Creating intermediate facts in WebSphere ILOG JRules using synthetic objects

Skill Level: Intermediate

Raj Rao (rrao2@us.ibm.com)
ILOG Solution Architect
IBM

10 Nov 2010

This article uses a scenario to introduce the intermediate fact pattern in business rule processing and describes a technique in IBM® WebSphere® ILOG® JRules to create intermediate facts using synthetic objects in the business object model.

Introduction

IBM WebSphere ILOG JRules is IBM's Business Rule Management System (BRMS) that enables business users to dynamically control automated business decisions with business rules. Under the hood, WebSphere ILOG JRules employs a forward-chaining inference engine. Typically, an inference engine infers several **intermediate facts** while progressing toward the final goal. These intermediate facts are posted to the working memory for use by other rules. Being transient in nature, intermediate facts are not part of input or output data.

This article outlines a scenario where intermediate objects are desirable, and describes a technique for creating intermediate facts as **synthetic objects**. Along the way, alternatives to this technique are also described, along with a technique for creating JRules enumerations in the business object model.

This article is intended for the intermediate WebSphere ILOG JRules developer with a basic understanding of the WebSphere ILOG JRules product from a developer's perspective. See Resources for links to help you acquire the background information you'll need. The product versions used for the purpose of this article are:

- IBM WebSphere ILOG Rule Studio V7.1.0

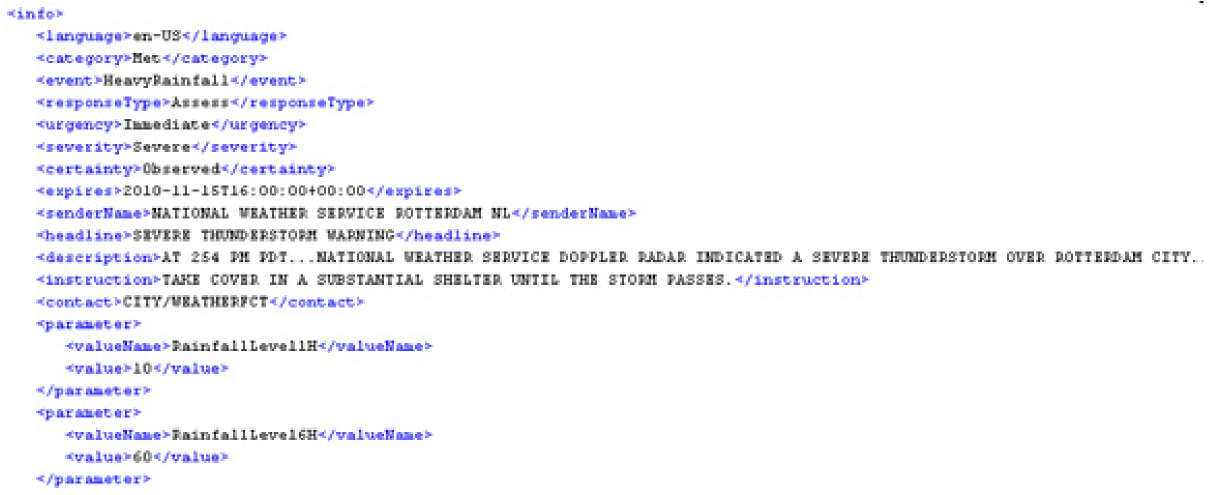
- IBM WebSphere ILOG Rule Execution Server V7.1.0

Sample scenario

Before beginning with the implementation of synthetic objects, let's review the sample scenario that will be used as reference point for this article, and also look at what is referred to as an intermediate fact.

This example uses a rule engine to handle severe weather alerts and to generate notifications and directives based on these alerts. The input and output to this rule engine conform to the Common Alerting Protocol (CAP), an XML specification produced by OASIS/ITU-T. The input data (Figure 1) contains information about weather alerts, such as rainfall, fog, and so on.

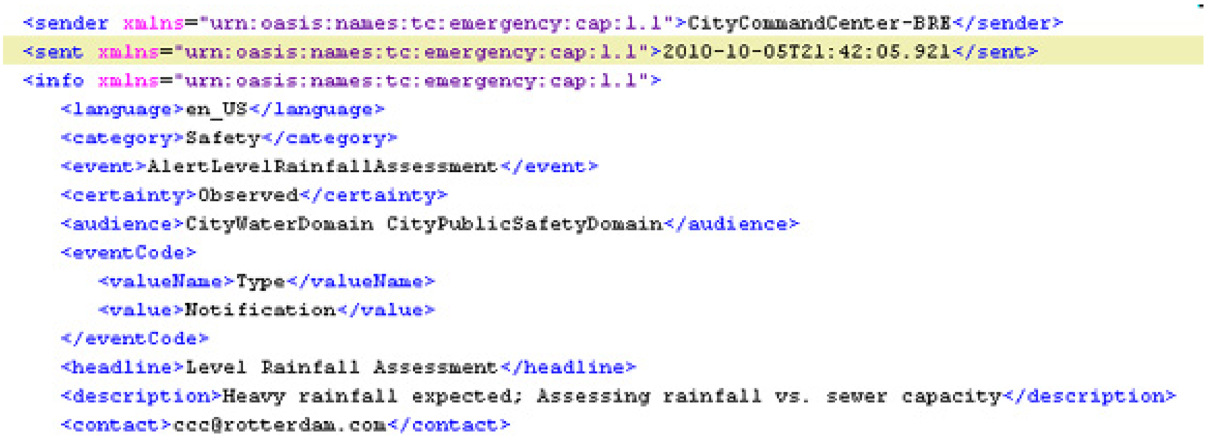
Figure 1. Input fragment



```
<info>
   <language>en-US</language>
   <category>Met</category>
   <event>HeavyRainfall</event>
   <responseType>Assess</responseType>
   <urgency>Immediate</urgency>
   <severity>Severe</severity>
   <certainty>Observed</certainty>
   <expires>2010-11-15T16:00:00+00:00</expires>
   <senderName>NATIONAL WEATHER SERVICE ROTTERDAM NL</senderName>
   <headline>SEVERE THUNDERSTORM WARNING</headline>
   <description>AT 254 PM PDT...NATIONAL WEATHER SERVICE DOPPLER RADAR INDICATED A SEVERE THUNDERSTORM OVER ROTTERDAM CITY..
   <instruction>TAKE COVER IN A SUBSTANTIAL SHELTER UNTIL THE STORM PASSES.</instruction>
   <contact>CITY/WEATHERFCT</contact>
   <parameter>
      <valueName>RainfallLevel1H</valueName>
      <value>10</value>
   </parameter>
   <parameter>
      <valueName>RainfallLevel6H</valueName>
      <value>60</value>
   </parameter>
```

Based on these alerts, the rule engine generates directives to various city departments, as seen in the output fragment (Figure 2).

Figure 2. Output fragment



```
<sender xmlns="urn:oasis:names:tc:emergency:cap:1.1">CityCommandCenter-BRE</sender>
<sent xmlns="urn:oasis:names:tc:emergency:cap:1.1">2010-10-05T21:42:05.921</sent>
<info xmlns="urn:oasis:names:tc:emergency:cap:1.1">
   <language>en_US</language>
   <category>Safety</category>
   <event>AlertLevelRainfallAssessment</event>
   <certainty>Observed</certainty>
   <audience>CityWaterDomain CityPublicSafetyDomain</audience>
   <eventCode>
      <valueName>Type</valueName>
      <value>Notification</value>
   </eventCode>
   <headline>Level Rainfall Assessment</headline>
   <description>Heavy rainfall expected; Assessing rainfall vs. sewer capacity</description>
   <contact>ccc@rotterdam.com</contact>
```

Broadly, the business rules used in this application fall into two categories:

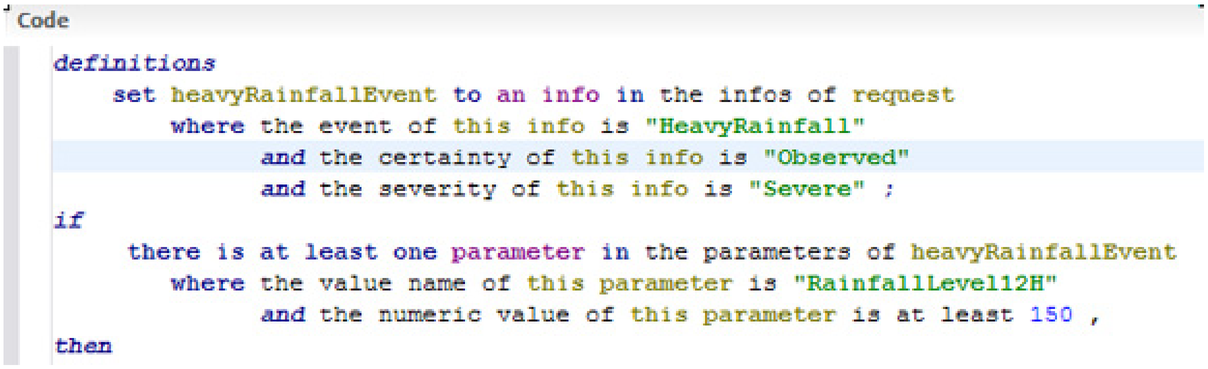
- Assessment rules that analyze and assess the alert information.
- Directive rules that generate notifications and directives based on the assessment.

For example, rules might assess that it is **HeavyRainfall** in any of these scenarios:

- Observed rainfall in the last 1 hour is over 15 mm.
- Observed rainfall in the last 1 hour is over 10 mm and the observed rainfall in the last 12 hours is over 100 mm.
- Observed rainfall in the last 12 hours is over 150 mm.
- Observed rainfall in the last 6 hours is over 80 mm.

These conditions are captured in individual rules, as shown in Figure 3.

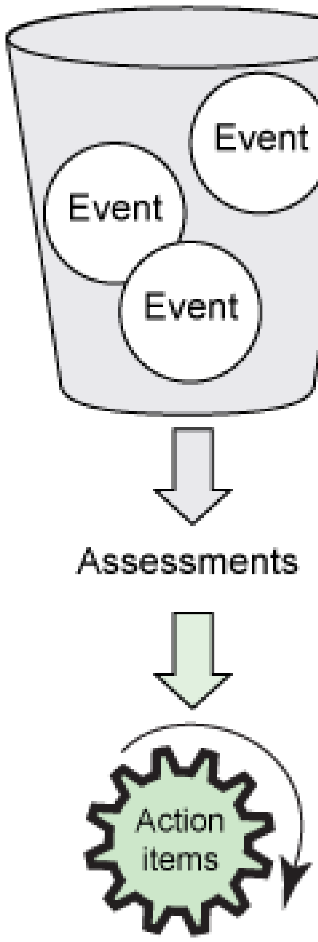
Figure 3. Example of assessment rule conditions


Code

```
definitions
    set heavyRainfallEvent to an info in the infos of request
        where the event of this info is "HeavyRainfall"
              and the certainty of this info is "Observed"
              and the severity of this info is "Severe" ;
if
     there is at least one parameter in the parameters of heavyRainfallEvent
        where the value name of this parameter is "RainfallLevel12H"
              and the numeric value of this parameter is at least 150 ,
then
```

Based on the assessment of HeavyRainfall, several notifications and directives are issued to several municipal departments, such as City Public Safety, City Sewage, and so on. Be aware that regardless of how this assessment was concluded -- whether it is based on the observed rainfall over the last hour, or the last six hours, or any of the other criteria -- the same actions are taken. However, these actions should not be repeated for each rule that assesses HeavyRainfall because that would lead to duplication of code and reduced maintainability of rules. Therefore, it is desirable for rules to generate an intermediate assessment before executing the actions (Figure 4). This **intermediate fact** would need to indicate the assessment type and hold a reference to the event that generated this fact, since some of the event information is used in generating the action items.

Figure 4. Event processing using intermediate assessments



Implementation options

The CAP 1.1 XSD file is supplied as the execution object model (XOM) and the business object model (BOM) is generated from it. To easily integrate with the Enterprise Service Bus (ESB), the rule engine is deployed as a Web service. This is done by leveraging the Hosted Transparent Decision Services (HTDS) capability of WebSphere ILOG JRules. HTDS precludes the use of a Java™ XOM. Additionally, it would be a convoluted exercise to reference an XML element in the CAP XSD from a Java class. Therefore, creating the intermediate facts as a Java XOM is not an option, and becomes somewhat of a challenge.

Two techniques can be used to overcome this challenge:

- **Option A: Supplemental XSD for intermediate facts**
  A new supplemental XSD is created to define the intermediate fact schema. For example, the XSD shown in Listing 1 creates a complex type named **Fact** that contains an AssessmentType and a reference to the

other parameters to be stored in the Fact. This XSD is loaded into WebSphere ILOG Rule Studio as a new BOM entry and can now be instantiated to store intermediate facts.

Listing 1

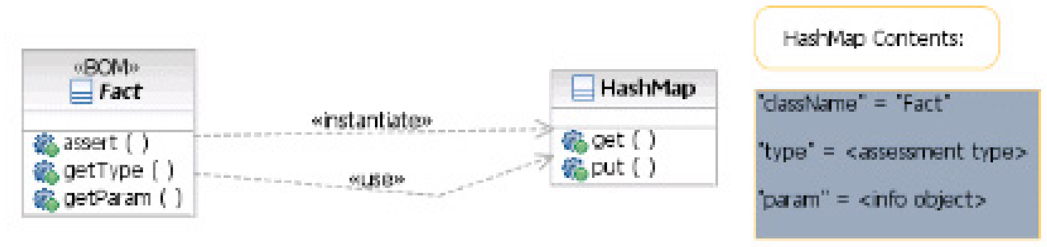
```
<?xml version="1.0" encoding="UTF-8"?>
<schema attributeFormDefault="unqualified" elementFormDefault="qualified"
        targetNamespace="urn:www.ibm.com:software:ilog"
        xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:cap="urn:oasis:names:tc:emergency:cap:1.1"
        xmlns:ilog="urn:www.ibm.com:software:ilog">

        <import namespace="urn:oasis:names:tc:emergency:cap:1.1"
                schemaLocation="capAlertMessage1.1.xsd" />

        <complexType name="Fact">
                <sequence>
                        <element name="type" type="ilog:AssessmentType"/>
                        <element ref="<add element reference here>"/>
                </sequence>
        </complexType>

        <simpleType name="AssessmentType">
                <restriction base="string">
                        <enumeration value="HEAVY_RAINFALL"/>
                        <enumeration value="HEAVY SNOW"/>
                        <enumeration value="HEAVY FOG"/>
                        <enumeration value="EXTREME COLD"/>
                </restriction>
        </simpleType>
        </schema>
```

This is a relatively simple and straightforward approach, but it could suffer from a few deficiencies under some scenarios. Being a reference to a domain element, the parameter creates a cross-xsd dependency and has potential namespace issues. Additionally, as is the case in this scenario, if the referred element is a nested element without a visible qualified name, it is not possible to refer to it from the Fact complex type.

- **Option B: Use synthetic objects for intermediate facts**
  Synthetic objects, also known as virtual objects, are those objects that are not defined in the XOM. They are BOM constructs that simulate the state of real objects. They are essentially an application of the decorator pattern wrapping a core Java type, such as String or Map. Virtual methods, which are defined in IRL code, provide the decoration.

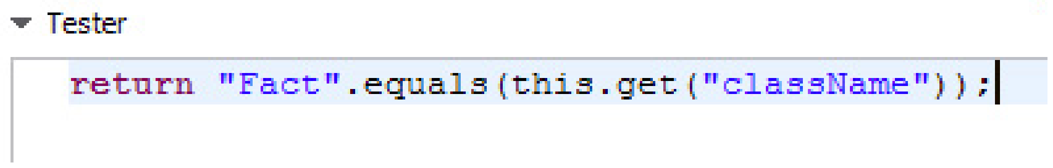
  For this scenario, a Fact BOM class is created which instantiates a HashMap when the fact is asserted. Here, Fact is a synthetic object with no corresponding class in the XOM. It merely wraps around the HashMap and simulates a real business object. Rules can be written on this synthetic business object just like any other real business object.

  Figure 5. Synthetic fact



When a fact is asserted, it sets className as "Fact" in the HashMap. A "Tester" is used to map the business class (Fact) to the HashMap. For this scenario, the tester filters out all maps that do not have "Fact" as the className (Figure 6).

Figure 6. Fact tester



```
return "Fact".equals(this.get("className"));
```

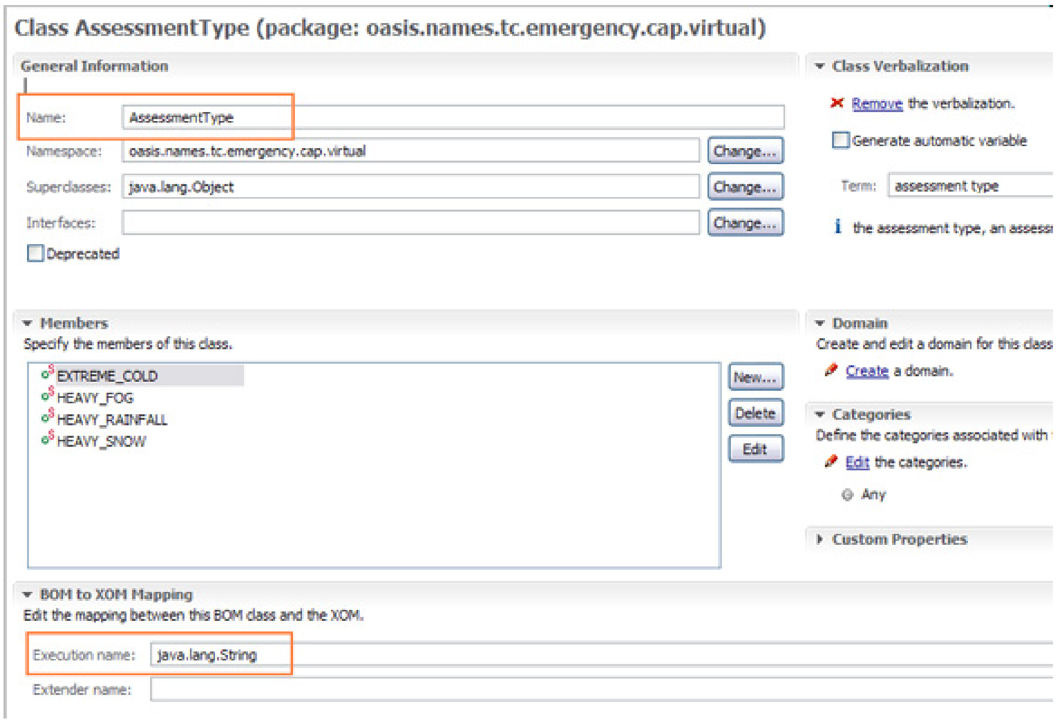
The implementation details of creating the synthetic objects are covered next.

Synthetic object implementation details

The technique employed here uses synthetic objects instead of regular Java objects as application data that is used for execution. BOM to XOM mapping is used to implement the synthetic objects.

Business object model

1. In the BOM Editor, create an enumeration called `AssessmentType`. This is an enumeration of the different intermediate assessments that rules can make, such as HEAVY_RAINFALL, EXTREME_COLD, and so on (Figure 7). Set the **Execution name** in the BOM to XOM mapping section to `java.lang.String`.
Figure 7. Create assessment type



Class AssessmentType (package: oasis.names.tc.emergency.cap.virtual)

General Information

Name:	AssessmentType
Namespace:	oasis.names.tc.emergency.cap.virtual
Superclasses:	java.lang.Object
Interfaces:	

☐ Deprecated

Class Verbalization

Remove the verbalization.

☐ Generate automatic variable

Term: assessment type

the assessment type, an assess

Members

Specify the members of this class.

EXTREME_COLD
HEAVY_FOG
HEAVY_RAINFALL
HEAVY_SNOW

Domain

Create and edit a domain for this class

Create a domain.

Categories

Define the categories associated with

Edit the categories.

Any

Custom Properties

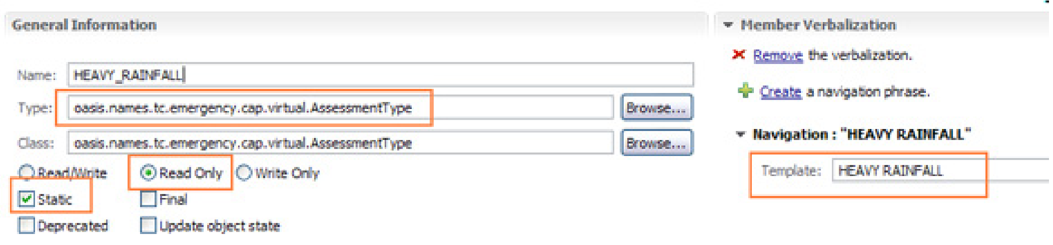
BOM to XOM Mapping

Edit the mapping between this BOM class and the XOM.

Execution name:	java.lang.String
Extender name:	

2. Define each enumerated value as an attribute of type AssessmentType and provide a BOM-to-XOM mapping with the value. Mark each of these attributes **Read Only** and **Static**, and provide an appropriate verbalization (Figure 8).

Figure 8. Enumerations for assessment type



General Information

Name:	HEAVY_RAINFALL
Type:	oasis.names.tc.emergency.cap.virtual.AssessmentType
Class:	oasis.names.tc.emergency.cap.virtual.AssessmentType

○ Read/Write [✓ Read Only] ○ Write Only

☑ Static ☐ Final

☐ Deprecated ☐ Update object state

Member Verbalization

Remove the verbalization.

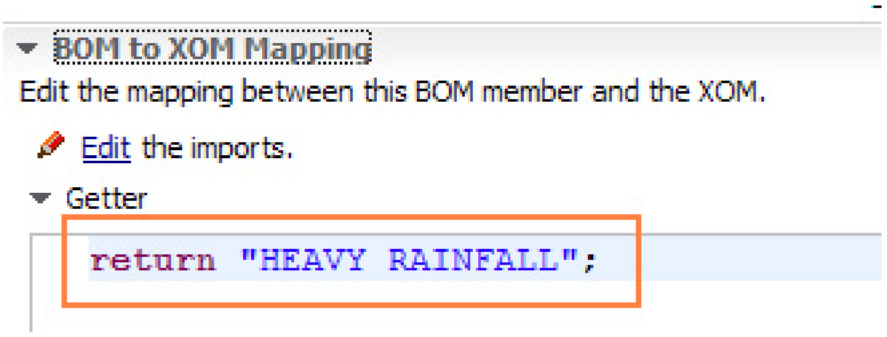
Create a navigation phrase.

Navigation : "HEAVY RAINFALL"

Template: HEAVY RAINFALL

3. Provide the enumeration value with a simple BOM-to-XOM mapping (Figure 9).

Figure 9. BOM to XOM for enumeration



BOM to XOM Mapping

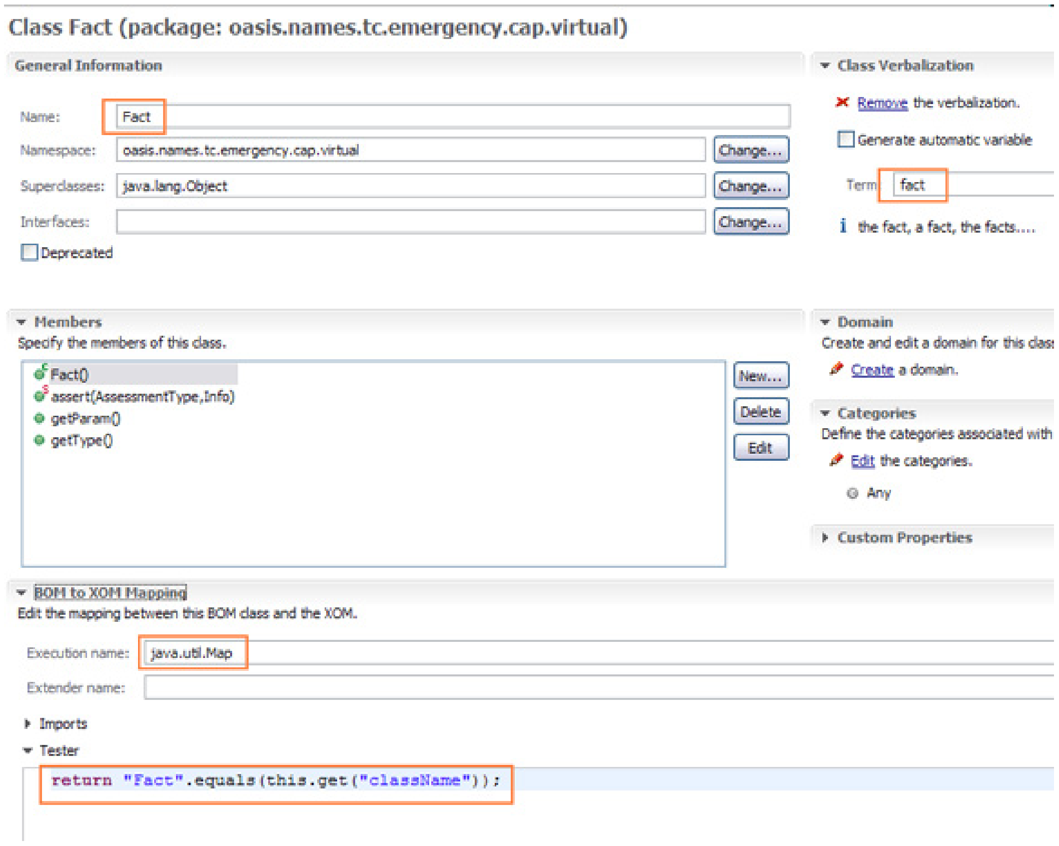
Edit the mapping between this BOM member and the XOM.

Edit the imports.

Getter

```
return "HEAVY RAINFALL";
```

4. You are now ready to create the synthetic class. In the BOM Editor, create a new BOM class called `Fact` (Figure 10). Set the **Execution name** in the BOM to XOM mapping section to `java.util.Map`.
Figure 10. Create Synthetic Class in BOM Editor



Class Fact (package: oasis.names.tc.emergency.cap.virtual)

General Information

Name: Fact
Namespace: oasis.names.tc.emergency.cap.virtual
Superclasses: java.lang.Object
Interfaces:
Deprecated

Class Verbalization

Remove the verbalization.
Generate automatic variable
Term: fact
the fact, a fact, the facts....

Members
Specify the members of this class.

Fact()
assert(AssessmentType,Info)
getParam()
getType()

Domain
Create and edit a domain for this class
Create a domain.

Categories
Define the categories associated with
Edit the categories.
Any

Custom Properties

BOM to XOM Mapping
Edit the mapping between this BOM class and the XOM.

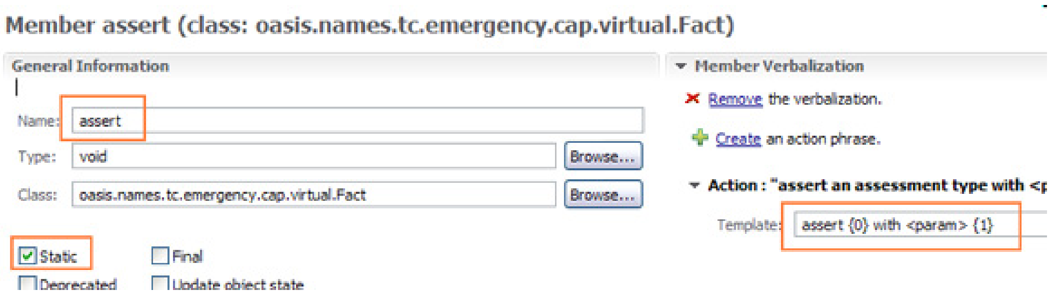
Execution name: java.util.Map
Extender name:

Imports
Tester

```
return "Fact".equals(this.get("className"));
```

5. The synthetic object basically is a HashMap that stored each of the attributes and values as name-value pairs in the HashMap. The synthetic object you need has two attributes:
   - **Type** is the assessment type
   - **Param** is a reference to the original event that created this assessment.

6. Define a method (called **assert** in this example) to create the fact and post it to the working memory. Mark it **Static** and provide a simple verbalization (Figure 11).
Figure 11. Create assert method on synthetic class



Member assert (class: oasis.names.tc.emergency.cap.virtual.Fact)

General Information

Name: assert

Type: void

Class: oasis.names.tc.emergency.cap.virtual.Fact

[✓ Static] Final

Deprecated Update object state

Member Verbalization

Remove the verbalization.

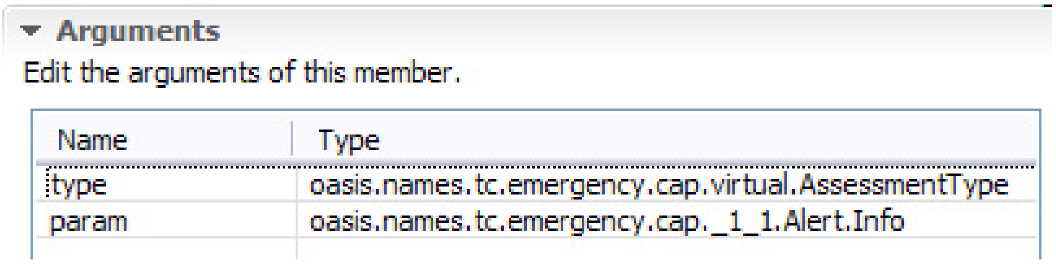
Create an action phrase.

Action : "assert an assessment type with <p

Template: assert {0} with <param> {1}

7. This method takes two arguments, the assessment type and a reference to the original event info (Figure 12).

Figure 12. Arguments for assert Method



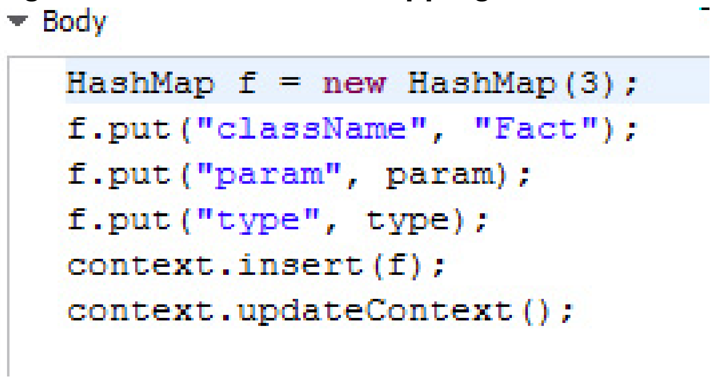
Arguments

Edit the arguments of this member.

Name	Type
type	oasis.names.tc.emergency.cap.virtual.AssessmentType
param	oasis.names.tc.emergency.cap._1_1.Alert.Info

8. In the BOM-to-XOM mapping, this method creates a HashMap and puts the parameters into the HashMap. In addition, it inserts the newly created object into working memory and updates the context to refresh the agenda (Figure 13).

Figure 13. BOM to XOM mapping for assert method

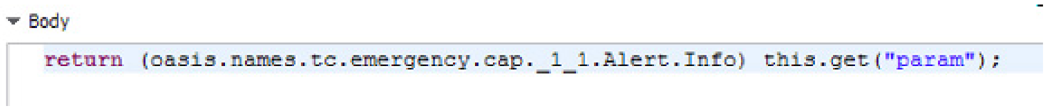


Body

```
HashMap f = new HashMap(3);
f.put("className", "Fact");
f.put("param", param);
f.put("type", type);
context.insert(f);
context.updateContext();
```

9. The two retrieval methods simply retrieve the value from the HashMap (Figure 14 and 15).

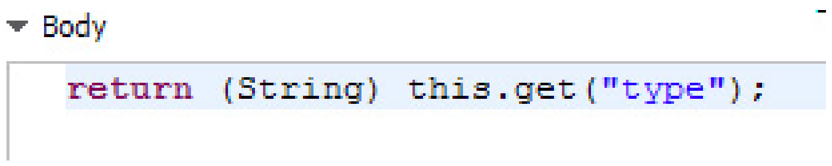
Figure 14. BOM to XOM for getParam method



Body

```
return (oasis.names.tc.emergency.cap._1_1.Alert.Info) this.get("param");
```

Figure 15. BOM to XOM for getType method


Body

```
return (String) this.get("type");
```

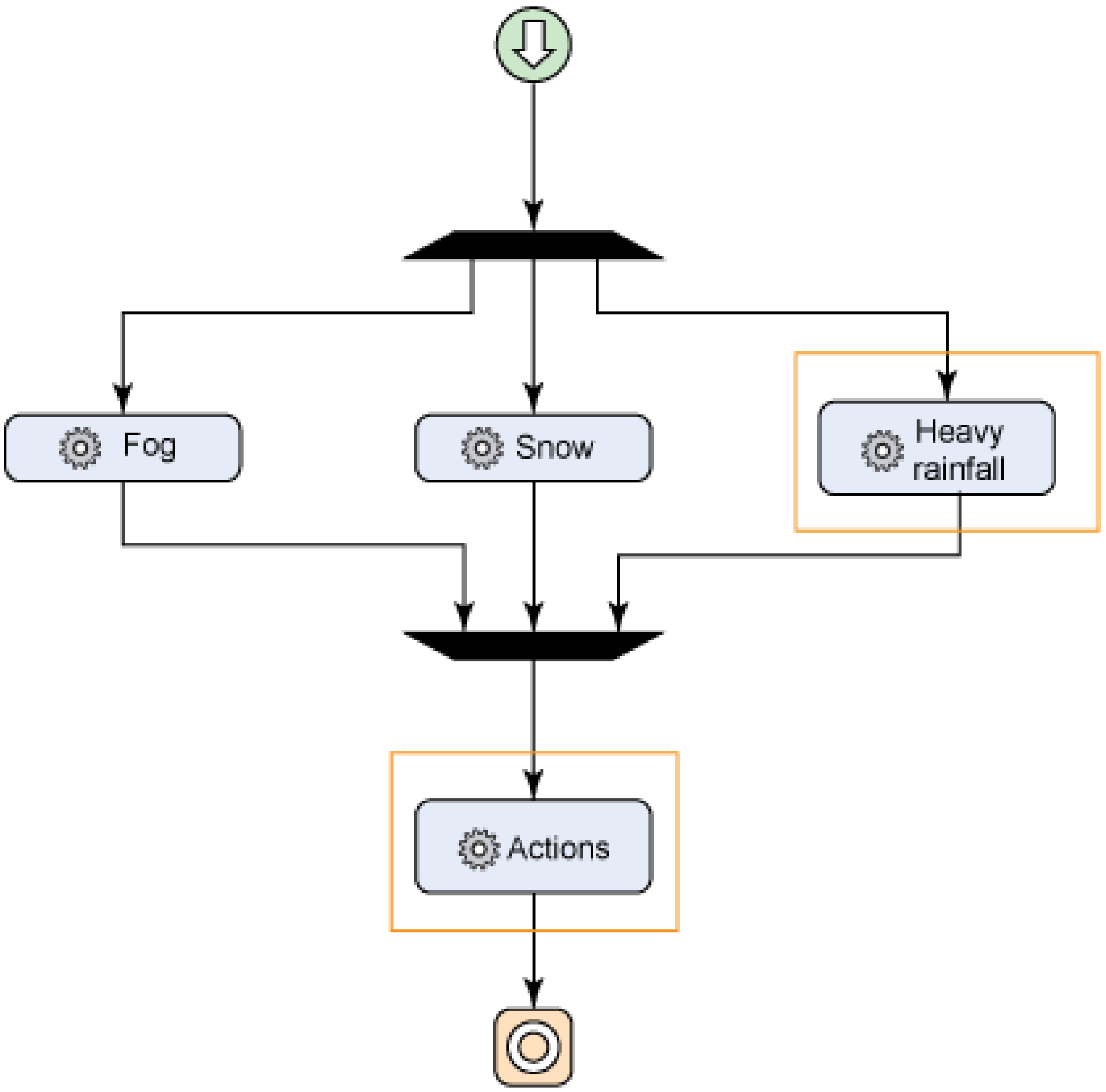
Rules

The overall ruleflow has two phases:

- During the assessment phase, the intermediate facts are derived.
- Based on these intermediate facts, notifications and other action items are created during the action phase.

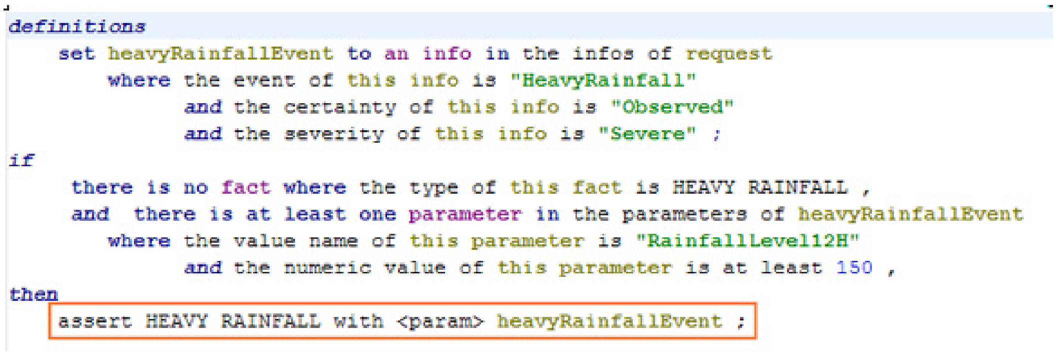
These phases are depicted in the ruleflow shown in Figure 16.

Figure 16. Overall ruleflow



- **Assessment rules**
  Assessment rules post intermediate facts based on incoming data. The example in Figure 17 posts a HEAVY RAINFALL assessment fact if more than 150 mm of rainfall is observed in the last 12 hours. The assert method creates the synthetic assessment fact and posts it to the working memory.

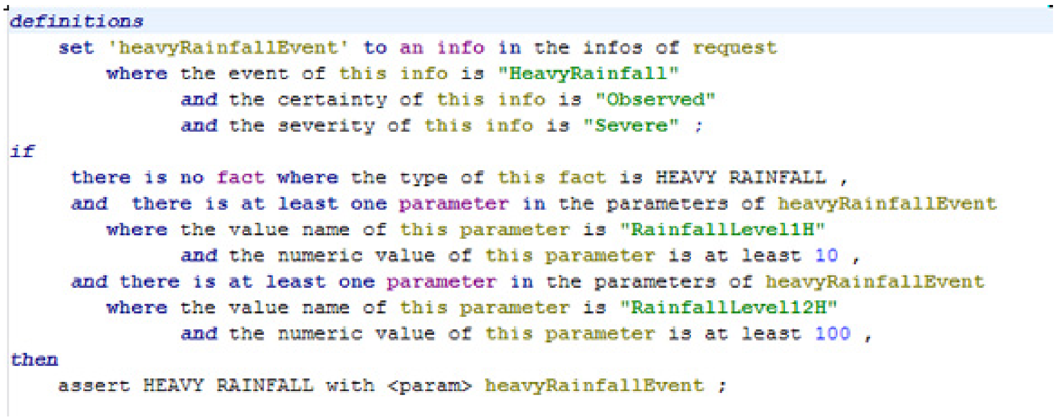
  Figure 17. Example rule based on: Rainfall over last 12 hours



```
definitions
    set heavyRainfallEvent to an info in the infos of request
        where the event of this info is "HeavyRainfall"
              and the certainty of this info is "Observed"
              and the severity of this info is "Severe" ;
if
     there is no fact where the type of this fact is HEAVY RAINFALL ,
     and  there is at least one parameter in the parameters of heavyRainfallEvent
        where the value name of this parameter is "RainfallLevel12H"
              and the numeric value of this parameter is at least 150 ,
then
    assert HEAVY RAINFALL with <param> heavyRainfallEvent ;
```

Another assessment rule checks the observed rainfall values over the last 1 hour and over the last 12 hours and suitably asserts the HEAVY RAINFALL assessment fact. To avoid multiple assessments of the same type being created, each of these assessment rules checks the working memory for the existence of an assessment fact (Figure 18).

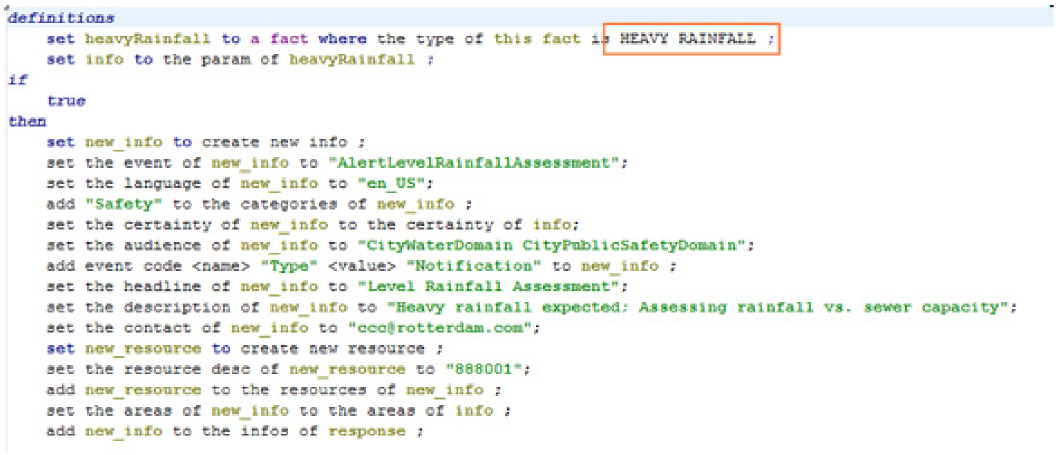
Figure 18. Example rule based on multiple rainfall parameters



```
definitions
    set 'heavyRainfallEvent' to an info in the infos of request
        where the event of this info is "HeavyRainfall"
              and the certainty of this info is "Observed"
              and the severity of this info is "Severe" ;
if
     there is no fact where the type of this fact is HEAVY RAINFALL ,
     and  there is at least one parameter in the parameters of heavyRainfallEvent
        where the value name of this parameter is "RainfallLevel1H"
              and the numeric value of this parameter is at least 10 ,
     and there is at least one parameter in the parameters of heavyRainfallEvent
        where the value name of this parameter is "RainfallLevel12H"
              and the numeric value of this parameter is at least 100 ,
then
    assert HEAVY RAINFALL with <param> heavyRainfallEvent ;
```

- **Action rules**
  Action rules match on the intermediate facts to generate action items, such as notifications and directives. Any number of events can lead to an assessment of HEAVY RAINFALL, but only one set of action items is created based on this assessment (Figure 19).

Figure 19. Rule to Create Action Items



```
definitions
    set heavyRainfall to a fact where the type of this fact is HEAVY RAINFALL ;
    set info to the param of heavyRainfall ;
if
    true
then
    set new_info to create new info ;
    set the event of new_info to "AlertLevelRainfallAssessment";
    set the language of new_info to "en_US";
    add "Safety" to the categories of new_info ;
    set the certainty of new_info to the certainty of info;
    set the audience of new_info to "CityWaterDomain CityPublicSafetyDomain";
    add event code <name> "Type" <value> "Notification" to new_info ;
    set the headline of new_info to "Level Rainfall Assessment";
    set the description of new_info to "Heavy rainfall expected; Assessing rainfall vs. sewer capacity";
    set the contact of new_info to "ccc@rotterdam.com";
    set new_resource to create new resource ;
    set the resource desc of new_resource to "888001";
    add new_resource to the resources of new_info ;
    set the areas of new_info to the areas of info ;
    add new_info to the infos of response ;
```

Conclusion

Creating intermediate facts is a useful pattern to avoid duplication of rule code. With the example described in this article, you have seen that synthetic objects can be created entirely from within the WebSphere JRules Editor. This technique for creating synthetic objects can be very useful, particularly when intermediate objects are desirable for rule processing, but Java execution object model is precluded by HTDS.

Resources

- WebSphere ILOG JRules Information Center
- WebSphere ILOG JRules product information
- Oasis Common Alerting Protocol 1.1
- IBM developerWorks WebSphere

About the author

Raj Rao
Rajesh (Raj) Rao has been working in the area of expert systems and business rule management systems for over 20 years, during which time he has applied business rules technology to build diagnostic, scheduling, qualification and configuration applications across various domains such as manufacturing, transportation and finance. He has been with IBM for close to 2 years. With a background in Artificial Intelligence, his interests include Natural Language Processing and Semantic Web.