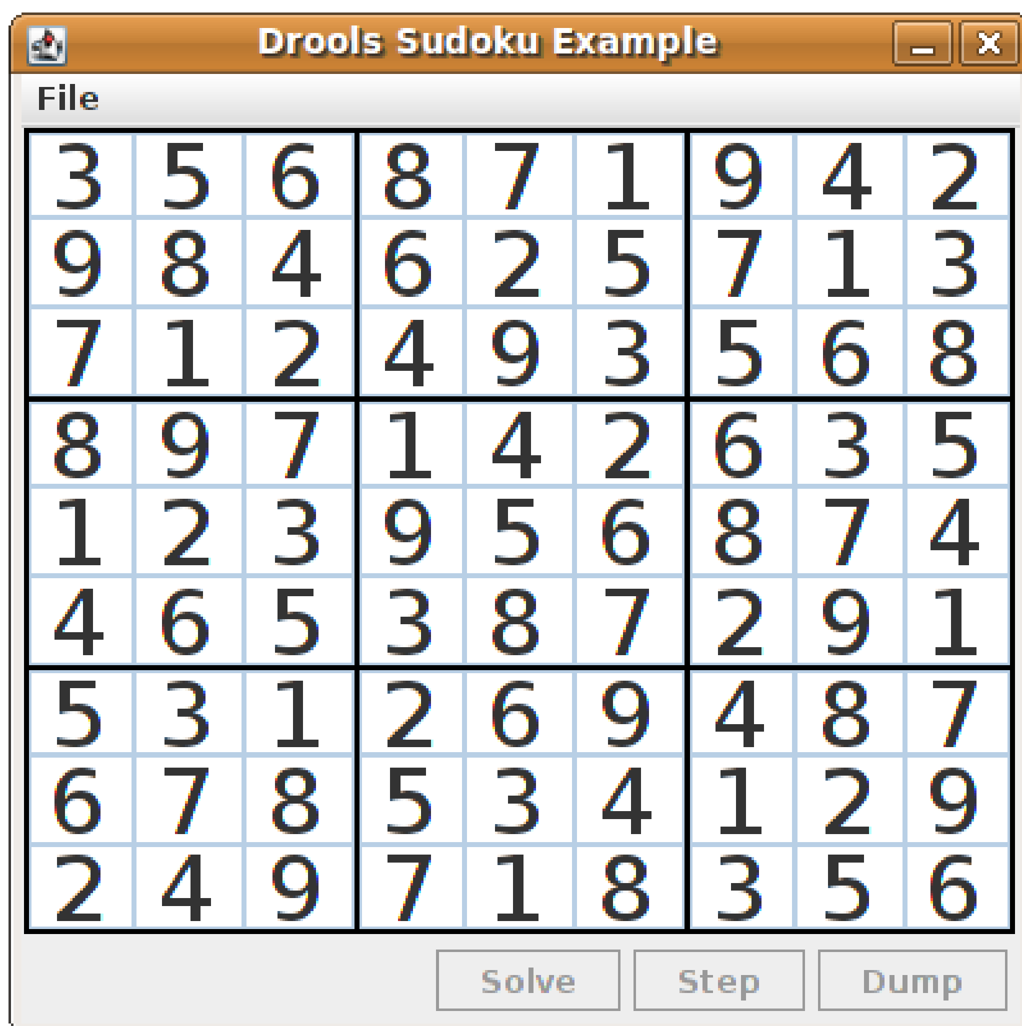
図9.20 Simple サンプルを読み込んだ後の数独例の GUI



以下のオプションから選択します。

- Solve をクリックして、数独の例に定義されているルールを実行し、残りの値を埋めていき、このボタンを再度無効にします。

図9.21 Simple サンプルの解決



- **Step** をクリックして、ルールセットに含まれる次の数字を表示します。IDE のコンソールウィンドウでは、解決手順を実行するルールに関する情報が詳細に表示されます。

  IDE コンソールでの手順実行の出力

```
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]
```

- **Dump** をクリックしてグリッドの状態を表示します。セルには、解決済みの値か、残りの候補値が表示されます。

  IDE コンソールでのダンプ実行の出力

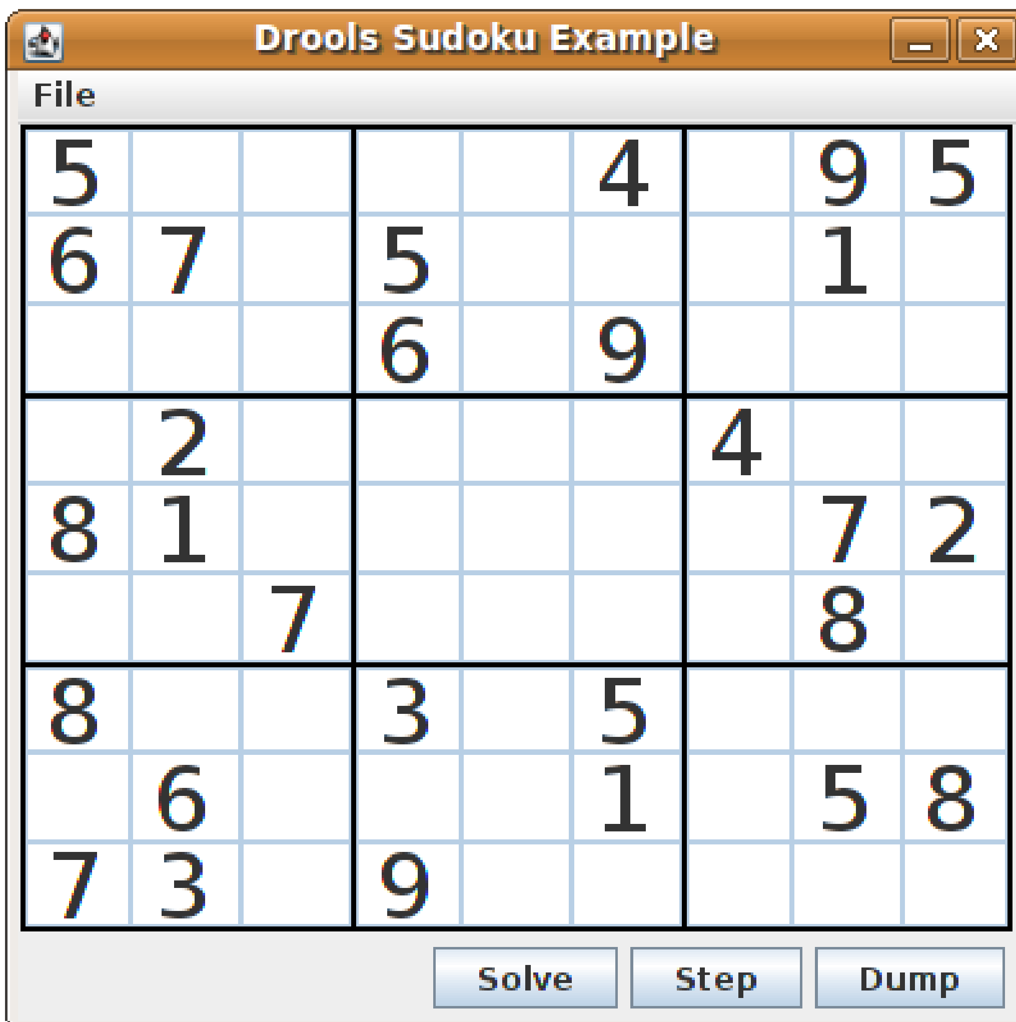
```
        Col: 0     Col: 1     Col: 2     Col: 3     Col: 4     Col: 5     Col: 6     Col: 7     Col: 8
Row 0:  123456789  --- 5 ---  --- 6 ---  --- 8 ---  123456789  --- 1 ---  --- 9 ---  --- 4 ---
123456789
Row 1:  --- 9 ---  123456789  123456789  --- 6 ---  123456789  --- 5 ---  123456789
123456789  --- 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
```

数独の例には、不備のあるサンプルファイルが意図的に含められています。このファイルは、例で定義したルールを使用して解決できます。

File → Samples → !DELIBERATELY BROKEN! をクリックして、不備のあるサンプルを読み込みます。グリッドは、最初の行に **5** の値を 2 回表示できないにもかかわらず、表示されるなど、問題が含まれた状態で表示されます。

図9.22 不備のある数独例の最初の状態



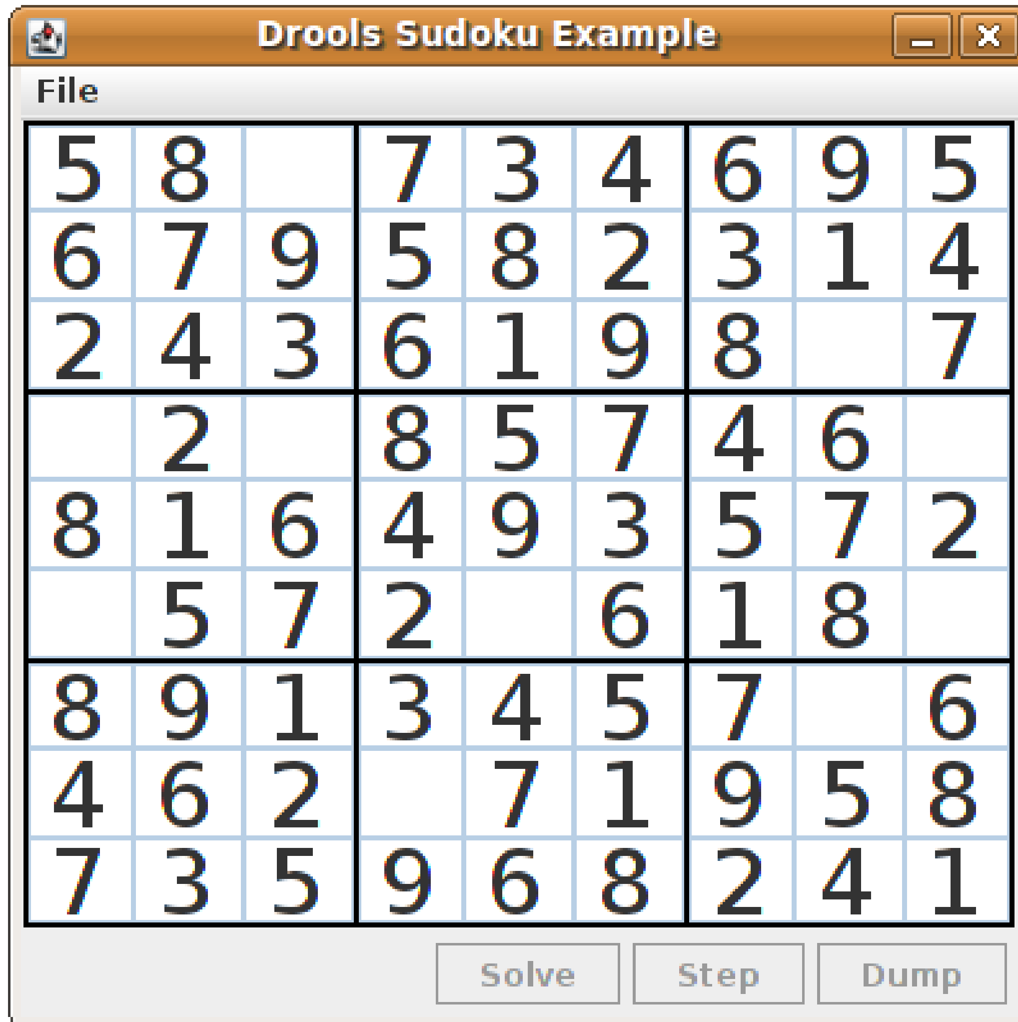
Solve をクリックしてこの無効なグリッドに解決ルールを適用します。数独の例に含まれる関連の解決ルールにより、サンプルの問題が検出され、できる限りパズルを解決します。このプロセスでは、すべてを完了させず、空白のセルをいくつか残します。

解決ルールのアクティビティーが IDE コンソールウィンドウに表示されます。

不備のあるサンプルでの問題検出

> 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.

図9.23 不備のあるサンプルの解決試行



Hard のラベルの付いた数独サンプルファイルはより複雑で、解決ルールを使用しても解決できない可能性があります。解決をしようとして失敗した場合には、IDE コンソールウィンドウに表示されます。

解決不可の Hard サンプル

> Validation complete.
> ...
> Sorry - can't solve this grid.

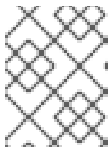
不備のあるサンプルを解決するためのルールでは、セルの候補となりえる値をもとにした標準の解決手法を実装します。たとえば、セットに値が 1 つ含まれる場合には、これが値になります。セルが 9 個あるグループの 1 つに値が 1 度挿入された場合に、ルールを使用して、特定のセルに対する値を持ち、タイプが **Setting** のファクトを挿入します。このファクトは、セルが含まれるグループの他のセルすべてからこの値を取り除き、この値を (選択肢から) 取り消します。

この例の他のルールで、セルに入力可能な値を減らしていきます。**"naked pair"**、**"hidden pair in row"**、**"hidden pair in column"** および **"hidden pair in square"** のルールでは、候補の絞り込みはできますが、回答を得ることはできません。**"X-wings in rows"**、"`X-wings in columns"`、 **"intersection**

removal row" および **"intersection removal column"** のルールは、より精緻な絞り込みを実行します。

数独例のクラス
org.drools.examples.sudoku.swing パッケージには、以下のように、数独パズルのフレームワークを実装する主なクラスセットが含まれます。

- **SudokuGridModel** は、9x9 グリッドの **Cell** オブジェクトとして数独パズルを格納するために実装可能なインターフェースを定義しています。
- **SudokuGridView** クラスは Swing コンポーネントで **SudokuGridModel** クラス実装の視覚化が可能です。
- **SudokuGridEvent** および **SudokuGridListener** クラスは、モデルとビューの間のステータスの変化をやり取りするために使用します。セルの値が解決または変更されると、イベントが実行されます。
- **SudokuGridSamples** クラスは、デモ目的に一部入力されている数独パズルを複数提供します。



注記

このパッケージには、Red Hat Process Automation Manager ライブラリーの依存関係は含まれません。

org.drools.examples.sudoku パッケージには、以下のように、基本的な **Cell** オブジェクトと各種アグリゲーションを実装する主なクラスセットが含まれます。

- **CellRow**、**CellCol** および **CellSqr**のサブクラスを含む **CellFile** クラス。これらすべては、**CellGroup** クラスのサブタイプになります。
- **Cell** と **CellGroup** は **SetOfNine** のサブクラスで、**Set<Integer>** 型の **free** プロパティーを提供します。**Cell** クラスは、個別の候補セットを表します。**CellGroup** は、セルの全候補セットの統合 (割り当ての必要のある数値セット) です。
  数独の例には、81 個の **Cell** と 27 個の **CellGroup** オブジェクト、 **Cell** プロパティーの **cellRow**、**cellCol** および **cellSqr** が提供するリンク、**CellGroup** プロパティー **cells** (**Cell** オブジェクトリスト) が提供するリストが含まれます。これらのコンポーネントを使用して、セルに値を割り当てたり、候補セットから値を取り除いたりできるように、特定の状態を検出するルールを記述できます。
- **Setting** クラスを使用して、値の割り当てに伴うオペレーションをトリガーします。**Setting** ファクトは、整合性の取れない中間の状態に対して反応しないように、新しい状況を検出する全ルールに配置して使用します。
- **Stepping** クラスは、優先順位が低いルールに使用して、**"Step"** が予期なく中断された場合に緊急停止を行います。この動作は、プログラムでパズルを解決できないということです。
- Main クラス **org.drools.examples.sudoku.SudokuExample** は、全コンポーネントを統合する Java アプリケーションを実装します。

数独の検証ルール (validate.drl)
数独の例の **validate.drl** ファイルには、セルグループで数が重複している状況を検出する検証ルールが含まれます。このグループは、**"validate"** アジェンダグループに統合され、ユーザーがパズルを読み込むと、明示的にルールをアクティベートできます。

"duplicate in cell …" の 3 つのルールの **when** 条件はすべて以下の方法で機能します。

- このルールの最初の条件で、割り当てられた値でセルを特定します。
- このルールの 2 番目の条件では、3 つのセルグループのどれかを所属先にプルします。
- 最終条件は、ルールに従い、最初のセル、同じ行、列、または四角に入る値と同じセル (上記のセル以外) を検索します。

ルール "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
```

ルール **"terminate group"** は最後に実行されます。このルールは、メッセージを出力して、シーケンスを停止します。

ルール "terminate group"

```
rule "terminate group"
    salience -100
  when
  then
    System.out.println( "Validation complete." );
    drools.halt();
end
```

数独の解決ルール (sudoku.drl)
数独の例の **sudoku.drl** ファイルには、3 種類の ルールタイプが含まれます。1 つ目のグループは、セルへの数値の割り当てを処理して、もう 1 つは実行可能な割り当てを検出して、3 つ目は候補セットからの値を削除します。

"set a value"、**"eliminate a value from Cell"** および **"retract setting"** のルールは、**Setting** オブジェ

クトの有無により左右されます。最初のルールは、セルへの割り当てと、3 つのセルグループの **free** セットから値を削除する操作を処理します。また、ゼロの場合には、このグループはカウンターを 1 つ減らし、**fireUntilHalt()** を呼び出した Java アプリケーションに制御を戻します。

"eliminate a value from Cell" ルールの目的は、新たに割り当てられたセルに関連する全セルの候補リストを絞り込むことです。最後に、すべての除外が完了したら、**"retract setting"** ルールにより、トリガーされている **Setting** ファクトを取り消します。

ルール "set a value"、"eliminate a value from a Cell" および "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
```

解決ルール 2 つを使用して、セルに数字を割り当てることができる状況を検出します。**"single"** のルールは、**Cell** に、数字が 1 つだけの候補セットが含まれる場合に実行されます。**"hidden single"** ルールは、候補が 1 つだけのセルが存在しない場合に実行されますが、セルに候補が含まれる場合には、セルが所属する 3 つのグループの 1 つに含まれるその他すべてのセルに、この候補が存在しないということです。いずれのルールも **Setting** ファクトを作成して、挿入します。

ルール "single" および "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
```

最大グループからのルール (個別または 2 - 3 のグループ単位) は、数独パズルを手作業で解決するのに使用する、さまざまな解決手法を実装します。

"naked pair" ルールは、グループの 2 つのセルで、全く同じ候補セットでサイズ **2** のものを検出します。これらの 2 つの値は、対象グループの他の候補セットすべてから削除することができます。

ルール "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
```

3 番目のルール **"hidden pair in ..."** は、ルール **"naked pair"** と同様に機能します。ルールはグループの 2 つのセルで 2 つの数字を検出します。どの値もこのグループの他のセルには入りません。つまり、他の候補はすべて、隠れたペアを持つ 2 つのセルから削除します。

ルール "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
```

2 つのルールは行と列で **"X-wings"** を処理します。2 つの異なる行 (または列) で、ある値を入力できるセルが 2 つしかなく、これらの候補が同じ列 (または行) に入る場合に、この列 (または行) のこの値に対する他の候補は除外できます。これらのルールの 1 つに含まれるパターンシーケンスに従うと、**same** または **only** などの用語で都合よく表現されている条件は、適切な制約が付けられたパターンになるか、**not** の接頭辞が付きます。

ルール "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
```

"intersection removal ..." の 2 つのルールは、1 つの四角の中に (1 つの行または列) 使用できる数字を制限するというルールに基づいています。つまり、この番号は行または列の中の 2-3 セルの 1 つに入っていないといけないのです。グループの別のセルすべての中にある候補セットから削除できます。このパターンは、発生制限を確立して、同じセルファイルの中かつ、四角の外のセルそれぞれに対して実行されます。

ルール "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
```

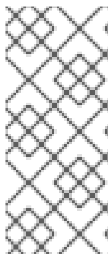
これらのルールは、すべてではありませんが、多くの数独パズルでは十分です。非常に難度の高いグリッドを解決するには、ルールセットにはさらに複雑なルールが必要です (最終的には、パズルは試行錯誤でしか解決できません)。

9.9. CONWAY の GAME OF LIFE のデシジョン例 (ルールフローグループおよび GUI 統合)

John Conway による有名なセルオートマトン (CA: Cellular automation) をベースにした Conway の Game of Life のデシジョンセットの例では、ルールでルールフローグループを使用してルール実行を制御する方法を説明しています。また、この例では、Red Hat Process Automation Manager ルールとグラフィカルユーザーインターフェース (GUI) の統合方法も説明しています。今回は、Conway の Game of Life を Swing ベースで実装しています。

以下は、Conway の Game of Life の例の概要です。

- **名前**: **conway**
- **Main クラス**: (**src/main/java** 内の) **org.drools.examples.conway.ConwayRuleFlowGroupRun**、**org.drools.examples.conway.ConwayAgendaGroupRun**
- **モジュール**: **droolsjbpm-integration-examples**
- **タイプ**: Java アプリケーション
- **ルールファイル**: (**src/main/resources** 内の) **org.drools.examples.conway.*.drl**
- **目的**: ルールフローグループと GUI 統合を例示します。



注記

Conway の Game of Life の例は、Red Hat Process Automation Manager に含まれる他のデシジョンセットの例の多くとは異なり、Red Hat カスタマーポータル から取得する **Red Hat Process Automation Manager 7.8.0 Source Distribution**の **~/rhpam-7.8.0-sources/src/droolsjbpm-integration-$VERSION/droolsjbpm-integration-examples** に配置されています。

Conway の Game of Life では、初期設定または定義済みのプロパティーで高度なパターンを作成して、初期状態からどのように進化していくかを観察することで、ユーザーはゲームと対話します。ゲームの目的は、世代ごとに人口の成長を表示します。各世代は、すべてのセル (細胞) が同時に進化していき、

前の世代をもとにして生み出されます。

以下の基本的なルールで、次の世代がどのようになるかを制御していきます。

- 生きているセルの近傍に、生きているセルが 2 個未満の場合は、孤独で死んでしまう。
- 生きているセルの近傍に、生きているセルが 4 個以上ある場合は、過密で死んでしまう。
- 死亡したセルの近傍に、生きているセルがちょうど 3 つある場合には、このセルは生き返る。

この基準のいずれも満たさないセルは、そのまま次の世代に残ります。

Conway の Game of Life の例は、**ruleflow-group** 属性が含まれる Red Hat Process Automation Manager ルールで、ゲームに実装されているパターンを定義します。この例には、アジェンダグループを使用して同じ動作を行うデシジョンセットのバージョンも含まれています。アジェンダグループは、デシジョンエンジンアジェンダをパーティションして、ルールのグループを実行制御できるようにします。デフォルトでは、すべてのルールがアジェンダグループ **MAIN** に含まれています。ルールに異なるアジェンダグループを指定するには、**agenda-group** 属性を使用できます。

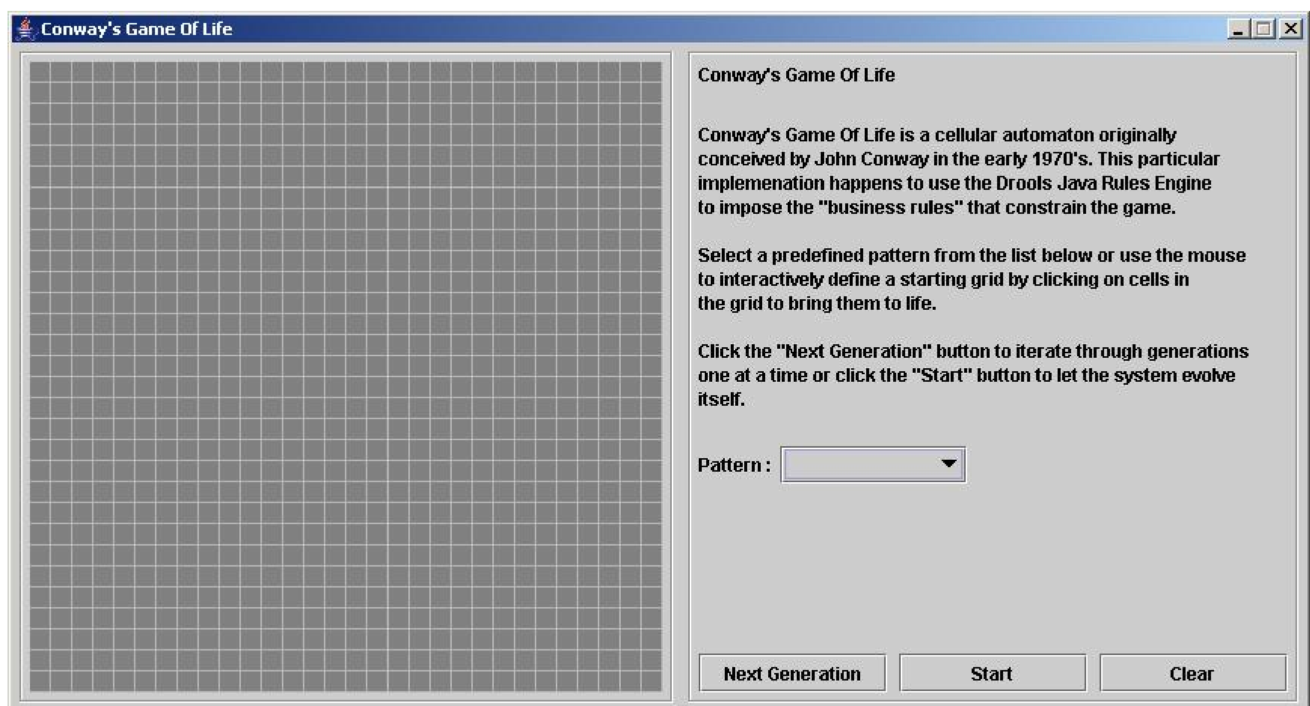
この概要では、Conway の例でアジェンダグループを使用したバージョンには触れません。アジェンダグループの詳細情報は、特にアジェンダグループについて対応している Red Hat Process Automation Manager のデシジョンセットの例を参照してください。

Conway 例の実行および対話

他の Red Hat Process Automation Manager のデシジョン例と同じように、お使いの IDE で **org.drools.examples.conway.ConwayRuleFlowGroupRun** クラスを Java アプリケーションとして実行し、Conway の例を実行します。

Conway の例を実行すると、**Conway’s Game of Life** GUI ウィンドウが表示されます。このウィンドウには、空のグリッドまたは "アリーナ" が含まれており、ここで生命のシミュレーションが行われます。システムにまだ生きているセルが含まれていないので、グリッドは最初は空白です。

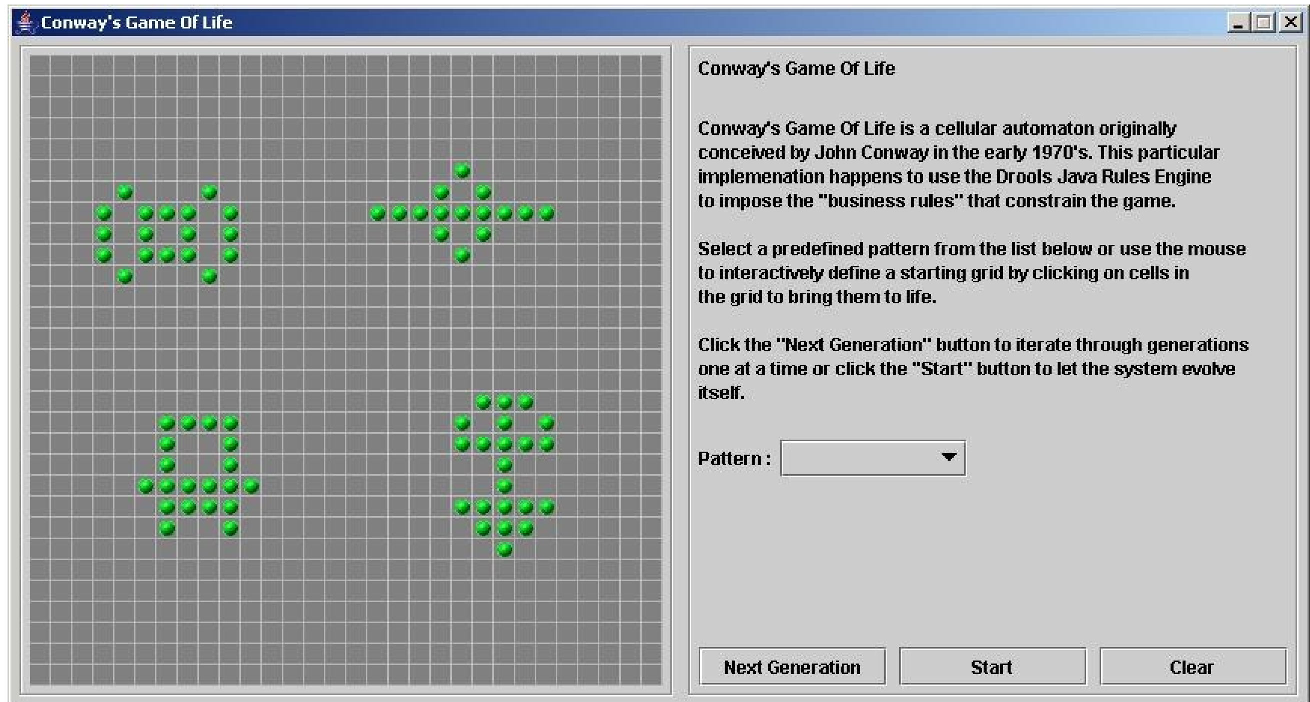
図9.24 起動後における Conway の例の GUI



パターン のドロップダウンメニューから事前定義済みのパターンを選択して、**次の世代** をクリックし、各人口の世代をクリックしていきます。セルは生きているか、死んでいるかのどちらかで、生きて

いるセルには緑のボールが含まれます。最初のパターンから人口が進化するにつれ、ゲームのルールをもとに、セルが近傍のセルに合わせて、生存するか、死亡していきます。

図9.25 Conway の例の世代進化



近傍には、上下左右のセルだけでなく対角線上につながっているセルも含まれるので、各セルには合計 8 つの近傍があります。例外は、角のセルと 4 辺上にあるセルで、それぞれ順に近傍が 3 つだけと、5 つだけになります。

セルをクリックすることで手動で介入して、セルを作成することも、死亡させることもできます。

最初のパターンから自動的に進化を実行するには、**スタート** をクリックします。

ルールグループを使用する Conway 例のルール

ConwayRuleFlowGroupRun の例のルールは、ルールフローグループを使用して、ルール実行を制御します。ルールフローグループは、**ruleflow-group** ルール属性に関連付けられたルールのグループです。これらのルールは、このグループがアクティベートされたときにしか実行されません。グループ自体は、ルールフローの図の詳細がグループを表すノードに到達してからでないと、有効化されません。

Conway の例では、ルールに以下のルールフローグループを使用します。

- **"register neighbor"**
- **"evaluate"**
- **"calculate"**
- **"reset calculate"**
- **"birth"**
- **"kill"**
- **"kill all"**

Cell オブジェクトはすべて、KIE セッションに挿入され、**"register neighbor"** ルールフローグループの **"register ..."** ルールがルールフロープロセスにより実行できるようになります。4 つのルールが含まれ

るこのグループは、セル同士の **Neighbor** の関係と、北東、北、北西、西の近傍との Neighbour の関係を作り出します。

この関係は双方向で、他の 4 方向を処理します。各辺上のセルは、特別な対応は必要ありません。これらのセルは、近傍のセルがなければペアは作成されません。

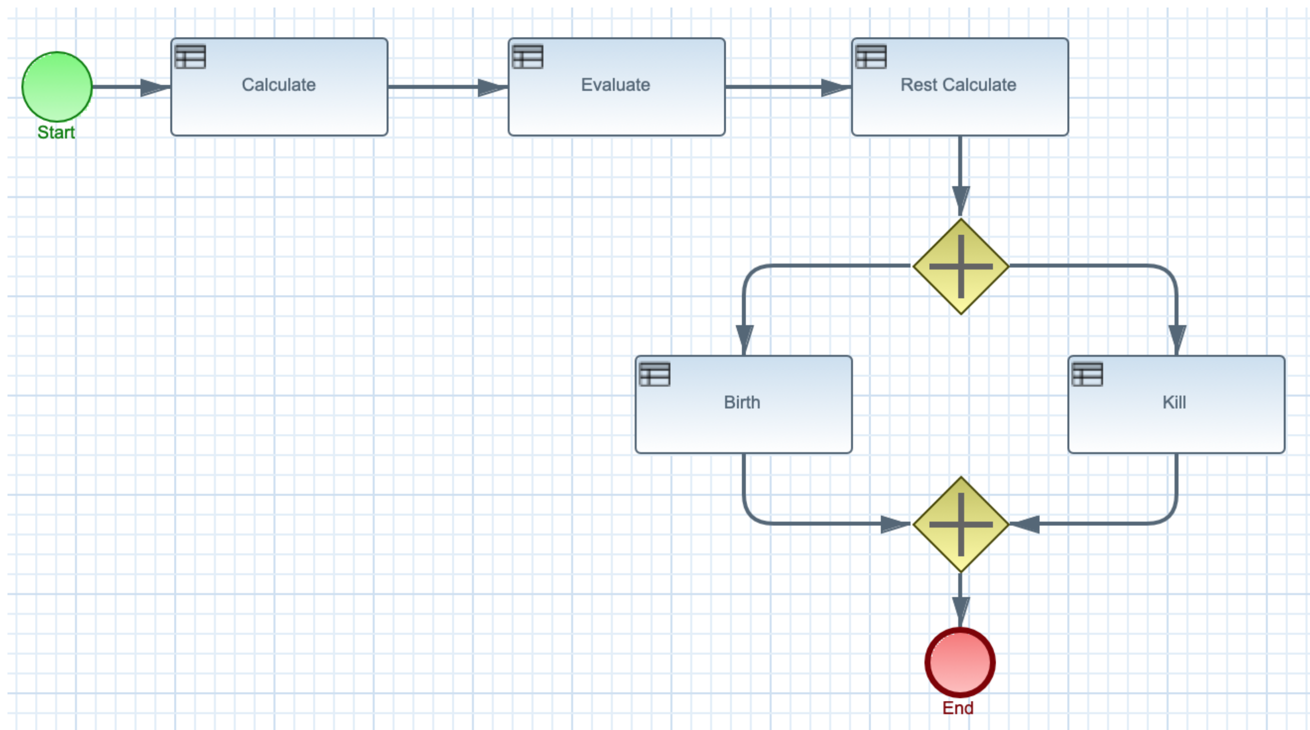
これらのルールに対して、すべてのアクティベーションが実行されるまで、全セルは、近傍の全セルと関係があります。

ルール "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
```

全セルが挿入されたら、Java コードはグリッドにパターンを適用し、特定のセルを **Live** に設定します。次に、ユーザーが **スタート** または **次の世代** をクリックすると、**Generation** のルールフローが実行されます。このルールフローは、世代のサイクルごとにセルの変更をすべて管理します。

図9.26 世代のルールフロー



ルールフロープロセスは、実行可能なグループに **"evaluate"** ルールフローグループおよび有効なルールを追加します。このグループの **"Kill the ..."** と **"Give Birth"** ルールを使用して、細胞の誕生または死亡セルにゲームのルールを適用します。この例では、**phase** 属性を使用して、特定のルールグループで **Cell** オブジェクトの理由付けをトリガーします。通常は、phase はルールフロープロセスの定義に含まれるルールフローグループに紐づけされています。

この例では、変更の適用前に評価を完全に完了しておく必要があるので、この時点では **Cell** オブジェクトの状態は変更されません。細胞の **phase** を **Phase.KILL** または **Phase.BIRTH** に適用し、後ほど **Cell** オブジェクトに適用されたアクションを制御するのに使用します。

ルール "Kill the ..." および "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
```

グリッド内の全 **Cell** オブジェクトが評価されると、この例では **"reset calculate"** ルールを使用して **"calculate"** ルールフローグループのアクティベーションを消去します。次に、ルールグループがアクティベートされると、**"kill"** と **"birth"** のルールを有効にするルールフローに分岐を挿入します。これらのルールにより状態の変更が適用されます。

ルール "reset calculate"、"kill" および "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
```

■

この段階では、複数の **Cell** オブジェクトの状態が **LIVE** または **DEAD** のいずれかに変更されています。この例では、細胞が生存または死亡すると、**"Calculate ..."** ルールの **Neighbor** 関係を使用して、周辺の細胞すべてに繰り返し実行し、**liveNeighbor** の数が増減します。数が変更された細胞は、**EVALUATE** フェーズに設定され、ルールフロー処理の評価段階の理由付けに含められるようにします。

生存数が判断され、全細胞に設定されると、ルールフロー処理が終了します。ユーザーが最初に **スタート** をクリックすると、その時点でデシジョンエンジンにより、ルールフローが再起動され、ユーザーが最初に **次の世代** をクリックした場合には、ユーザーは別の世代を要求することができます。

ルール "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 のデシジョン例 (後向き連鎖および再帰)

House of Doom のデシジョンセットの例では、デシジョンエンジンが後向き連鎖と再帰を使用して、階層システムで定義した目的やサブゴールに到達する方法を説明しています。

以下は House of Doom の例の概要です。

- **名前**: **backwardchaining**
- **Main クラス**: (**src/main/java** 内の) **org.drools.examples.backwardchaining.HouseOfDoomMain**
- **モジュール**: **drools-examples**
- **タイプ**: Java アプリケーション

- **ルールファイル**: (**src/main/resources** 内の) **org.drools.examples.backwardchaining.BC-Example.drl**
- **目的**: 後向き連鎖と再帰を例示します。

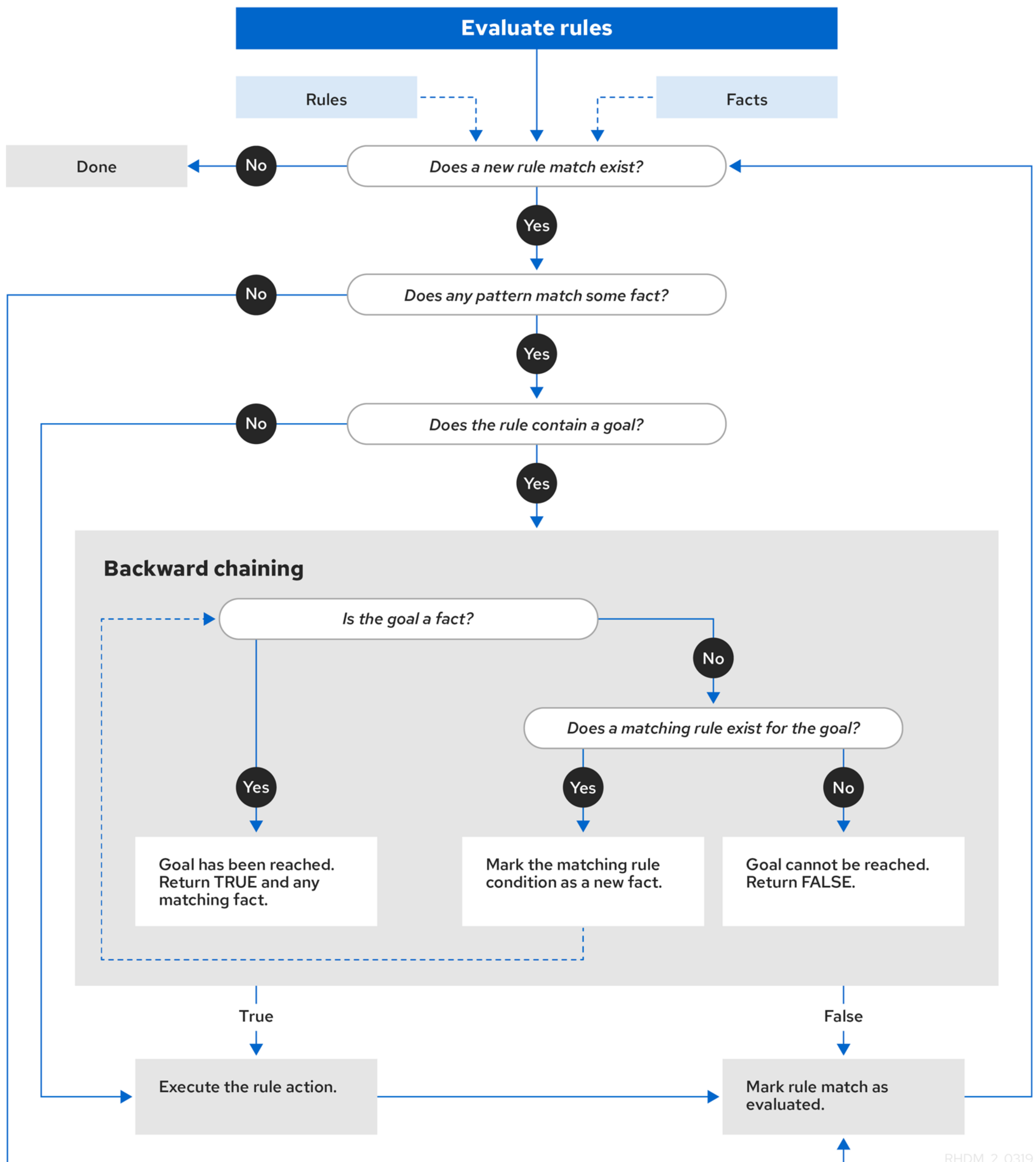
後向き連鎖のルールシステムは、通常再帰を使用して、デシジョンエンジンが満たそうとする結論から開始する目的駆動型のシステムです。システムが結論または目的に到達できない場合には、サブとなる目的、つまり、現在の目的の一部を完了する結論を検索します。システムは、最初の結論が満たされるか、すべてのサブとなる目的が満たされるまで続行されます。

反対に、前向き連鎖のルールシステムは、デシジョンエンジンのワーキングメモリーにあるファクトで開始して、そのファクトへの変更に反応するデータ駆動型のシステムです。オブジェクトがワーキングメモリーに挿入されると、その変更の結果として True となってルールの条件が、アジェンダにより実行がスケジュールされます。

Red Hat Process Automation Manager のデシジョンエンジンは、前向き連鎖と後向き連鎖の両方を使用してルールを評価します。

以下の図は、デシジョンエンジンが、ロジックフローで後向き連鎖のセグメントと、前向き連鎖全体とを使用してルールを評価する方法を例示します。

図9.27 前向き連鎖と後向き連鎖を使用したルール評価のロジック



House of Doom の例は、さまざまなクエリータイプが含まれるルールを使用し、部屋の場所と家の中のアイテムを探し出します。**Location.java** のサンプルクラスには、この例で使用する **item** と **location** 要素が含まれます。**HouseOfDoomMain.java** のサンプルクラスで、家の該当の場所にアイテムまたは部屋を挿入して、ルールを実行します。

HouseOfDoomMain.java クラスでのアイテムと場所

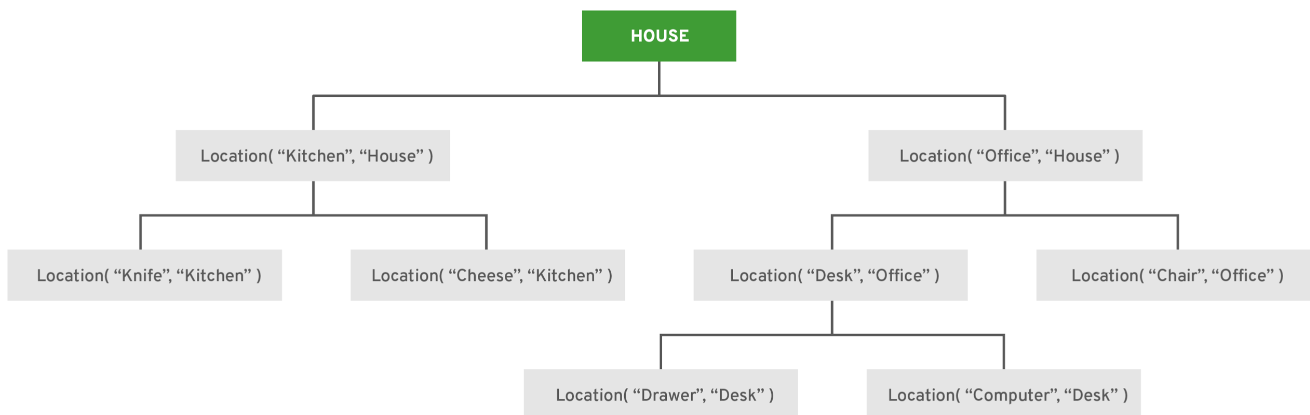
```
ksession.insert( new Location("Office", "House") );
ksession.insert( new Location("Kitchen", "House") );
ksession.insert( new Location("Knife", "Kitchen") );
ksession.insert( new Location("Cheese", "Kitchen") );
```

```
ksession.insert( new Location("Desk", "Office") );
ksession.insert( new Location("Chair", "Office") );
ksession.insert( new Location("Computer", "Desk") );
ksession.insert( new Location("Drawer", "Desk") );
```

ルールの例では、家の構造の中で全アイテムおよび部屋の場所を判断するのに、後向き連鎖と再帰を使用します。

以下の図は、House of Doom の構造と、その構造内のアイテムと部屋を示しています。

図9.28 House of Doom の構造



この例を実行するには、IDE で Java アプリケーションとして **org.drools.examples.backwardchaining.HouseOfDoomMain** クラスを実行します。

実行後に、以下の出力が IDE コンソールウィンドウに表示されます。

IDE コンソールでの実行出力

```
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
```

この例のルールはすべて実行されて、家の全アイテムの場所を検出し、家の中の全アイテムの場所を検出して、出力でそれぞれの場所を出力します。

再帰クエリーおよび関連のルール
再帰クエリーは、要素間の関係におけるデータ構造階層を使用して繰り返し検索を行います。

House of Doom の例では、**BC-Example.drl** ファイルに、この例のルールの大半が使用する **isContainedIn** クエリーが含まれており、家のデータ構造を再帰的に評価して、デシジョンエンジンに挿入するデータがないかを確認します。

BC-Example.drl の再帰クエリー

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

"go" のルールは、システムに挿入する文字列をすべて出力し、アイテムをどのように導入し、**"go1"** ルールが **isContainedIn** クエリーを呼び出すかを判断します。

ルール "go" および "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
```

この例は、**"go1"** 文字列をデシジョンエンジンに挿入して、**"go1"** ルールを有効にし、**House** の場所にある **Office** アイテムを検出します。

文字列の挿入とルールの実行

```
ksession.insert( "go1" );
ksession.fireAllRules();
```

IDE コンソールでの ルール "go1" の出力

```
go1
Office is in the House
```

推移閉包ルール

推移閉包は、階層構造で複数レベル、上層にある親要素に含まれる要素間の関係です。

"go2" ルールは、**Drawer** と **House** の推移閉包の関係を特定します。**Drawer** は、**House** の中の、**Office** の中の、**Desk** の中にあります。

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

この例は、**"go2"** 文字列をデシジョンエンジンに挿入して、**"go2"** ルールを有効化し、最終的に **House** の場所に含まれる **Drawer** アイテムを検出します。

文字列の挿入とルールの実行

```
ksession.insert( "go2" );
ksession.fireAllRules();
```

IDE コンソールのルール "go2" の出力

```
go2
Drawer is in the House
```

デシジョンエンジンは、以下のロジックをもとにこの結果を判断します。

1. クエリーは再帰的に、家の中の複数レベルを検索して、**Drawer** と **House** の間の推移閉包を検出します。
2. **Drawer** は **House** に直接含まれないので、**Location(x, y;)** を使用する代わりに、このクエリーは **(z, y;)** の値を使用します。
3. **z** の引数は現在バインドされておらず、値が指定されていないので、引数に含まれるものはすべて返されます。
4. **y** の引数は現在、**House** にバインドされているので、**z** は **Office** と **Kitchen** を返します。

5. クエリーは、**Office** からの情報を収集して、**Drawer** が **Office** に含まれているかを再帰的にチェックします。これらのパラメーターに対して、クエリーの行 **isContainedIn(x, z;)** が呼び出されます。

6. **Office** に直接含まれる **Drawer** が存在しないので、一致するものはありません。

7. **z** のバインドがない場合は、このクエリーでは **Office** 内のデータが返され、**z == Desk** と判断されます。

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

8. **isContainedIn** クエリーは再帰的に 3 回検索し、3 回目に、このクエリーにより **Desk** の中に **Drawer** があることが検出されます。

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

9. 最初の場所で上記が一致した後に、このクエリーにより再帰的に構造を上方向に検索し、**Drawer** が **Desk** の中に、**Desk** が **Office** の中に、**Office** が **House** の中にあることを判断します。このように、**Drawer** は **House** の中にあるので、このルールは満たされます。

リアクティブクエリールール

リアクティブクエリーでは、データ構造の階層を検索して、要素間に関係があるかを確認し、構造内の要素が変更されると動的に更新されます。

"go3" ルールは、リアクティブクエリーとして機能し、推移閉包により、新しいアイテム **Key** が **Office** に含まれるかどうかを検出します (**Office** の中の **Key** の中の **Drawer** など)。

ルール "go3"

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

この例は、**"go3"** 文字列をデシジョンエンジンに挿入して、**"go3"** ルールを有効化します。最初は、**Key** が家の構造に存在するので、このルールは満たされないため、出力は生成されません。

文字列の挿入とルールの実行

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

IDE コンソールのルール "go3" の出力 (条件を満たさない)

```
go3
```

この例では、**Office** の中にある **Drawer** の場所に、新しいアイテム **Key** を挿入します。この変更で、**"go3"** ルールの推移閉包が満たされ、それに合わせて出力が生成されます。

新規アイテムの場所の挿入とルールの実行

```
ksession.insert( new Location("Key", "Drawer") );
ksession.fireAllRules();
```

IDE コンソールのルール "go3" の出力 (条件を満たす)

```
オフィス内の鍵
```

またこの変更で、クエリーにより、後に続く再帰検索に含まれるよう、この構造に別のレベルが追加されます。

ルールにバインドなしの引数が含まれたクエリー
バインドなしの引数が 1 つ以上あるクエリーでは、クエリーの定義済み (バインドされている) 引数に含まれる未定義 (バインドされていない) アイテムすべてを返します。クエリー内の引数でバインドされているものがない場合には、クエリーはクエリーの範囲内のアイテムをすべて返します。

"go4" ルールは、バインドされている引数を使用して、**Office** 内の特定のアイテムを検索するのではなく、バインドされていない引数 **thing** を使用して、バインドされている引数 **Office** 内の全アイテムを検索します。

ルール "go4"

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

この例では **"go4"** 文字列をデシジョンエンジンに挿入して、**"go4"** ルールをアクティベートし、**Office** の全アイテムを返します。

文字列の挿入とルールの実行

```
ksession.insert( "go4" );
ksession.fireAllRules();
```

IDE コンソールのルール "go4" の出力

```
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" ルールは、バインドされていない引数 **thing** と **location** を使用して、**House** の全データ構造の中に含まれる全アイテムとその場所を検索します。

ルール "go5"

```
rule "go5"
  when
```

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

この例は **"go5"** 文字列をデシジョンエンジンに挿入して、**"go5"** ルールをアクティベートし、**House** データ構造に含まれる全アイテムとその場所を返します。

文字列の挿入とルールの実行

```
ksession.insert( "go5" );
ksession.fireAllRules();
```

IDE コンソールのルール "go5" の出力

```
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
```

第10章 デシジョンエンジン使用時のパフォーマンスチューニングに関する考慮点

以下の主要な概念または推奨のプラクティスを使用すると、デシジョンエンジンのパフォーマンス最適化に役立ちます。本セクションでこの概念についてまとめており、随時、他のドキュメントを相互参照して詳細を説明します。本セクションは、Red Hat Process Automation Manager の新しいリリースで、必要に応じて拡張または変更します。

デシジョンエンジンの重要な更新を必要としない、ステートレス KIE セッションには順次モードを使用する

順次モードは、デシジョンエンジンにおける高度なルールベースの設定です。このモードでは、デシジョンエンジンは、作業メモリーの変更に関係なく、デシジョンエンジンアジェンダに記載されている順番にルールを 1 回評価できます。このため、ルールの実行は順次モードの方が速くなりますが、重要な更新はルールに適用されない可能性があります。順次モードは、ステートレス KIE セッションのみに適用されます。
順次モードを有効にするには、システムプロパティー **drools.sequential** を **true** に設定します。

順次モードや、このモードの有効化に関する他のオプションの情報は、「Phreak における順次モード」を参照してください。

イベントリスナーを使用する場合は簡単な操作を使用する

イベントリスナーの数や、実行する操作の種類を制限します。デバッグロギングや設定プロパティーなどの簡単な操作にイベントリスナーを使用します。リスナーでネットワーク呼び出しなどの複雑な操作を行うと、ルールの実行が阻害される可能性があります。KIE セッションでの作業が完了した後には、セッションを消去できるように、アタッチされているイベントリスナーを以下の例のように削除します。

使用後に削除されるイベントリスナーの例

```
Listener listener = ...;
StatelessKnowledgeSession ksession = createSession();
try {
    ksession.insert(fact);
    ksession.fireAllRules();
    ...
} finally {
    if (session != null) {
        ksession.detachListener(listener);
        ksession.dispose();
    }
}
```

同梱のイベントリスナーとデシジョンエンジンでのデバッグロギングに関する情報は、8章 *デシジョンエンジンのイベントリスナーおよびデバッグロギング*を参照してください。

実行可能なモデルビルドの LambdaIntrospector のキャッシュサイズ設定

実行可能なモデルビルドで使用される **LambdaIntrospector.methodFingerprintsMap** キャッシュのサイズを設定できます。キャッシュのデフォルトサイズは **32** です。キャッシュサイズに小さい値を設定すると、メモリーの使用量が減少します。たとえば、システムプロパティー **drools.lambda.introspector.cache.size** を **0** に設定すると、メモリーの使用量を最小限に抑えることができます。キャッシュサイズが小さくなると、ビルドパフォーマンスが低下する点に注意してください。

実行可能モデルでの lambda 外部化の使用

lambda 外部化を有効にして、実行時のメモリーの消費量を最適化します。この設定は、実行可能モデルで生成され使用される lambda を書き換えるので、同じ lambda を全パターンおよび同じ制約で複数回再利用できます。 rete または phreak がインスタンス化されると、実行可能モデルはガベージコレクション機能を使用できるようになります。
実行可能モデルの lambda 外部化を有効にするには、以下のプロパティーを追加します。

```
-Ddrools.externaliseCanonicalModelLambda=true
```

第11章 関連資料

- 『Red Hat Process Automation Manager のデシジョン管理アーキテクチャーの設計』
- 『デシジョンサービスのスタートガイド』
- 『DRL ルールを使用したデシジョンサービスの作成』
- 『Red Hat Process Automation Manager プロジェクトのパッケージ化およびデプロイ』

付録A バージョン情報

本書の最終更新日: 2020 年 9 月 8 日 (木)