hp Unified Correlation Analyzer

Unified Correlation Analyzer
for
Event Based Correlation

Version 3.2

Reference Guide

Edition: 1.0

For the HP-UX (11.31) and Linux (5.9 & 6.5) Operating Systems

April 2015

Legal Notices

Warranty

License Requirement and U.S. Government Legend

Copyright Notices

Trademark Notices

Contents

Preface **10**

Chapter 1 .... **13**

Introduction **13**

1.1 Overview 14
1.2 Value Pack concept 14

Chapter 2 .... **16**

Solution Architecture **16**

2.1 Solution Architecture 16
2.2 Components 17
2.3 Value Pack 18
2.4 Scenario 19
2.4.1 Filters 19
2.4.2 Inference Engine 20
2.4.3 Rules 20

Chapter 3 .... **22**

Value Packs and Scenarios **22**

3.1 Definitions 22
3.2 Value packs core features 22
3.2.1 Mediation Flow integration 22
3.2.2 Alarm collection and validation 23
3.3 Scenario core features 24
3.3.1 Alarm compression 24
3.3.2 Alarm or Event filtering 25
3.3.3 Alarm or Event enrichment 25
3.3.4 Alarm lifecycle 26
3.3.5 Automatic rules firing 31
3.4 Value Pack Definition 32
3.4.1 Files and distribution 32
3.4.2 Value pack definition file 32
3.5 Scenario 37
3.5.1 Files and distribution 37
3.5.2 Scenario definition file 38
3.5.3 Filter definition file 40
3.5.4 Filter tags file 47
3.5.5 Mappers definition file 51
3.5.6 Alarm Enrichment 57
3.5.7 Scenario Initialization class 58
3.5.8 Rules files 59
3.6 Value Pack Life Cycle 66
3.6.1 Installing a Value Pack 66
3.6.2 Deploying a Value Pack 66
3.6.3 Starting a Value Pack 67

3.6.4 Stopping a Value Pack ... 67
3.6.5 Un-deploying a Value Pack ... 68
3.6.6 Removing a Value Pack ... 68
3.7 Scenario Life Cycle and Status ... 69

Chapter 4 ... 70

Scenario Policies ... 70

4.1.1 Processing policies ... 70
4.1.2 Time-related settings ... 73
4.1.3 Alarm eligibility ... 74
4.1.4 Garbage Collection ... 75
4.1.5 Automatic handling of configuration files modifications ... 75

Chapter 5 ... 77

Common Objects ... 77

5.1 Events ... 77
5.1.1 Identification attributes ... 77
5.1.2 Orchestration attributes ... 77
5.1.3 Interfaces hierarchy ... 78
5.2 Alarms models used in the rules ... 80
5.2.1 Alarm ... 80
5.2.2 AlarmStateChange ... 83
5.2.3 AlarmAttributeValueChange ... 84
5.2.4 AlarmDeletion ... 86
5.2.5 Properties of alarms ... 87
5.3 Actions ... 91
5.3.1 Action Class ... 92
5.3.2 Action registry ... 96
5.3.3 Action callbacks ... 99
5.4 Flags ... 102
5.4.1 Flag ... 102
5.4.2 ScenarioInitFlag ... 103
5.4.3 TickFlag ... 104
5.4.4 SynchronizationFlag ... 105
5.4.5 Internal Flags ... 106
5.5 Groups ... 107
5.5.1 Group ... 108
5.5.2 PropagationGroup ... 109
5.6 State ... 110
5.7 LocalVariable ... 111
5.8 Watchdog Item ... 113
5.9 Watchdog Item callbacks ... 115
5.10 Rule Session ... 119
5.11 Collection Flows ... 122
5.12 Scenario Loggers ... 124

Chapter 6 ... 126

Advanced UCA for EBC features ... 126

6.1 Orchestration of event cascading between scenarios ... 126

6.1.1 Orchestration Principles 126
6.1.2 Orchestra Routes Configuration File 129
6.1.3 Orchestration definition example 132
6.1.4 Orchestration looping option 134
6.1.5 Scenario mode (STREAM/CLOUD) impact on event cascading 134
6.1.6 Orchestration API 137
6.2 Scenario Specific Configuration 141
6.2.1 Setting-up a Value pack specific Configuration file 142
6.3 Persisting alarms into a DB 146
6.3.1 Configuring the DB persistence feature 146
6.3.2 Accessing DB to retrieve persisted alarms 146

Appendix A 147
A.1 UCA for EBC REST API – DB Access 147
Indications for reading this guide 147
Getting started 147
Knowing which are the AlarmDao objects defined 147
Getting stored alarms 148
Doing actions on stored alarms 148
Storing new alarms 148
Deleting alarms 148
Registering for DB updates through REST 148
Registering for DB updates through NOM adapter 148
Advanced Requests 149
Optional arguments 149
A.2 UCA for EBC XML schemas 151

Glossary 152

Figures

Figure 1 - Simplified Solution architecture 16
Figure 2 - Extended Solution architecture 17
Figure 3 - Scenario 19
Figure 4 - Working Memory in STREAM mode 29
Figure 5 - Referencing a filter file inside a ValuePackConfiguration.xml file 41
Figure 6 - Referencing multiple filter files inside a ValuePackConfiguration.xml file 41
Figure 7 - Referencing a filter tags file inside a ValuePackConfiguration.xml file 48
Figure 8 - Editing Filter Tags with UCA for EBC Admin GUI 51
Figure 9 - Value Pack Life Cycle 66
Figure 10 - Scenario Lifecycle 69
Figure 11 - COPY Route example 127
Figure 12 - JOIN Rule example 129
Figure 13 - Cascading from STREAM to STREAM 135
Figure 14 - Cascading from CLOUD to STREAM 137

XML Configuration

XML Configuration 1 - ValuePackConfiguration.xml example 34
XML Configuration 2 - Scenario configuration example 40
XML Configuration 3 - Filter definition example 47
XML Configuration 4 - Filter definition example that contains tags 48
XML Configuration 5 - Template definition in Scenario 62
XML Configuration 6 - Rule Parameter file example 63
XML Configuration 7 – Enabling the CompressionMode for a Scenario 73
XML Configuration 8 - ActionRegistry.xml example 98
XML Configuration 9 - Orchestration configuration example 134
XML Configuration 10 - Specific Configuration Schema example 142
XML Configuration 11 - Specific Configuration XML file example.. 143
XML Configuration 12 - Binding Java Class example 144
XML Configuration 13 – XmlConfiguration Wrapper 145
XML Configuration 14 – Scenario Specific Configuration integration example..................... 145

Rule Samples

Rule Sample 1 - Using ‘justInserted’, ‘hasAVCChanged’, ‘hasStateChanged’ and ‘aboutToBeRetracted’ alarm flags 28
Rule Sample 2 - STREAM mode and Alarms 30
Rule Sample 3 - Template rules file main components 61
Rule Sample 4 - Template Rules file example 63
Rule Sample 5 - Rule generated from Template example 65
Rule Sample 6 - Alarm attributes 81
Rule Sample 7 - AlarmType example 87
Rule Sample 8 - PerceivedSeverity example 88
Rule Sample 9 - NetworkState example 88
Rule Sample 10 - OperatorState example 89
Rule Sample 11 - ProblemState example 90
Rule Sample 12 - CustomField example 91
Rule Sample 13 - Simple Action example (synchronous) 94
Rule Sample 14 - Simple Action example (asynchronous) 95
Rule Sample 15 - Action Callback example 101
Rule Sample 16 - Flag example 103
Rule Sample 17 - TickFlag example 105
Rule Sample 18 - SynchronizationFlag example 106
Rule Sample 19 - Group example 109
Rule Sample 20 - Group example 110
Rule Sample 21 - LocalVariable example 112
Rule Sample 22 - WatchdogItem example 115
Rule Sample 23 - WatchdogItem callback example 118
Rule Sample 24 - RuleSession (WorkingMemory) mandatory access 121
Rule Sample 25 – Mediation Flow example 124
Rule Sample 26 - Scenario logger example 125
Rule Sample 27 - Orchestraion API scenario rule example 139
Rule Sample 28 - Orchestraion API scenario rule for source scenario in JOIN example 140
Rule Sample 29 - Orchestraion API scenario rule for target scenario in JOIN example....... 141

Tables

Table 1 - Alarm Collection validation 23
Table 2 - Summary CLOUD versus STREAM mode 30
Table 3 - Value Pack distribution files 32
Table 4 - Value Pack properties 33
Table 5 - Mediation Flows properties 36
Table 6 - Scenario files distribution 38
Table 7 - Scenario properties 39
Table 8 - File structure of a deployed Value Pack 67
Table 9 - Event identification attributes and methods 77
Table 10 - Event orchestration attributes and methods 77
Table 11 - Alarm attributes and methods 81
Table 12 - Alarm status attributes and methods 81
Table 13 - Alarm Correlation attributes and methods 81
Table 14 - Alarm Association attributes and methods 81
Table 15 - AlarmStateChange identifier attributes and methods 83
Table 16 - AlarmStateChange general attributes and methods 83
Table 17 - AlarmAttributeValueChange identifier attributes and methods 84
Table 18 - AlarmAttributeValueChange general attributes and methods 85
Table 19 - AlarmDeletion identifier attributes and methods 86
Table 20 - AlarmDeletion general attributes and methods 86
Table 21 - AlarmType values 87
Table 22 - PerceivedSeverity values 88
Table 23 - NetworkState values 88
Table 24 - OperatorState values 89
Table 25 - ProblemState values 89
Table 26 - CustomFields attributes and methods 90
Table 27 - CustomField attributes and methods 90
Table 28 - AttributeChanges attributes and methods 91
Table 29 - AttributeChange attributes and methods 91
Table 30 - ActionRegistry - MediationValuePack properties 97
Table 31 - ActionRegistry – Action properties 97
Table 32 - Flags attributes and methods 102
Table 33 - Value Pack properties 107
Table 34 - Group attributes and methods 109
Table 35 - Group attributes and methods 110
Table 36 - LocalVariable methods 111
Table 37 - WatchdogItem usage 113
Table 38 - WatchdogItem Callback usage 116
Table 39 - Rule Session usage 121
Table 40 - OrchestraWorkflow Tag 131
Table 41 - Routes Tag 131
Table 42 - Route Tag 131
Table 43 - Orchestra COPY route properties 131
Table 44 - Orchestra JOIN route properties 131
Table 45 - Destinations Tag 132
Table 46 - Destination Tag 132
Table 47 - Sources Tag 132
Table 48 - Orchestra Scenario identification (<Source> or <Target> tags) 132
Table 49 - Orchestration method 138
Table 50 - JOIN (Convergence case) methods 138

Preface

This guide provides an overview of Unified Correlated Analyzer for Event Based Correlation (EBC) product and describes how to create Value Packs to target customer specific use cases.

Product Name: Unified Correlation Analyzer for Event Based Correlation (also referred in this document as UCA for EBC)

Product Version: 3.2

Kit Version: V3.2

Intended Audience

Here are some recommendations based on possible reader profiles:

- Solution Developers and integrators
- Software Development Engineers

Software Versions

The term UNIX is used as a generic reference to the operating system, unless otherwise specified.

The software versions referred to in this document are as follows:

Product Version	Supported Operating systems
UCA for Event Based Correlation Server Version V3.2	• HP-UX 11.31 for Itanium • Red Hat Enterprise Linux Server release 5.9 & 6.5
UCA for Event Based Channel Adapter Version V3.2	• HP-UX 11.31 for Itanium • Red Hat Enterprise Linux Server release 5.9 & 6.5
UCA for Event Based Correlation Software Development Kit Version V3.2	• Windows XP / Vista • Windows Server 2007 • Windows 7 • Red Hat Enterprise Linux Server release 5.9 & 6.5

Typographical Conventions

`Courier` Font:

- Source code and examples of file contents
- Commands that you enter on the screen

- Pathnames
- Keyboard key names

Italic Text:

- Filenames, programs and parameters
- The names of other documents referenced in this manual

Bold Text:

- To introduce new terms and to emphasize important words

Associated Documents

The following documents contain useful reference information:

References

[R1] *Unified Correlation Analyzer for Event Based Correlation – Installation Guide*

[R2] *Unified Correlation Analyzer for Event Based Correlation – Administration, Configuration and Troubleshooting Guide*

[R3] *Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Engine (C:\%UCA_EBC_DEV_HOME%\apidoc\uca-expert-engine\index.html)*

[R4] *Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Actions (C:\%UCA_EBC_DEV_HOME%\apidoc\uca-mediation-action-client\index.html)*

[R5] *JBoss Drools Expert guide –* http://docs.jboss.org/drools/release/5.5.0.Final/drools-expert-docs/html_single/index.html

[R6] *JBoss Drools Fusion guide –* http://docs.jboss.org/drools/release/5.5.0.Final/drools-fusion-docs/html_single/index.html

[R7] *Unified Correlation Analyzer for Event Based Correlation – Value Pack Development Guide*

[R8] *OSS Open Mediation V7.1 Functional Specification*

[R9] *OSS Open Mediation V7.1 Installation and Configuration Guide*

[R10] *Unified Correlation Analyzer for Event Based Correlation – User Interface Guide*

[R11] *Unified Correlation Analyzer for Event Based Correlation – Topology Extension*

[R12] Unified Correlation Analyzer for Event Based Correlation – Value Pack Examples

Support

Please visit our HP Software Support Online Web site at https://softwaresupport.hp.com/ for contact information, and details about HP Software products, services, and support.

The Software support area of the Software Web site includes the following:

- Downloadable documentation.
- Troubleshooting information.
- Patches and updates.
- Problem reporting.
- Training information.
- Support program information.

Chapter 1

Introduction

This guide describes the capabilities of the UCA for Event Based Correlation (UCA for EBC) software product.

This document is organized as follows:

Chapter 1 provides a product overview

Chapter 2 describes the high level architecture of the product and its main components.

Chapter 3, Chapter 4 and Chapter 5 describe some common notions:

- Value packs
- Scenarios
- Common Objects used in Working Memory
- Actions

Chapter 6 describes advanced features such as Cascading capabilities and Scenario Specific Configuration.

Note

Throughout this document, we use the `${UCA_EBC_HOME}` environment variable to reference the root directory ("static" part) of UCA for EBC. The default value for the `${UCA_EBC_HOME}` environment variable is `/opt/UCA-EBC`. The `${UCA_EBC_HOME}` environment variable thus references the `/opt/UCA-EBC` directory unless UCA for EBC "static" part has been installed in an alternate directory.

We also use `${UCA_EBC_DATA}` environment variable to reference the data directory ("variable" part) of UCA for EBC. The default value for the `${UCA_EBC_DATA}` environment variable is `/var/opt/UCA-EBC`. The `${UCA_EBC_DATA}` environment variable thus references the `/var/opt/UCA-EBC` directory unless UCA for EBC "variable" part has been installed in an alternate directory.

Since UCA-EBC V2.0, on Linux and HP-UX systems, the `${UCA_EBC_DATA}` directory may contain multiple instances of UCA-EBC. In this document, we will use the value `${UCA_EBC_INSTANCE}` for referring to `${UCA_EBC_DATA}/instances/<instance-name>` directory on Linux/HP-UX systems and to `${UCA_EBC_DATA}` on Windows systems.
Note that at installation time on Linux/HP-UX, a single <instance-name> is configured: default.

☞ For more information on how to install the UCA for EBC product, please refer to: [R1] *Unified Correlation Analyzer for Event Based Correlation – Installation Guide*.

1.1 Overview

The Unified Correlation Analyzer for Event Based Correlation product (also known as 'UCA Expert' by analogy with the legacy 'TeMIP Expert' Software) offers a new and generalized event based correlation solution.

Based on the JBoss Drools 5.5.0.Final "rule engine", UCA for EBC offers the capability to create comprehensive functional correlation sets called '**Value packs**' that implement the correlation logic. This correlation is performed by rules execution (the rules are written in a Java-based language). Any Value Pack can support/use predefined functionalities such as Alarm collection, filtering, life cycle as well as Generic Actions.

In terms of functionalities, UCA for EBC is able to:

Collect alarms (the Alarm model is based on a mix of X733 and OSS/J Fault Management Model) and map them into Operator Alarm model (Alarm)

Run several scenarios (rule engines) in parallel and/or in sequence in order to implement complex correlation algorithms. Each set of scenarios implementing a single correlation solution is grouped inside a UCA for EBC Value Pack.

Dispatch 'Alarm' objects to the different scenarios

Execute rules based on scenario input stream and generate suitable output (e.g. actions to external systems).

Control the scenario input stream using an Alarm based filtering layer.

Execute actions such as storing to a database, creating a Trouble Ticket, creating a new Alarm, group alarms, forward an alarm to another scenario or execute a Generic Action through the OSS Open Mediation V7.1 (NOM V7.1) layer.

Rules files are actually JBoss Rules files so both JBoss Expert and Fusion rules are supported in UCA for EBC rule files. JBoss Drools Expert and JBoss Drools Fusion are JBoss Drools basic modules.

On top of this basic functionality, UCA for EBC also provides a Software Development Kit (SDK) that allows solution developers to easily build UCA for EBC Value Packs (Functional Correlation block). Administration tools (both command-line and a GUI) are also available to manage, monitor and troubleshoot the product.

UCA for EBC can be connected with a mediation bus (OSS Open Mediation V7.1) providing the capability to collect alarms coming from any number of sources (NMS) and performing actions in return.

1.2 Value Pack concept

A Value Pack is conceptually a consistent set of correlation capabilities for managing some use cases such as a Low Level Filtering, a Domain-specific based correlation like IP MPLS or L2 Metro Ethernet or a simple "operator" use case that wants to group/correlate alarms based on specific criteria.

In terms of implementation, a Value Pack resides in a "functional container" that is made of one or more scenarios responsible for implementing the correlation logic (rules). Please note that the scenarios logic can be cascaded in order to implement more complex correlation algorithms.

Several Value Packs can also be deployed in the same UCA for EBC system. Additionally several versions of the same Value Pack can also be deployed on a UCA for EBC system.

The product comes with demonstration examples: the Low Level Event Filtering Value Pack and the Cascading Value Pack. Both of them can be deployed and used for demos (as is or customized) or used as templates to create new Value Packs.

HP also delivers out-of-the-box value packs with the Inference Machine Software Development Kit that bring Problem Detection and Topology State Propagator scenarios.

Chapter 2

Solution Architecture

2.1 Solution Architecture

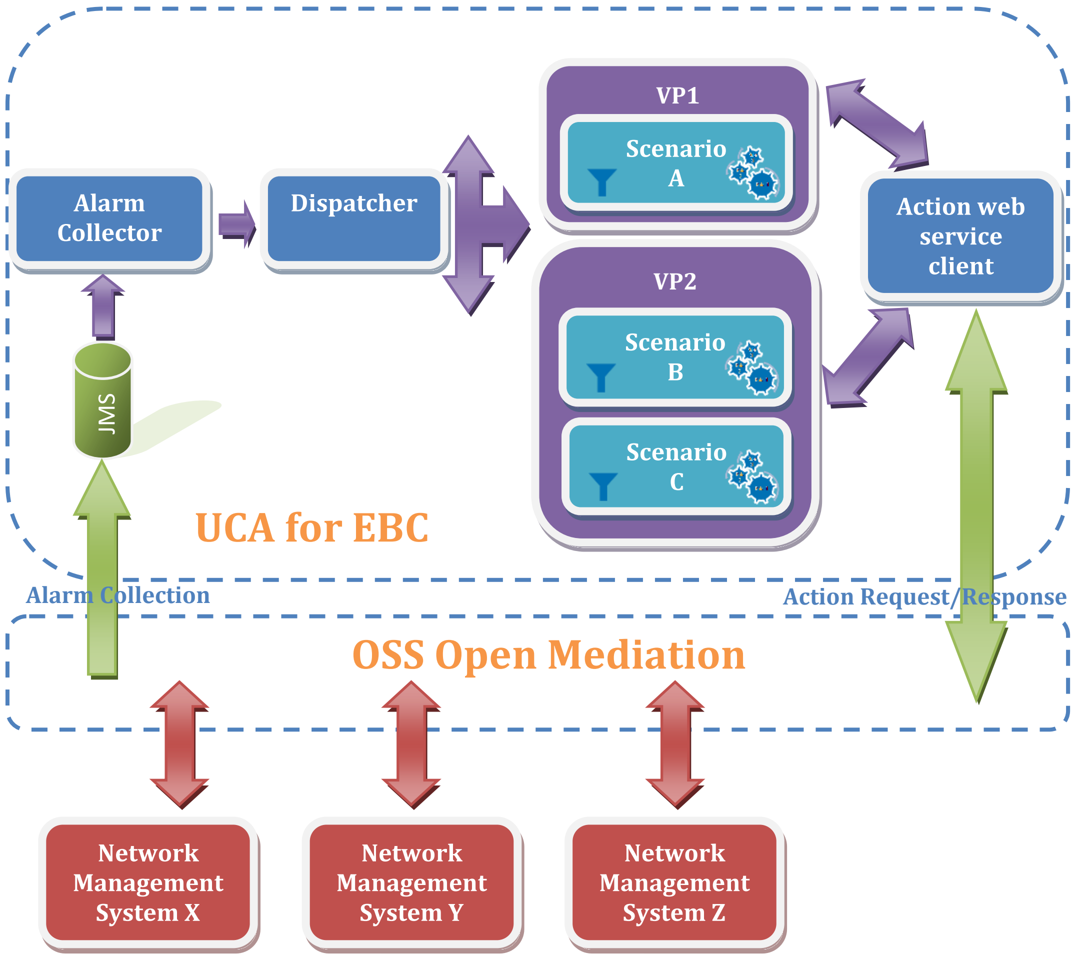


Figure 1 - Simplified Solution architecture

UCA for EBC provides the capability to:

- Collect alarms through OSS Open Mediation V7.1 (NOM V7.1).
- Dispatch these alarms to Value Packs, then to each scenario.
- Apply filters to each scenario and then run correlations rules on the inference engine associated with the scenario.
- Execute actions on NMS (Network Management Systems), for example TeMIP, through OSS Open Mediation V7.1 (NOM V7.1).

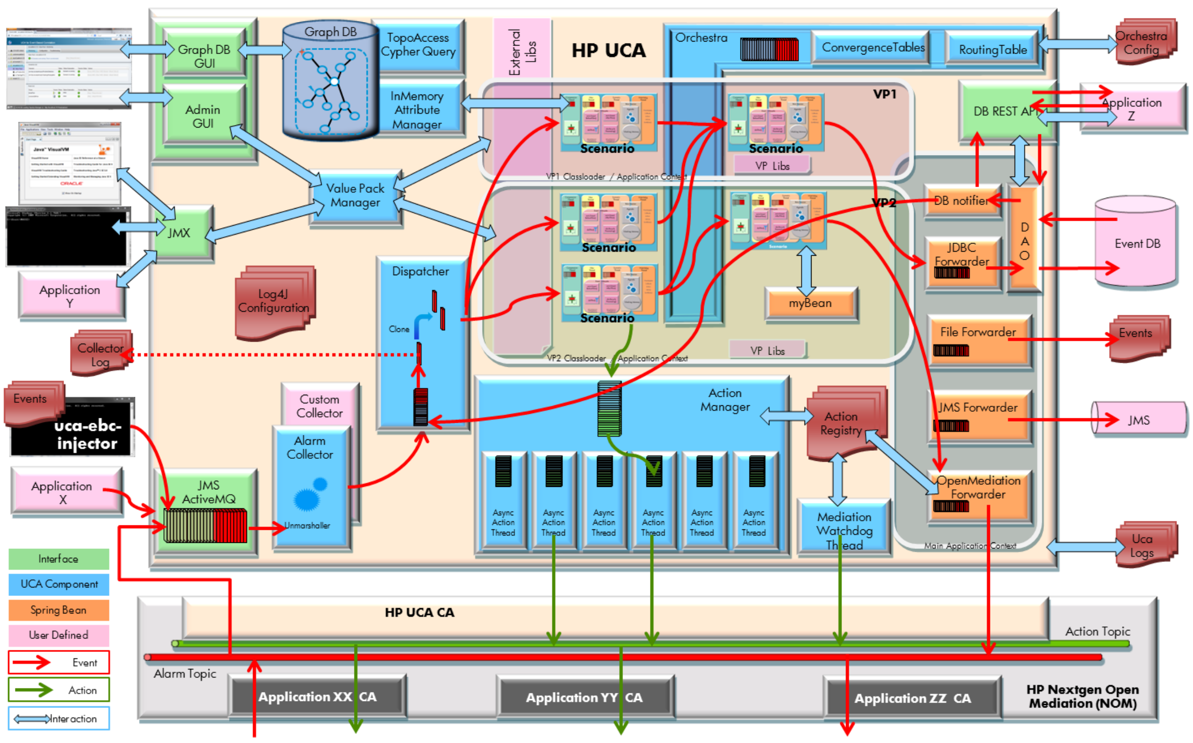


Figure 2 - Extended Solution architecture

2.2 Components

The UCA for EBC solution is made of the following main building blocks:

Alarm Collector:

This component is responsible for collecting incoming alarms. It collects XML alarms from a **JMS Queue** (this queue is implemented as a JMS Topic), in a format similar to the CITT X733 Standard. The *Alarm Collector* validates incoming alarms and forwards them to the *Dispatcher*.

Dispatcher:

This component is responsible for dispatching incoming alarms to Scenarios within Value Packs. The Dispatcher dispatches alarms only to the Scenarios configured as 'eligibleForBroadcast' (See Chapter 4.1.3.1 eligibleForBroadcast).

If the incoming alarms' "TargetValuePack" (1) property contains a reference to a valid Value Pack, and that this Value Pack is active (i.e. in a "Running" or "Degraded" state), it only dispatches alarms to the Scenarios of this specific Value Pack. Otherwise it broadcasts the alarms to eligible Scenarios of all active (started) Value Packs.

Scenario:

This key component is responsible for filtering incoming alarms and inserting "matching" alarms into the working memory of its *rule engine*. Each scenario runs its own rule engine and executes its own set of rules. This where alarm correlation really happens.

Action web service client:

Component responsible for forwarding action requests to Network Management Systems through OSS Open Mediation V7.1 (NOM V7.1) and retrieving the responses.

Note

(1) The format of the “TargetValuePack” property is the following: <*value pack name*>-<*value pack version*>##<*mediation flow name*>, for example: pd-example-2.1##temipFlow.

2.3 Value Pack

A UCA for EBC Value Pack offers a comprehensive set of correlation capabilities. It is identified with a name and a version.

A Value Pack may contain one or more scenarios.

A Value Pack may be packaged with its own set of external libraries (jar files) required by the Value Pack (usually the custom code called from rules).

A single XML configuration file (ValuePackConfiguration.xml) describes the Value Pack and its Scenarios.

2.4 Scenario

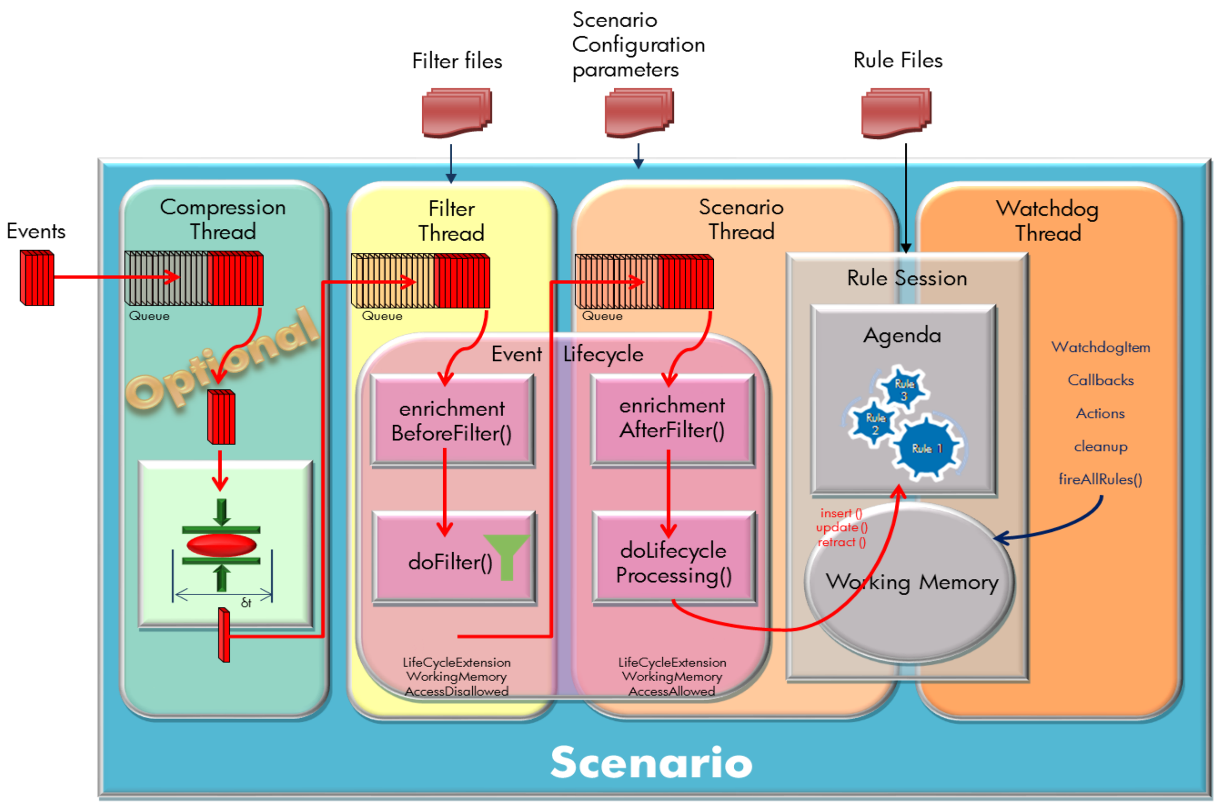


Figure 3 - Scenario

A Scenario is made of:

- Filters
- Inference engine (including the Working Memory)
- Rules

Each scenario is defined in the Value Pack XML configuration file. Detailed information on how to define a scenario is provided in section 3.5 “Scenario”.

A scenario is independent building block that can be developed and tested separately. A complex correlation solution can therefore be built with from simpler independent pieces. This eases the design, development, testing and maintenance of the overall correlation solution.

2.4.1 Filters

The filter is the precondition for an alarm to be eligible to a scenario.

The filter can be complex, with nested sub-filters, using any logical operator combinations, as well as, strings operators, integers, or date filter statements.

Filters are defined in XML configuration files, and described in section 3.5.3 “Filter definition file”.

Each scenario has an internal queue holding alarms to be filtered. The dispatcher pushes alarms to this queue. The filter discards alarms not meaningful for the scenario.

2.4.2 Inference Engine

The Drools Inference Engine matches facts (i.e. alarms other objects) against rules (i.e. knowledge base) to decide what actions to take.

UCA for EBC inference engine is based on *JBOSS Drools 5.5.0.Final*

JBOSS Drools 5.5.0 implements and extends the *Rete* algorithm which is an efficient pattern matching algorithm for implementing production rule systems. It also provides an enhanced and optimized implementation of the *Rete* algorithm for object oriented named *ReteOO*.

☞ For additional information on Drools Rule engine, you can refer to [R5] *JBoss Drools Expert guide , Chapter 1.1 - What is a Rule Engine?*

Depending on the targeted use case, the UCA for EBC rule engine can be configured in two different modes.

- **CLOUD** mode
- **STREAM** mode

With the CLOUD mode, all facts in the working memory are taken into account for rules evaluation. There is no notion of ordering (flow) or time constraints.

With the STREAM mode, the alarm ordering and the time dimension are keys. When using the STREAM mode, the engine is able to support sliding windows, and the concept of “now”.

☞ For details on these modes, please refer to [R6] *JBoss Drools Fusion guide , Chapter 2.5. Event Processing Modes* and Chapter 2.6 Sliding windows.

2.4.3 Rules

A Rule is a two-part structure made of:

- Some <conditions> (called Left Hand Side - LHS)
- Some <actions> (called Right Hand Side - RHS).

```
when
    <conditions>
then
    <actions>;
```

A scenario is usually implemented using several rules that can be defined in one or more rules files.

Rules are evaluated against the contents of the Working Memory. When the conditions of a rule are true the rule is executed.

☞ More information on rules can be found in [R5] *JBoss Drools Expert guide* , Chapter 4.1.2. “*What makes a rule?*” and in the [R7] *Unified Correlation Analyzer for Event Based Correlation – Value Pack Development Guide*, Chapter “*Developing the scenario rules*”.

☞ All documentation about *JBOSS Drools 5.5.0* can be found at:
http://www.jboss.org/drools/documentation.html

Chapter 3

Value Packs and Scenarios

3.1 Definitions

An UCA for EBC Value Pack is made of a set of scenarios packaged together. These scenarios are usually assembled in order to implement a specific business case in terms of correlation.

A Value Pack is identified by a name and a version. It is delivered as a zip file.

Each Value Pack can be deployed / started / stopped / un-deployed independently of other Value Packs.

Each Value Pack is also independent from a “Java class loading” point of view. Each Value Pack has its own Java class loader, thus Java class loading conflicts between scenarios are avoided.

Each scenario is composed of a set of rules that run in the context of the scenario’s own rule engine and working memory.

3.2 Value packs core features

All UCA for EBC Value Packs are built on top of a set of basic functionalities provided by the UCA for EBC framework.

These functionalities are:

- Mediation Flow integration
- Alarm collection and validation

3.2.1 Mediation Flow integration

The UCA for EBC mediation (NOM V7.1 and Channel Adapters) is tightly integrated with UCA for EBC at the Value Pack level.

If the Channel Adapter you’re targeting supports the “dynamic flows” feature, the Value Pack itself can create and delete specific Alarm Flows directly on the mediation layer.

The creation of dynamic flows is automatically requested at Value Pack start-up (unless the automaticStart property of the mediation flow is set to false in the `ValuePackConfiguration.xml` file.

☞ See Chapter 3.4.2.2 “Defining Collection flows” for more information). Re-creation of the mediation flow is also automatically requested after the value pack has started if the value pack detects that this flow does not exist anymore.

Each mediation flow can be started, stopped or resynchronized separately from the others, either from the GUI, the Java JMX console or the rules of the value pack.

The status of each mediation flow is displayed at the GUI. At the GUI, it is possible to view, edit, apply and save the configuration of the mediation flows for each value pack.

The status of each mediation flow is also displayed at the Java JMX console along with troubleshooting information.

☞ See [R10] *Unified Correlation Analyzer for Event Based Correlation – User Interface Guide*

☞ See Chapter 3.4.2.2 Defining Collection flows for additional information.

3.2.2 Alarm collection and validation

A UCA for EBC Value Pack is able to receive any of the following types of alarms/events:

- Alarm creation (Alarm): a new alarm.
- Alarm state change (AlarmStateChange): an alarm indicating a status change (networkState, operatorState, problemState).
- Alarm attribute value change (AlarmAttributeValueChange): an alarm indicating an attribute value change (except state attributes).

Alarm deletion (AlarmDeletion): a deleted alarm.

The full definition of these Alarm types is described in the Chapter 0 "

All flavors of Alarms objects inherit from Event class.

Alarm" of this document.

The UCA for EBC core system can guarantee that the alarms forwarded to the Value Packs have a predefined set of mandatory fields defined.

The table below indicates the list of mandatory alarm fields (the presence of all mandatory fields is enforced by the UCA framework):

Alarm type	Mandatory fields	Comments
Alarm	- Identifier - Source identifier - originatingManagedEntity - probableCause - alarmType - perceivedSeverity	networkState, operatorState, problemState are 'noted' as mandatory in the schema but functionally, they are optional
AlarmStateChange	-Identifier	
AlarmAttributeValueChange	-Identifier	
AlarmDeletion	-Identifier	

Table 1 - Alarm Collection validation

In addition to validating the presence of mandatory fields, the framework does the following processing during alarm collection:

If an alarm is missing a mandatory attribute, the alarm is ignored and an error message is added to the `${UCA_EBC_INSTANCE}/logs/uca-ebc.log` file.

If the alarmRaisedTime attribute is not provided, this attribute is set with the current time.

If either one of the networkState, operatorState and problemState attributes is missing, the following default values are used: NOT_CLEARED, NOT_ACKNOWLEDGED and NOT_CLOSED.

The UCA framework validates that each incoming alarm message conforms to the UCA for EBC Alarm schema during collection. As this validation is CPU consuming, it can be de-activated by setting the following property in the UCA for EBC properties file (located at: `${UCA_EBC_INSTANCE}/conf/uca-ebc.properties`):

collector.messages.validation=false

On the other hand, to activate alarm validation by the alarm collector, you need to set the following property in the UCA for EBC properties file:

collector.messages.validation=true

☞ See UCA for EBC Administration chapter in [R2] *Unified Correlation Analyzer for Event Based Correlation – Administration, Configuration and Troubleshooting Guide*

3.3 Scenario core features

All UCA for EBC Scenarios are built on top of a set of basic functionalities provided by the UCA for EBC framework.

These functionalities are:

- Alarm compression
- Alarm or Event filtering
- Alarm or Event enrichment
- Alarm or Event life cycle
- Automatic rules firing

3.3.1 Alarm compression

UCA for EBC offers a way to the Scenario developer to implement alarm compression. This compression takes place when an Alarm or Alarm-related event has been dispatched to a Scenario by the Dispatcher component.

Alarm compression is a scenario policy that can be set and configured for each scenario defined in the `ValuePackConfiguration.xml` file of a Value Pack. By default alarm compression is not enabled. If enabled, Alarm Attribute Value Change and Alarm State Change events will be compressed (grouped together) over a period of time (called compressionPeriod) before being sent to the Scenario.

The goal of Alarm compression is to improve the efficiency and performance of a Scenario by reducing the flow of alarm Attribute Value Change and alarm State Change events.

☞ For more information regarding compression and how to enable/disable compression for a Scenario, please go to Chapter 4.1.1.5 “compressionMode”

3.3.2 Alarm or Event filtering

UCA for EBC offers a way to the Scenario developer to implement alarm or event filtering. This filtering takes place when an Alarm or Alarm-related event has been dispatched to a Scenario by the Dispatcher component, right after the compression phase (if enabled) described in chapter 3.3.1 “Alarm compression”.

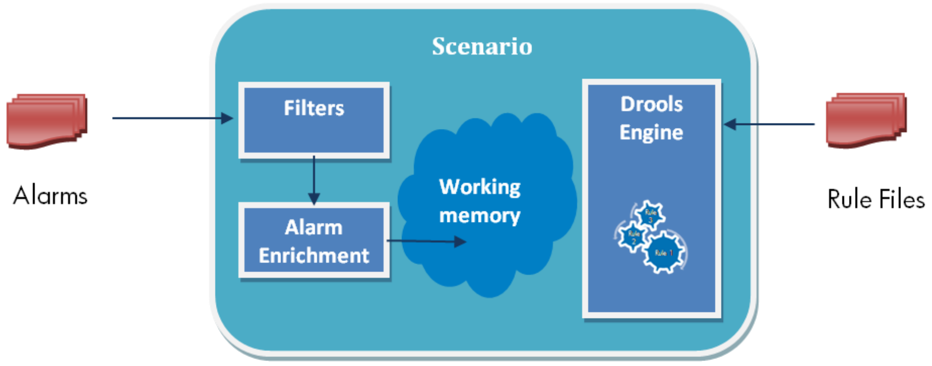
Alarm filtering is based on filter files. Each scenario is associated with a filter file in the Scenario specific section of the `ValuePackConfiguration.xml` file of a Value Pack.

The goal of Alarm filtering is to reduce the flow of Alarms and Alarm-related events being sent to a Scenario by defining a set of filters. In the filter file, you can define the type(s) of Alarms and Alarm-related events that should be forwarded to the Scenario and the ones that should be filtered out using pattern patching on the fields of the Alarms and Alarm-related events.

☞ For more information regarding filtering and how to configure filtering for a Scenario, please go to Chapter 3.5.3 “Filter definition file”

3.3.3 Alarm or Event enrichment

UCA for EBC offers a way to the Scenario developer to implement alarm enrichment. This enrichment takes place after the Alarm has been collected and before it is injected in the Scenario Working memory.



The above picture shows incoming Alarms but it applies also to incoming Events.

This feature is typically used when some external additional information is necessary to accomplish the alarm correlation. Such additional information can be some topology information coming from the Topology database or from any other source of information. This information is used to enrich the processed alarm before applying the correlation rules.

Another typical use of alarm enrichment is when you need to process your own objects (other than the default Alarm and Alarm-related objects) in the rules.

☞ For more detailed information on how to implement Alarm Enrichment, See:

Chapter 3.5.6 “Alarm Enrichment” of this document

“Implementing alarm enrichment” chapter [R7] *Unified Correlation Analyzer for Event Based Correlation – Value Pack Development Guide*

[R11] *Unified Correlation Analyzer for Event Based Correlation – Topology Extension*

3.3.4 Alarm lifecycle

UCA for EBC scenarios can be either in STREAM or CLOUD processing mode. Choosing one or the other of these processing modes has an impact on the way the Alarm lifecycle is managed by the UCA for EBC framework within the scenario.

3.3.4.1 CLOUD Mode

The CLOUD processing mode is the “usual” processing mode of inference engines. Users of rules engine are familiar with this mode because it behaves exactly the same way as any pure forward chaining rules engine, including previous versions of Drools or TeMIP Expert for instance.

When running in CLOUD mode, the engine sees all facts in the working memory. There is no notion of flow of time, although events have a timestamp as usual. In other words, although the engine knows that a given event was created, for instance, on January 1st 2012, at 09:35:40.767, it is not possible for the engine to determine how "old" the event is, because there is no concept of "now".

In this mode, the Drools engine will apply its usual many-to-many pattern matching algorithm, using the rules constraints to find the matching tuples, activate and fire rules as usual.

This mode does not impose any kind of additional requirements on facts. So for instance:

There is no notion of time. No clock synchronization requirements.

There is no requirement on event ordering. The engine looks at the events as an unordered cloud against which the engine tries to match rules.

In CLOUD mode, it is not possible to use sliding windows, because sliding windows are based on the concept of "now" and there is no such concept in CLOUD mode.

With the CLOUD mode, the UCA for EBC framework fully controls the lifecycle of the Alarms objects stored in Working Memory by applying the following algorithm:

On ‘Alarm’ message reception, an Alarm Object is created (if not already present) in each scenario for which this alarm matches the alarm filter. This Alarm Object will be constructed with the information from the ‘Alarm message’. Each new alarm object is inserted into the scenario’s Working Memory if this alarm is eligible (Refer to 4.1.3.2 alarmEligibilityPolicy).

Note: Any new alarm inserted to the Working memory has the specific attribute ‘justInserted’ set to ‘true’. This alarm attribute can be used in rule condition in order for the rule to be triggered on each new alarm insertion. It is then the responsibility of the rule developer to clear the ‘justInserted’ flag by calling the alarm method `setJustInserted(false)` in the rule ‘then’ section.

On any ‘Alarm state change’ message reception (AlarmStateChange), the associated alarm Object is retrieved from the scenarios’ Working Memory where it is present. The alarm object state attributes (networkState, problemState, operatorState) are updated given the new values from the AlarmStateChange message. In the case when the alarm becomes ‘Not Eligible’ anymore (Refer to 4.1.3.2 alarmEligibilityPolicy) the alarm is retracted from Working Memory after the alarm attribute ‘aboutToBeRetracted’ to ‘false’ (in the same manner as for AlarmDeletion messages). Then the rules are fired again.

Note: Any alarm that has its state updated in Working Memory has the specific attribute ‘hasStateChanged’ set to ‘true’. This alarm attribute can be used in rule condition in order for the rule to be triggered on alarm state changes. It is then the responsibility to the rule developer the clear the 'hasStateChanged' flag by calling the alarm method `setHasStateChanged(false)` in the rule 'then' section.

On reception of an 'Alarm attribute change' message (AlarmAttributeChange) such as perceivedSeverity… the corresponding attributes of the WM alarm Object is updated (for each scenario where the alarm object is present), and then the rules evaluation is triggered again.

Note: Any alarm that has attributes updated in Working Memory has the specific attribute 'hasAVCChanged' set to 'true' (AVC stands for Attribute Value Change). This alarm attribute can be used in rule condition in order for the rule to be triggered on alarm attribute changes. It is then the responsibility to the rule developer the clear the 'hasAVCChanged' flag by calling the alarm method `setHasAVCChanged(false)` in the rule 'then' section.

On 'AlarmDeletion' reception, the associated alarm objects are retrieved from the scenarios' working memory. The alarm retraction is done in two steps:

Update the alarm attribute 'aboutToBeRetracted' to 'true' and then fire the rules again.

Finally a garbage collection mechanism will automatically retract all alarms with the attribute 'aboutToBeRetracted' set to 'true'. If you wanted to cancel the removal of the alarm from Working Memory, you could clear the 'aboutToBeRetracted' flag by setting it to 'false' using the setAboutToBeRetracted(boolean) method of the Alarm object.

The Alarm Object will remain in the scenario's working memory as long as no AlarmDeletion message is received, as long as the alarm remains valid according to the alarmEligibilityPolicy (See Chapter 4.1.3.2 alarmEligibilityPolicy), or as long as it is not explicitly retracted from Working Memory (using a "retract" statement in the rules).

Note: the four alarm attributes ('justInserted', 'hasStateChanged', 'hasAVCChanged', and 'aboutToBeRetracted') give the developper the possibility to implement specific processing at alarm Working Memory insertion, attribute change, state change or retraction.

The following is an example of rules using these alarm attributes in the rules condition (fireAllRulesPolicy set to EACH_ACCESS, See Chapter 4.1.1.2 fireAllRulesPolicy):

```
package hp.uca.expert.engine
#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;

rule "Processing at alarm creation"
when
        a : Alarm (justInserted == true)
then
        System.out.println ("========Alarm just inserted in Working
Memory : " + a.getIdentifier());

        // place your alarm processing code here

        // Optionally reset the JustInserted flag
        a.setJustInserted(false);
End

rule "Processing at alarm attribute change"
when
        a : Alarm (hasAVCChanged == true)
then
```

```
        System.out.println ("========Alarm attributes just updated in
Working Memory : " + a.getIdentifier());

        // place your alarm processing code here

        // Optionally reset the HasAVCChanged flag
        a.setHasAVCChanged(false);
End

rule "Processing at alarm state change"
when
        a : Alarm (hasStateChanged == true)
then
        System.out.println ("========Alarm state just updated in
Working Memory : " + a.getIdentifier());

        // place your alarm processing code here

        // Optionally reset the HasStateChanged flag
        a.setHasStateChanged(false);
End
rule "Processing at alarm deletion"
when
        a : Alarm (aboutToBeRetracted == true)
then
        System.out.println ("========Alarm about to be retracted from
Working Memory : " + a.getIdentifier());

        // place your alarm processing code here

        // Optionally reset the AboutToBeRetracted flag to cancel
        // alarm removal from Working Memory
        a.setAboutToBeRetracted (false);
end
```

Rule Sample 1 - Using 'justInserted', 'hasAVCChanged', 'hasStateChanged' and 'aboutToBeRetracted' alarm flags

3.3.4.2 STREAM (Complex Event Processing) Mode

When using the STREAM, the Drools engine knows the concept of time flow and the concept of "now", i.e., the engine understands how old events are based on the current timestamp read from the Session Clock. This characteristic extends the temporal reasoning possibilities of the engine with features like:

- Sliding Window support (ex: window:time() or window:length())
- Automatic Event Retraction (expiration time)
- Automatic Rule Delaying when using Negative Patterns (ex: not(Alarm(this after[0s,10s] $f))

With the STREAM mode any type of alarm message (Alarm, AlarmStateChange, AlarmAttributeChange, AlarmDeletion) has its corresponding object inserted in the working memory by the UCA for EBC framework.

The Framework does not manage any Alarm lifecycle in STREAM mode. However it automatically retracts each created Working Memory object after a predefined (configurable) period of time.

In STREAM mode, it is recommended to set the fireAllRulesPolicy to 'EACH_ACCESS' (See Chapter 4.1.1.2 fireAllRulesPolicy).

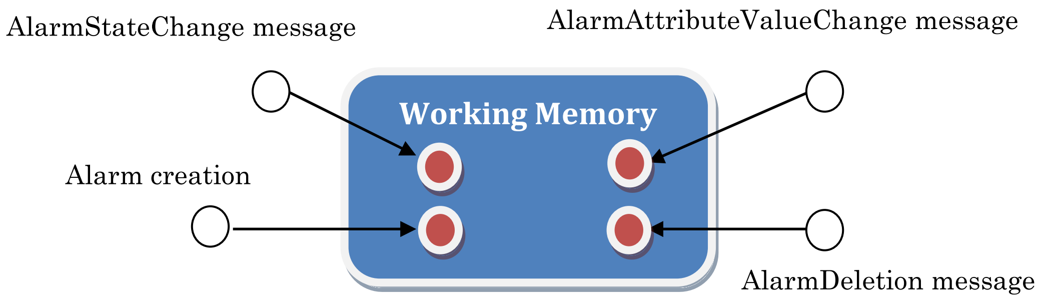


Figure 4 - Working Memory in STREAM mode

The STREAM mode is specifically useful for writing rules where the order of incoming alarms messages is key or if the time at which the messages arrive is important.

The following is an example of rule implementation in STREAM mode:

```
package hp.uca.expert.engine
#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.alarm.AlarmStateChange;
import com.hp.uca.expert.alarm.AlarmAttributeValueChange;
import com.hp.uca.expert.alarm.AlarmDeletion;

rule "Trace for alarm creation"
when
        a : Alarm ()
then
        System.out.println ("======== New Alarm ====================" +
a.getIdentifier());
end

rule "Trace for alarm state change"
when
        a : AlarmStateChange ()
then
        System.out.println ("======== New Alarm State Change
====================" + a.getIdentifier());
end

rule "Trace for alarm AVC change"
when
        a : AlarmAttributeValueChange ()
then
        System.out.println ("======== New Alarm AVC
====================" + a.getIdentifier());
end

rule "Trace for alarm deletion"
when
        a : AlarmDeletion ()
```

```
 then
        System.out.println ("======== New Alarm deletion
====================" + a.getIdentifier());
end
```

Rule Sample 2 - STREAM mode and Alarms

3.3.4.3 Differences between CLOUD and STREAM modes

The following table summarizes the differences between CLOUD and STREAM Scenario modes:

Feature	Cloud	Stream (CEP)
Rule syntax	Standard	Standard + Time based
Object lifetime in WM	No expiration time, Depends on Eligibility Policy criteria, Until explicitely retracted (using the retract statement)	Automatic (or manual) Event retraction at expiration time
Object types in WM	Alarm + *Any other object directly inserted into Working Memory*	Alarm *(i.e. Alarm Creation)* AlarmStateChange AlarmAttributeValueChange AlarmDeletion + *Any other object directly inserted into Working Memory*
Lifecycle Mgt	Alarm is transparently updated based on State Change, Attribute Value Change, Deletion messages received from the mediation layer	N/A
Filtering	Via XML file configuration	Via XML file configuration
Orchestration of Scenarios Cascading	Via Scenario object in rules	Via Scenario object in rules
Completed Action retraction	Via Action retraction policy	Via Action retraction policy
FireAllRule recommended policy	WATCHDOG: to avoid unnecessary (intermediate) rules activation (and save processing time in case of transient rule activation) EACH_ACCESS: all rules activation	EACH_ACCESS

Table 2 - Summary CLOUD versus STREAM mode

3.3.5 Automatic rules firing

The rules loaded in UCA for EBC are automatically fired by the Scenario component.

The Drools Agenda is updated with the list of rules that have to be fired (Rule Activation) during the next “fireAllRules” request when an alarm is:

- inserted in the Scenario’s Working Memory (either an alarm coming from the Mediation layer, or an alarm directly inserted in Working Memory by a rule)
- updated in the Scenario’s Working Memory (when the value of an attribute of an Alarm or any Object has changed)
- retracted from the Scenario’s Working memory

Depending on a specific Scenario policy (the fireAllRules policy), the ‘fireAllRules’ is automatically requested:

- Either on regular basis
- Or after each Working Memory access

☞ See Chapter 4.1.1.2 fireAllRulesPolicy, for more information about the automatic rule firing mechanism.

3.4 Value Pack Definition

3.4.1 Files and distribution

The directory named *`${UCA_EBC_INSTANCE}/deploy`* is the location where UCA for EBC deploys Value Packs. Once deployed, each Value Pack has its own root directory within this "`deploy`" directory.

Each Value Pack has the following folder structure:

Content of ${UCA_EBC_INSTANCE}/deploy/<Value Pack name>-<Value Pack version>/:

`conf/`	Configuration directory of the Value Pack. Contains *`ValuePackConfiguration.xml`* file that is main configuration file of any Value Pack.	
	`ValuePackConfiguration.xml`	This file is used to define the configuration of the Value Pack, its Scenarios, and Mediation Flows.
	`context.xml` (optional)	This file can be used to define Spring beans that can be used in the "global" part of the definition of Scenarios in the *`ValuePackConfiguration.xml`* file. These global variables (associated with Spring beans) can then be used in rules files. See [R7] *Unified Correlation Analyzer for Event Based Correlation – Value Pack Development Guide*, Chapter Spring Framework Integration.
`lib/`	Java Libraries necessary for the Value Pack	
`<scenario name>/`	Each scenario has a directory named after it. This directory has a specific structure (see next paragraph about Scenario)	

Table 3 - Value Pack distribution files

Defining a Value pack is a task that involves:

- specifying the basic attributes of the Value Pack, such as its name and version
- listing the different Scenarios used by this Value Pack (and their configuration parameters)
- optionally, listing the different Mediation Flows used by this Value Pack (and their configuration parameters)

3.4.2 Value pack definition file

The Value Pack is defined in the ***`ValuePackConfiguration.xml`*** file:

This file is located in the ${UCA_EBC_INSTANCE}/deploy/<Value Pack Name>/conf/ directory.

The following table lists the different properties that define a Value Pack:

Type	Name	Value
Attribute	name	Property ‘name’ that identifies the name of the Value Pack.
Attribute	version	Property ‘version’ that indicates the version of the Value Pack.
Property	scenarios	List of the Scenarios defined by the Value Pack
Property	mediationFlows	**Optional**. List of the Mediation Flows defined by the Value Pack
Property	dbFlows	**Optional**. List of the DB Flows defined by the Value Pack

Table 4 - Value Pack properties

3.4.2.1 Value Pack definition example

Following is a sample `ValuePackConfiguration.xml` file where the Value Pack, its Scenarios and Mediation Flows are defined:

```
<?xml version="1.0" encoding="UTF-8"?>


<valuePackConfiguration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        name="skeleton_project"
         version="1.0">


        <scenarios>
             <scenario name="skeleton">
                 <filterFile>src/main/resources/valuepack/skeleton/filters-file.xml</filterFile>
                 <fireAllRulesPolicy>EACH_ACCESS</fireAllRulesPolicy>
                 <globals></globals>
                 <processingMode>STREAM</processingMode>
                 <rulesFiles>
                         <rulesFile>
                     <filename>file:./src/main/resources/valuepack/skeleton/skeleton.drl</filename>
                             <name>rulesFile1</name>
                             <ruleFileType>DRL</ruleFileType>
                         <rulesFile>
                 </rulesFiles>
             </scenario>
        </scenarios>


        <mediationFlows>
             <mediationFlow name="temipFlow"
                         actionReference="TeMIP_FlowManagement"
                         flowNameKey="flowName">
                 <!-- Comment out the flowCreation and flowDeletion sections to use static flows
                        instead of dynamic flows -->
                 <flowCreation>
                         <actionParameter>
                                 <key>operation</key>
                                 <value>CreateFlow</value>
                         </actionParameter>
                         <actionParameter>
                                 <key>flowType</key>
                                 <!-- flowType can only be dynamic in the case of flowDeletion -->
                                 <value>dynamic</value>
                         </actionParameter>
                         <actionParameter>
                                 <key>operationContext</key>
                                 <!-- a valid TeMIP Operation Context name -->
                                 <value>oc_xxx</value>
                         </actionParameter>
                 </flowCreation>
                 <flowDeletion>
```

```
                        <actionParameter>
                                <key>operation</key>
                                <value>DeleteFlow</value>
                        </actionParameter>
                        <actionParameter>
                                <key>flowType</key>
                                 <!-- flowType can only be dynamic in the case of flowDeletion -->
                                <value>dynamic</value>
                        </actionParameter>
                </flowDeletion>
                <flowResynchronization>
                        <actionParameter>
                                <key>operation</key>
                                <value>ResynchFlow</value>
                        </actionParameter>
                        <actionParameter>
                                <key>flowType</key>
                                <!-- flowType can be either static or dynamic -->
                                <value>dynamic</value>
                        </actionParameter>
                </flowResynchronization>
                <flowStatus>
                        <actionParameter>
                                <key>operation</key>
                                <value>StatusFlow</value>
                        </actionParameter>
                        <actionParameter>
                                <key>flowType</key>
                                <!-- flowType can be either static or dynamic -->
                                <value>dynamic</value>
                        </actionParameter>
                </flowStatus>
             </mediationFlow>
        </mediationFlows>
</valuePackConfiguration>
```

XML Configuration 1 - ValuePackConfiguration.xml example

3.4.2.2 Defining Collection flows

The <**mediationFlows**> tags of the Value Pack Configuration file indicates the list of Mediation Flows to use with the Value Pack.

Inside the <**mediationFlows**> tag, you can define as many Mediation Flows as you want, each Mediation Flow being defined inside a <**mediationFlow**> tag.

Each <**mediationFlow**> tag defines a list of Action parameters used to set-up a Mediation Flow.

For Channel Adapters that support 'dynamic' Alarm Flows, the configuration of the flow (creation, deletion, resynchronization or status request) is done through a standard "configuration" Action, targeted to the Channel Adapter itself.

As described in *Chapter 5.3 Actions,* the Action is a set of key/value pairs sent to an OSS OpenMediation Channel Adapter (the routing information is uniquely defined via the 'actionReference' parameter, see *Chapter 5.3.2 Action registry*).

Each <**mediationFlow**> tag is configured using the following **sequence** of XML tags (the order is important).

Type	Name	Value
Attribute	name	**Mandatory**. Attribute 'name' that identifies the name of the Mediation Flow. This attribute will be used when automatically constructing the value of the 'flowName' key/value pair. For instance, if the flowNameKey attribute is set to 'flowName', the Value Pack name-version is 'myVP-

Type	Name	Value

		0.1’, and the Flow name is ‘myFlow’, then the following key/value pair is automatically added to all CreateFlow/DeleteFlow/ResynchFlow/StatusFlow directives:
		
		`<actionParameter>`
		`        <key>flowName</key>`
		`        <value>myVP-0.1##myFlow</value>`
		`</actionParameter>`

Attribute	automaticStart	**Optional**. The automaticStart attribute of type Boolean indicates whether to automatically start the Mediation Flow when the Value Pack starts (this is the default value if the automaticStart attribute is omitted) or not. If the Mediation Flow is not set to automatically start when the Value Pack start (or even if it is), it can be started (also stopped or resynchronized) from the Rules of any Scenarios of the Value Pack. ☞ Please see Chapter 5.11 Collection Flows or refer to [R3] *Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Engine* for more information on how to start/stop/resynchronize a Mediation Flow from the Rules of any Scenario.

Attribute	flowNameKey	**Mandatory**.
		
		The flowNameKey attribute is a parameter that is automatically added in the arguments of the Action generated using this set of parameters. For all phase (creation, deletion, resynchronization, status), an Action key/value parameter is automatically added with the ValuePack identifier.
		
		For instance, if the flowNameKey attribute is set to ‘flowName’, the Value Pack name-version is ‘myVP-0.1’, and the Flow name is ‘myFlow’, then the following parameter is automatically added in the Action:
		`<actionParameter>`
		`        <key>flowName</key>`
		`        <value>myVP-0.1##myFlow</value>`
		`</actionParameter>`
		In addition of the one defined in the selected phase:
		`<actionParameter>`
		`        <key>operation</key>`
		`        <value>StatusFlow</value>`
		`</actionParameter>`
		`<actionParameter>`
		`        <key>flowType</key>`
		`        <value>dynamic</value>`
		`</actionParameter>`

Attribute	actionReference	**Mandatory**. The actionReference defined in the ActionRegistry (see Chapter 5.3.2 Action registry)

Type	Name	Value
Attribute	lastEventReceivedFirstDuringResynchronization	**Mandatory**. This attribute is of type Boolean. When set to true, it means that the Flow is sending the Events in INVERTED order (most recent Event first). Typical value is true for a TeMIP flow.
Property	flowCreation	**Recommended** if the targetted Channel Adapter supports 'dynamic' flows, but optional. List of key/value pairs that will define the Action sent to the Channel Adapter to create a Value Pack specific Alarm Flow when the Value Pack starts.
Property	flowDeletion	**Recommended** if the targetted Channel Adapter supports 'dynamic' flows, but optional. List of key/value pairs that will define the Action sent to the Channel Adapter to delete a Value Pack specific Alarm Flow when the Value Pack stops.
Property	flowResynchronization	**Recommended** but optional. List of key/value pairs that will define the Action sent to the targeted Channel Adapter to resynchronize a Value Pack specific Alarm Flow when a resynchronization of the Value Pack is requested. This parameter is valid, whether the Channel Adapter supports 'dynamic' or 'static' flows. When this tag is not present, the button "Resynchronize" will not be present in the UCA Administration GUI for this Value Pack (See [R10] Unified Correlation Analyzer for Event Based Correlation – User Interface Guide)
Property	flowStatus	**Mandatory**. List of key/value pairs that will define the Action sent to the Channel Adapter to check the status of the Channel Adapter and its flow. This parameter is valid, whether the Channel Adapter supports 'dynamic' or 'static' flows. The UCA framework will check the status of the flow every 30s and the Value Pack status reflects the Mediation Flow status (turns to DEGRADED when the mediation is no more available for instance).

Table 5 - Mediation Flows properties

In the above table, in the "Type" column, the term "Attribute" refers to an XML attribute of the <**mediationFlow**> tag, while the term "Property" refers to a separate XML tag inside the <**mediationFlow**> tag.

The <**dbFlows**> tags of the Value Pack Configuration file indicates the list of DB Flows to use with the Value Pack.

Inside the <**dbFlows**> tag, you can define as many DB Flows as you want, each DB Flow being defined inside a <**dbFlow**> tag.

Each <**dbFlow**> tag defines the parameters needed to set-up a DB Flow.

Each <**dbFlow**> tag is configured using only following attributes.

Type	Name	Value
Attribute	name	**Mandatory**. Identifies the name of the DB Flow.
Attribute	dbNotifierName	**Mandatory**. Identifies the name of the DB Notifier bean instantiated to handle this DB flow
Attribute	lastEventReceivedFirstDuringResynchronization	**Mandatory**. This attribute is of type Boolean. When set to true, it means that the Flow is sending the Events in INVERTED order (most recent Event first). Typical value is false for a standard DB flow.
Attribute	automaticStart	**Optional**. The automaticStart attribute of type Boolean indicates whether to automatically start the DB Flow when the Value Pack starts (this is the default value if the automaticStart attribute is omitted) or not. If the DB Flow is not set to automatically start when the Value Pack start (or even if it is), it can be started (also stopped or resynchronized) from the Rules of any Scenarios of the Value Pack. ☞ Please see Chapter 5.11 Collection Flows or refer to [R3] *Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Engine* for more information on how to start/stop/resynchronize a DB Flow from the Rules of any Scenario.
Attribute	sourceIdentifier	**Optional**. When an alarm is collected though DB flow, the sourceIdentifier is replaced by this value.
Property	eligibilityScope	**Recommended.** It is a Java evaluated boolean expression defining the eligibility of an alarm to pass through the DB flow at synchronization time. If not specified, all alarms are eligible through that DB flow.

3.5 Scenario

3.5.1 Files and distribution

Each scenario directory should contain:

Name	Format	Description	Comments
**-filter.xml*	XML	Contains the definition of the filter for this scenario	The name of the filter file must match the name of the filter file referenced in the scenario definition part in the `ValuePackConfiguration.xml` file
<rule file 1>.drl *<rule file 2>.drl* …	DRL	Contains a set of rules for the scenario	The names of the rules files must match the names of the rules files referenced in the scenario definition part in the `ValuePackConfiguration.xml` file. Apart from this constraint, rule file names can be anything, provided they have a .drl extension.

Table 6 - Scenario files distribution

3.5.2 Scenario definition file

Each scenario is defined in the ValuePackConfiguration.xml file:

${UCA_EBC_INSTANCE}/deploy/<Value Pack Name>/conf

The <**scenarios**> tags of the Value Pack Configuration file indicates the list of Scenarios defined for the Value Pack.

Inside the <**scenarios**> tag, you can define as many Scenarios as you want, each Scenario being defined inside a <**scenario**> tag.

Each <**scenario**> tag defines the parameters and policies of the Scenario.

Each Scenarios is configured using the following **sequence** of XML tags (the order is important):

Type	Name	Value
Attribute	name	**Mandatory**. Property 'name' that identifies the name of the Scenario.
Property	automaticRefreshOnConfigurationChange	See Chapter 4.1.5 Automatic handling of configuration files modifications
Property	actionRetractedAutomaticallyWhenCompleted	See Chapter 4.1.4.1 actionRetractedAutomaticallyWhenCompleted
Property	alarmEligibilityPolicy	See Chapter 4.1.3.2 alarmEligibilityPolicy
Property	asyncActionPeriod	See Chapter 4.1.2.3 asyncActionPeriod
Property	clockTypeMode	Not Used
Property	eligibleForBroadcast	See Chapter 4.1.3.1 eligibleForBroadcast

Type	Name	Value
Property	filterFiles	At least one filter file must be defined
Property	filterFile	See Chapter 3.5.3 Filter definition file. The filterFile property is for backward-compatibility only.
Property	filterTagsFile	See Chapter 3.5.3 Filter definition file
Property	mapperFile	See Chapter 3.5.5 Mappers definition file
Property	fireAllRulePeriod	See Chapter 4.1.2.1 fireAllRulePeriod
Property	fireAllRulesDuringResynchronization	See Chapter 4.1.1.3 fireAllRulesDuringResynchronization
Property	fireAllRulesPolicy	**Mandatory**. See Chapter 4.1.1.2 fireAllRulesPolicy
Property	garbageCollectionPeriod	See Chapter 4.1.2.4 garbageCollectionPeriod
Property	globals	**Mandatory**. List of SpringFramework Beans reference that should be injected in the Scenario (Use these objects in rule). See [R7] *Unified Correlation Analyzer for Event Based Correlation – Value Pack Development Guide*
Property	global	**Mandatory only if used in the Scenario rule.** A SpringFramework Beans reference that should be injected in the Scenario (Use these objects in rule).
Property	processingMode	**Mandatory**. See Chapter 4.1.1.1 Processing Mode
Property	rulesFiles	**Mandatory**. See Chapter 3.5.8 Rules files
Property	tickPeriod	See Chapter 4.1.2.2 tickPeriod
Property	customLifeCycleClass	See Chapter 3.5.6 Alarm Enrichment
Property	customInitializationClass	See Chapter 3.5.7Scenario Initialization class
Property	compressionMode	See Chapter 4.1.1.5 compressionMode
Property	compressionPeriod	See Chapter 4.1.1.5 compressionMode
Property	retractOnResyncPolicy	See Chapter 4.1.1.4 retractOnResyncPolicy

Table 7 - Scenario properties

In the above table, in the "Type" column, the term "Attribute" refers to an XML attribute of the <**scenario**> tag, while the term "Property" refers to a separate XML tag inside the <**scenario**> tag.

3.5.2.1 Scenario definition example

The following is a sample ValuePackConfiguration.xml file (assuming this file is deployed in the `${UCA_EBC_INSTANCE}/deploy/myValuePack-1.0/conf/` directory):

```
<?xml version="1.0" encoding="UTF-8"?>
<valuePackConfiguration
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     name="myValuePack"
     version="1.0">

     <scenarios>
        <scenario name="com.acme.dummy.Scenario1">

<actionRetractedAutomaticallyWhenCompleted>true</actionRetractedAutomaticallyWhenCompleted>
          <asyncActionPeriod>1000</asyncActionPeriod>
          <clockTypeMode>NORMAL</clockTypeMode>
          <filterFile>deploy/myValuePack-1.0/scenario1/myFilters.xml</filterFile>
          <fireAllRulePeriod>1000</fireAllRulePeriod>
          <fireAllRulesDuringResynchronization>true</fireAllRulesDuringResynchronization>
          <fireAllRulesPolicy>WATCHDOG</fireAllRulesPolicy>
          <globals></globals>
          <processingMode>CLOUD</processingMode>
          <rulesFiles>
              <rulesFile>
          <filename>file:./deploy/myValuePack-1.0/scenario1/myRules.drl</filename>
                    <name>My Rules</name>
                    <ruleFileType>DRL</ruleFileType>
              </rulesFile>
          </rulesFiles>
          <tickPeriod>30000</tickPeriod>
       </scenario>
    </scenarios>
</valuePackConfiguration>
```

XML Configuration 2 - Scenario configuration example

3.5.3 Filter definition file

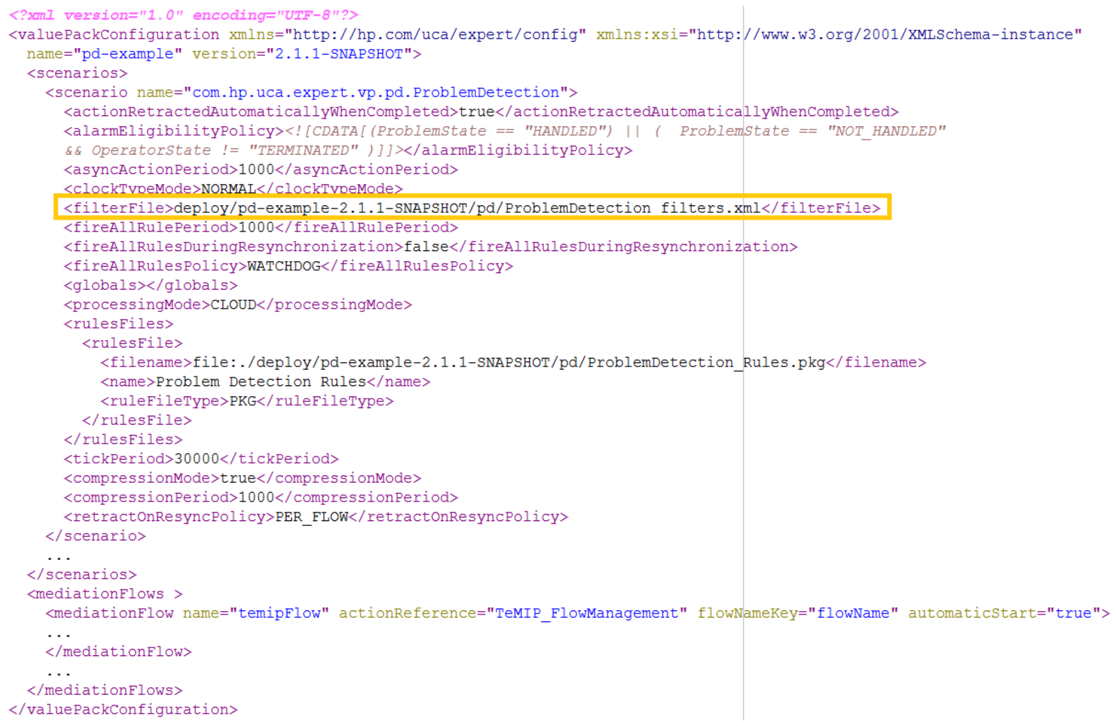
The filter definition file of a scenario is stored at the following location: *`${UCA_EBC_INSTANCE}/deploy/<myValuepack>-<version>/<myScenario>`*

Where:

- myValuepack is the Value pack name
- version is the Value pack version
- myScenario is the scenario name.

The name of the filter file can be anything, but the path for the filter file must be recorded in the configuration file for the Value Pack (the *`${UCA_EBC_INSTANCE}/deploy/<myValuepack>-<version>/conf/ValuePackConfiguration.xml`* file).

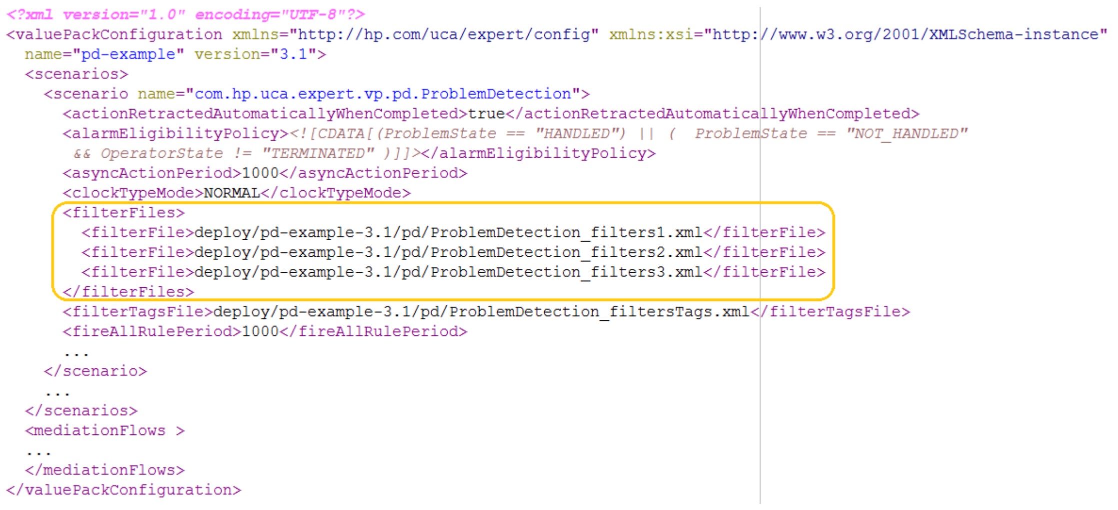
In the *`ValuePackConfiguration.xml file`*, the path for the filter file must be specified inside a `<filterFile>…</filterFile>` XML tag (inside a `<scenario>…</scenario>` XML tag), as shown in the screen capture below:



```
<?xml version="1.0" encoding="UTF-8"?>
<valuePackConfiguration xmlns="http://hp.com/uca/expert/config" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  name="pd-example" version="2.1.1-SNAPSHOT">
  <scenarios>
    <scenario name="com.hp.uca.expert.vp.pd.ProblemDetection">
      <actionRetractedAutomaticallyWhenCompleted>true</actionRetractedAutomaticallyWhenCompleted>
      <alarmEligibilityPolicy><![CDATA[(ProblemState == "HANDLED") || (  ProblemState == "NOT_HANDLED"
      && OperatorState != "TERMINATED" )]]></alarmEligibilityPolicy>
      <asyncActionPeriod>1000</asyncActionPeriod>
      <clockTypeMode>NORMAL</clockTypeMode>
      <filterFile>deploy/pd-example-2.1.1-SNAPSHOT/pd/ProblemDetection filters.xml</filterFile>
      <fireAllRulePeriod>1000</fireAllRulePeriod>
      <fireAllRulesDuringResynchronization>false</fireAllRulesDuringResynchronization>
      <fireAllRulesPolicy>WATCHDOG</fireAllRulesPolicy>
      <globals></globals>
      <processingMode>CLOUD</processingMode>
      <rulesFiles>
        <rulesFile>
          <filename>file:./deploy/pd-example-2.1.1-SNAPSHOT/pd/ProblemDetection_Rules.pkg</filename>
          <name>Problem Detection Rules</name>
          <ruleFileType>PKG</ruleFileType>
        </rulesFile>
      </rulesFiles>
      <tickPeriod>30000</tickPeriod>
      <compressionMode>true</compressionMode>
      <compressionPeriod>1000</compressionPeriod>
      <retractOnResyncPolicy>PER_FLOW</retractOnResyncPolicy>
    </scenario>
    ...
  </scenarios>
  <mediationFlows >
    <mediationFlow name="temipFlow" actionReference="TeMIP_FlowManagement" flowNameKey="flowName" automaticStart="true">
    ...
    </mediationFlow>
    ...
  </mediationFlows>
</valuePackConfiguration>
```

Figure 5 - Referencing a filter file inside a ValuePackConfiguration.xml file

<u>Note:</u> It is possible to define more than one filter file for a scenario. Instead of using the <filterFile>…</filterFile> tag to define the filter, you can use the <filterFiles>…</filterFiles> tag to define multiple filter files. All the filter files will be taken into account during the filtering process, as if there was just one filter file containing all the top filters defined in all the filter files. Below is an example of how to define multiple filter files:



```
<?xml version="1.0" encoding="UTF-8"?>
<valuePackConfiguration xmlns="http://hp.com/uca/expert/config" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  name="pd-example" version="3.1">
  <scenarios>
    <scenario name="com.hp.uca.expert.vp.pd.ProblemDetection">
      <actionRetractedAutomaticallyWhenCompleted>true</actionRetractedAutomaticallyWhenCompleted>
      <alarmEligibilityPolicy><![CDATA[(ProblemState == "HANDLED") || (  ProblemState == "NOT_HANDLED"
       && OperatorState != "TERMINATED" )]]></alarmEligibilityPolicy>
      <asyncActionPeriod>1000</asyncActionPeriod>
      <clockTypeMode>NORMAL</clockTypeMode>
      <filterFiles>
        <filterFile>deploy/pd-example-3.1/pd/ProblemDetection_filters1.xml</filterFile>
        <filterFile>deploy/pd-example-3.1/pd/ProblemDetection_filters2.xml</filterFile>
        <filterFile>deploy/pd-example-3.1/pd/ProblemDetection_filters3.xml</filterFile>
      </filterFiles>
      <filterTagsFile>deploy/pd-example-3.1/pd/ProblemDetection_filtersTags.xml</filterTagsFile>
      <fireAllRulePeriod>1000</fireAllRulePeriod>
      ...
    </scenario>
    ...
  </scenarios>
  <mediationFlows >
  ...
  </mediationFlows>
</valuePackConfiguration>
```

Figure 6 - Referencing multiple filter files inside a ValuePackConfiguration.xml file

A filter file defines a set of top filters. If the collected Alarm passes one of these top filters (there's an implicit OR operator between each top filter), the alarm will be sent to the scenario and if eligible (Please refer to section 4.1.3 "Alarm eligibility" for more information on alarm eligibility) the alarm will be inserted into the scenario's working memory.

Some value packs may need to know which filter allowed an alarm to be inserted into a scenario's working memory. The `getPassingFilters()` method on the Alarm object returns the list of such filters.

3.5.3.1 Filter definition file syntax

The complete syntax of the filter definition XML file is defined in the uca-expert-file.xsd XML Schema file located in the *${UCA_EBC_HOME}/schemas* folder.

Any filter definition file must contain one and only one <filters xmlns=*"http://hp.com/uca/expert/filter*">…</filters> XML element.

Inside the <filters xmlns=*"http://hp.com/uca/expert/filter*">…</filters> XML element, you can define one or more <topFilter>…</topFilter> XML elements, one for each top filter that you want to define. There's an implicit OR operator between each top filter. Incoming alarms need only pass one of the top filters.

3.5.3.1.1. The <topFilter> XML element

Each <topFilter>…</topFilter> XML element has a mandatory name attribute that defines the name of the top filter:

<topFilter name=*"mytopfilter"*>…</topFilter>

A <topFilter> XML element has an optional "tagsGroup" attribute that defines the group of tags that need to be proposed by the GUI tag editor utility. It should contain a valid group name or a list of comma-separated group names:

<topFilter name=*"mytopfilter"* tagsGroup="myTags1"> … </topFilter>

<topFilter name=*"mytopfilter"* tagsGroup="myTags1,myTags2"> … </topFilter>

Inside a <topFilter>…</topFilter> XML element, you can have a sequence of either of the following XML elements (there's an implicit AND operator on all the XML elements in the sequence):

<allCondition>…</allCondition>

<anyCondition>…</anyCondition>

<anyNotCondition>…</anyNotCondition>

<notCondition>…</notCondition>

Each of these XML elements can have an optional tag attribute (see section 3.5.4 "Filter tags" for more information on filter tags).

3.5.3.1.2. The <allCondition> XML element

<allCondition>…</allCondition> XML elements define a group of conditions that must all be true in order for the "allCondition" to be true.

Inside an <allCondition>…</allCondition> XML element, you can have a sequence of either of the following XML elements (there's an implicit AND operator on all the XML elements in the sequence):

<allCondition>…</allCondition>

<anyCondition>…</anyCondition>

<anyNotCondition>…</anyNotCondition>

<notCondition>…</notCondition>

<dateFilterStatement>…</dateFilterStatement>

<stringFilterStatement>…</stringFilterStatement>

<intFilterStatement>…</intFilterStatement>

<doubleFilterStatement>…</doubleFilterStatement>

<instanceOfFilterStatement>…</instanceOfFilterStatement>

< isPresentFilterStatement>…</isPresentFilterStatement>

Each of these XML elements can have an optional tag attribute (see section 3.5.4 “Filter tags” for more information on filter tags).

3.5.3.1.3. The <anyCondition> XML element

<anyCondition>…</anyCondition> XML elements define a group of conditions, only one of which must be true in order for the “anyCondition” to be true.

Inside an <anyCondition>…</anyCondition> XML element, you can have a sequence of either of the following XML elements (there’s an implicit OR operator on all the XML elements in the sequence):

<allCondition>…</allCondition>

<anyCondition>…</anyCondition>

<anyNotCondition>…</anyNotCondition>

<notCondition>…</notCondition>

<dateFilterStatement>…</dateFilterStatement>

<stringFilterStatement>…</stringFilterStatement>

<intFilterStatement>…</intFilterStatement>

<doubleFilterStatement>…</doubleFilterStatement>

<instanceOfFilterStatement>…</instanceOfFilterStatement>

< isPresentFilterStatement>…</isPresentFilterStatement>

Each of these XML elements can have an optional tag attribute (see section 3.5.4 “Filter tags” for more information on filter tags).

3.5.3.1.4. The <anyNotCondition> XML element

<anyNotCondition>…</anyNotCondition> XML elements define a group of conditions, only one of which must be false in order for the “anyNotCondition” to be true.

Inside an <anyNotCondition>…</anyNotCondition> XML element, you can have a sequence of either of the following XML elements (there’s an implicit OR operator on all the XML elements in the sequence):

<allCondition>…</allCondition>

<anyCondition>…</anyCondition>

<anyNotCondition>…</anyNotCondition>

<notCondition>…</notCondition>

<dateFilterStatement>…</dateFilterStatement>

<stringFilterStatement>…</stringFilterStatement>

<intFilterStatement>…</intFilterStatement>

<doubleFilterStatement>…</doubleFilterStatement>

<instanceOfFilterStatement>…</instanceOfFilterStatement>

< isPresentFilterStatement>…</isPresentFilterStatement>

Each of these XML elements can have an optional tag attribute (see section 3.5.4 "Filter tags" for more information on filter tags).

3.5.3.1.5. The <notCondition> XML element

<notCondition>…</notCondition> XML elements define a group of conditions that must all be false in order for the "notCondition" to be true.

Inside an <notCondition>…</notCondition> XML element, you can have a sequence of either of the following XML elements (there's an implicit AND operator on all the XML elements in the sequence):

<allCondition>…</allCondition>

<anyCondition>…</anyCondition>

<anyNotCondition>…</anyNotCondition>

<notCondition>…</notCondition>

<dateFilterStatement>…</dateFilterStatement>

<stringFilterStatement>…</stringFilterStatement>

<intFilterStatement>…</intFilterStatement>

<doubleFilterStatement>…</doubleFilterStatement>

<instanceOfFilterStatement>…</instanceOfFilterStatement>

< isPresentFilterStatement>…</isPresentFilterStatement>

Each of these XML elements can have an optional tag attribute (see section 3.5.4 "Filter tags" for more information on filter tags).

3.5.3.1.6. The <dateFilterStatement> XML element

<dateFilterStatement>…</dateFilterStatement> XML elements define a condition on an attribute of the Alarm of type "date".

Inside an <dateFilterStatement>…</dateFilterStatement> XML element, you must have a sequence of the following mandatory XML elements:

<fieldName>…</fieldName>. The value of the <fieldName>…</fieldName> XML element must be the name of an attribute of the Alarm of type "date", for example "alarmRaisedTime"

<operator>…</operator>. The following values are possible for the <operator>…</operator> XML element:

- isBefore
- isAfter
- isEqual

<fieldValue>…</fieldValue>. The value of the <fieldValue>…</fieldValue> XML element must be a "date" constant, for example "2009-09-16T10:44:55.803+02:00"

<dateFilterStatement> XML elements can have an optional tag attribute (see section 3.5.4 "Filter tags" for more information on filter tags).

3.5.3.1.7. The <stringFilterStatement> XML element

<stringFilterStatement>…</stringFilterStatement> XML elements define a condition on an attribute of the Alarm of type "string".

Inside an <stringFilterStatement>…</stringFilterStatement> XML element, you must have a sequence of the following mandatory XML elements:

<fieldName>…</fieldName>. The value of the <fieldName>…</fieldName> XML element must be the name of an attribute of the Alarm of type "string", for example "originatingManagedEntity"

<operator>…</operator>. The following values are possible for the <operator>…</operator> XML element:

- isEqual
- isNotEqual
- contains
- doesNotContain
- matches
- startsWith
- endsWith
- isListedIn

<fieldValue>…</fieldValue>. The value of the **fieldValue** XML element must be a "string" constant, for example: *"BOX B1"*.

<stringFilterStatement> XML elements can have an optional tag attribute (see section 3.5.4 "Filter tags" for more information on filter tags).

matches operator

When the operator is "matches", then the value of the **fieldValue** XML element must be a "string" constant representing a Regular Expression, for example "BOX .*".

The Regular Expression must match the whole value of the **fieldValue** XML element, not just a sub-string. If you want to match a sub-string of the **fieldValue** XML element, please use the "contains" operator instead.

For a full detail of the syntax of Regular Expressions used by the "matches" operator, please refer to:

http://docs.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html

Note

Please note that by default the Regular Expression .* will not match character strings that contain line terminators. This is the default behavior of the "matches" operator. If you want to use a Regular Expression that will match character strings that contain line terminators, then you should add (?s) to your Regular Expression.

For example, the Regular Expression (?s).* will match any multi-line character string, whereas the Regular Expression .* will only match single-line character strings.

isListedIn operator

When the operator is "isListedIn", then the **fieldValue** represents a comma separated list of strings.

Strings containing commas must be quoted with double-quotes.

The double-quotre character must be escaped with a backslash '\' in a quoted string. For example: `BOX B1, "BOX, B2",BOX \"B3\".`

3.5.3.1.8. The <intFilterStatement> XML element

<intFilterStatement>…</intFilterStatement> XML elements define a condition on an attribute of the Alarm of type "integer".

Inside an <intFilterStatement>…</intFilterStatement> XML element, you must have a sequence of the following mandatory XML elements:

<fieldName>…</fieldName>. The value of the <fieldName>…</fieldName> XML element must be the name of an attribute of the Alarm of type "integer".

<operator>…</operator>. The following values are possible for the <operator>…</operator> XML element:

- isEqual
- isNotEqual
- isLower
- isGreater
- isLowerEqual
- isGreaterEqual

<fieldValue>…</fieldValue>. The value of the <fieldValue>…</fieldValue> XML element must be an "integer" constant, for example "12345"

<intFilterStatement> XML elements can have an optional tag attribute (see section 3.5.4 "Filter tags" for more information on filter tags).

3.5.3.1.9. The <doubleFilterStatement> XML element

<doubleFilterStatement> is an XML element similar to <intFilterStatement>except that the condition value is a "double".

3.5.3.1.10. The < instanceOfFilterStatement> XML element

< instanceOfFilterStatement> XML elements define a condition on the class of the incoming Event.

Inside an < instanceOfFilterStatement>…</instanceOfFilterStatement> XML element, you must have a sequence of the following mandatory XML element:

<fullClassName>…</fullClassName>. The value of this XML element must be the fully qualified class name of the incoming Event.

Example for an Alarm:
<fullClassName>com.hp.uca.expert.alarm.AlarmCommon</fullClassName>

<instanceOfFilterStatement> XML elements can have an optional tag attribute (see section 3.5.4 "Filter tags" for more information on filter tags).

3.5.3.1.11. The < isPresentFilterStatement> XML element

<isPresentFilterStatement> XML elements define a condition on the presence of a specific attribute of the Alarm.

Inside an <isPresentFilterStatement>…</isPresentFilterStatement> XML element, you must have a sequence of the following mandatory XML elements:

<fieldName>...</fieldName>. The value of the <fieldName>...</fieldName> XML element must be the name of the required attribute to be part of the Alarm.

<isPresentFilterStatement> XML elements can have an optional tag attribute (see section 3.5.4 “Filter tags” for more information on filter tags).

Note

☞ For more information on the XML schema of filter files, please refer to schema files delivered in `${UCA_EBC_HOME}/schemas`

3.5.3.2 Filter definition sample

The following example defines a top filter where all conditions must be true (there’s an implicit AND operator between each condition because the conditions are defined inside a <allCondition>...</allCondition> XML tag):

- alarmRaisedTime after 2009-09-16T10:44:55.803+02:00
- originatingManagedEntity matches “BOX .*”

```
<?xml version="1.0" encoding="UTF-8" ?>
<filters xmlns="http://hp.com/uca/expert/filter">
   <topFilter name="test">
     <allCondition>
       <dateFilterStatement>
         <fieldName>alarmRaisedTime</fieldName>
         <operator>isAfter</operator>
         <fieldValue>2009-09-16T10:44:55.803+02:00</fieldValue>
       </dateFilterStatement>
       <stringFilterStatement>
          <fieldName>originatingManagedEntity</fieldName>
          <operator>matches</operator>
          <fieldValue>BOX .*</fieldValue>
        </stringFilterStatement>
     </allCondition>
   </topFilter>
</filters>
```

XML Configuration 3 - Filter definition example

3.5.4 Filter tags file

The filter definition XML schema lets you define “tags” at various levels of the definition of the filter. Tags are a way to assign a name (a tag) to conditions or groups of conditions in a filter, so that once the alarm has passed a filter it is possible to retrieve the list of conditions or groups of conditions that were true when the alarm was evaluated against the filter.

To retrieve the list of such tags that were true when the alarm was evaluated against the filter, the `getPassingFiltersTags(topFiltername)` method on the Alarm object can be used.

Below is an example of the definition of a top filter that contains tags:

```
<?xml version="1.0" encoding="UTF-8" ?>
<filters xmlns="http://hp.com/uca/expert/filter">
```

```
    <topFilter name="test">
      <anyCondition tag="BOX">
        <stringFilterStatement>
           <fieldName>originatingManagedEntity</fieldName>
           <operator>matches</operator>
           <fieldValue>BOX .*</fieldValue>
         </stringFilterStatement>
      </anyCondition>
      <anyCondition tag="CARD">
        <stringFilterStatement>
           <fieldName>originatingManagedEntity</fieldName>
           <operator>matches</operator>
           <fieldValue>CARD .*</fieldValue>
         </stringFilterStatement>
      </anyCondition>
    </topFilter>
</filters>
```

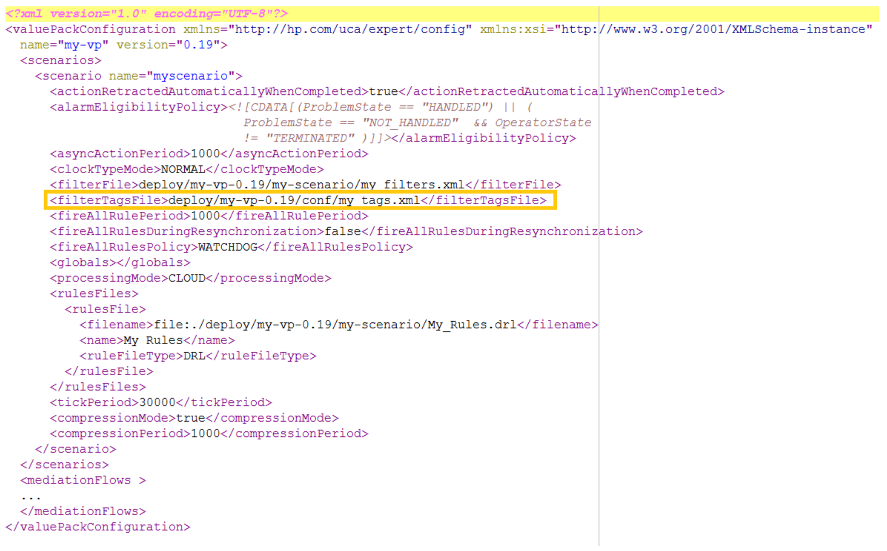
XML Configuration 4 - Filter definition example that contains tags

3.5.4.1 Editing Filter Tags with UCA for EBC Admin GUI

Within your scenario definition file, you have (since UCA for EBC V3.0) the possibility to specify a list of tags that can be understood by the UCA for EBC Admin GUI in order to display a tag editor form when filling out the tag of a filter. This greatly eases the use of tags within your scenario and ensures that you only set valid tags in your filter file(s).

Just like for the filter file, the name of the filter tags file can be anything, but the path for the filter tags file must be recorded in the configuration file for the Value Pack (the *`${UCA_EBC_INSTANCE}/deploy/<myValuepack>-<version>/conf/ValuePackConfiguration.xml`* file).

In the *`ValuePackConfiguration.xml file`*, the path for the filter tags file must be specified inside a `<filterTagsFile>…</filterTagsFile>` XML tag (inside a `<scenario>…</scenario>` XML tag), as shown in the screen capture below:



```
<?xml version="1.0" encoding="UTF-8"?>
<valuePackConfiguration xmlns="http://hp.com/uca/expert/config" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  name="my-vp" version="0.19">
  <scenarios>
    <scenario name="myscenario">
      <actionRetractedAutomaticallyWhenCompleted>true</actionRetractedAutomaticallyWhenCompleted>
      <alarmEligibilityPolicy><![CDATA[(ProblemState == "HANDLED") || (
                          ProblemState == "NOT_HANDLED"  && OperatorState
                          != "TERMINATED" )]]></alarmEligibilityPolicy>
      <asyncActionPeriod>1000</asyncActionPeriod>
      <clockTypeMode>NORMAL</clockTypeMode>
      <filterFile>deploy/my-vp-0.19/my-scenario/my_filters.xml</filterFile>
      <filterTagsFile>deploy/my-vp-0.19/conf/my_tags.xml</filterTagsFile>
      <fireAllRulePeriod>1000</fireAllRulePeriod>
      <fireAllRulesDuringResynchronization>false</fireAllRulesDuringResynchronization>
      <fireAllRulesPolicy>WATCHDOG</fireAllRulesPolicy>
      <globals></globals>
      <processingMode>CLOUD</processingMode>
      <rulesFiles>
        <rulesFile>
          <filename>file:./deploy/my-vp-0.19/my-scenario/My_Rules.drl</filename>
          <name>My Rules</name>
          <ruleFileType>DRL</ruleFileType>
        </rulesFile>
      </rulesFiles>
      <tickPeriod>30000</tickPeriod>
      <compressionMode>true</compressionMode>
      <compressionPeriod>1000</compressionPeriod>
    </scenario>
  </scenarios>
  <mediationFlows >
  ...
  </mediationFlows>
</valuePackConfiguration>
```

Figure 7 - Referencing a filter tags file inside a ValuePackConfiguration.xml file

Then you have to populate your file according the *`uca-expert-filter-tags.xsd`* XML schema definition file (a copy of this file is available in the

${UCA_EBC_HOME}/schemas directory). Below is an example of a filter tags XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<tags xmlns="http://hp.com/uca/expert/filter/tags">
  <groups>
    <group>
       <simpleTags>
               <simpleTag name="BOX"/>
               <simpleTag name="CARD"/>
       </simpleTags>
    </group>
    <group name="correlation">
       <paramTags>
               <paramTag name="CORR_KEY" enum="Node,Location,null"/>
               <paramTag name="CORR_KEY2" enum="Node,Location,null"/>
               <paramTag name="CORR_NAME" />
               <paramTag name="CORR_PRIORITY" />
       </paramTags>
    </group>
    <group name="trigger">
       <simpleTags>
               <simpleTag name="Trigger"/>
       </simpleTags>
       <paramTags>
               <paramTag name="TIME_LIMIT_SECOND" />
               <paramTag name="TRIGGER_PRIORITY" />
       </paramTags>
    </group>
  </groups>
</tags>
```

As shown above, the root XML tag of a filter tags file is: `<tags>…</tags>`.

Inside this root XML, a list of groups (called tags groups) can be defined inside a `<groups>…</groups>` XML tag.

Each tags group is defined by a `<group>…</group>` XML tag. Tags groups can be assigned a name by setting the `name` attribute of a `<group>…</group>` XML tag. For example: `<group name=”correlation”>…</group>`. The `name` attribute is optional. If not present, the tags defined in the tags group are considered “general purpose” and can be used as tags anywhere in an associated filter file. If present, the tags defined in the tags group are considered “specific” and can be used as tags in an associated filter file, only inside top filters where the `tagsGroup` attribute of the top filter equals the `name` attribute of the tags group or where the `tagsGroup` attribute of the top filter contains a list of tags groups names, one of which equals the `name` attribute of the tags group.

There are two kinds of tags that can be defined in a tags group:

Simple tags: tags are defined as simple free text strings

Parameter tags: tags are defined as key/value pairs. The value can either be a simple free text string or a text value from a list of possible values.

All tags are defined inside a `<group>…</group>` XML tag. Simple tags are defined inside using a `<simpleTags>…</simpleTags>` XML tag, whereas parameter tags are defined inside using a `<paramTags>…</paramTags>` XML tag.

Each simple tag is defined by a `<simpleTag>…</simpleTag>` XML tag. The mandatory `name` attribute indicates the name of the tag, to be used when tagging filter conditions inside a filter file by using the `tag` attribute. For example:

<simpleTag name="Trigger"/>

Each parameter tag is defined by a `<paramTag>…</paramTag>` XML tag. The mandatory `name` attribute indicates the name of the tag, to be used when tagging filter conditions inside a filter file by using the `tag` attribute.

The `<paramTag>` can have optional attributes:

Attribute	**Description**
enum	Used to specify a list of authorized values for that tag.
type	Used by UI to control the edition of the value for that tag. Can be set for example to “numeric”, “boolean”, etc… Is also used by UI Filter Builder utility to separate tags when this attribute is set to “separator” or “submenu”.
default	Used by UI Filter Builder utility to set a default value when adding this tag.
tooltip	Used by UI Filter Builder utility to display a comment when mouse is over that tag.
use	When set to “required”, it means to the UI Filter Builder utility that this tag should always be present.

If the optional `enum` attribute is omitted, the value of the tag is understood to be a simple free text string. For example:

<paramTag name="CORR_NAME"/>

On the other hand, if the optional `enum` attribute is present, the value of the tag is to be one value among the list of possible values indicated by the value of the `enum` attribute. The value of the `enum` attribute should be a comma-separated list of possible values for the tag. For example:

<paramTag name="CORR_KEY" enum="Node,Location,null"/>

In the associated filter file, filter conditions can be properly tagged according to the filter tag file using a comma-separated list of valid simple tags or parameter tags. For example:

<allCondition tag="Trigger,CORR_NAME=LTE,CORR_KEY=Node">

Filter file tags can be easily edited in the filter file using the UCA for EBC Admin GUI. Clicking on the “Edit tag” icon will display a tag editor form that abides by the associated filter tag file. The tag editor displays groups of tags according to the following algorithm:

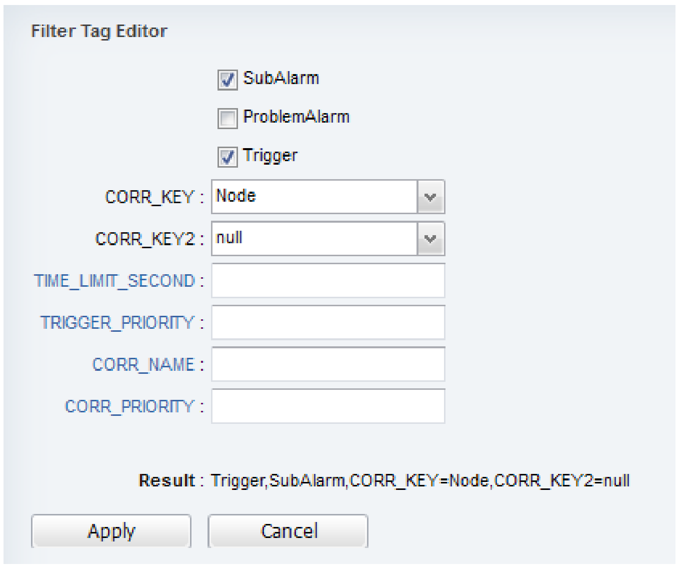
If the <topFilter> `tagsGroup` attribute is not specified if the filter file, all groups are displayed.

If the <topFilter> `tagsGroup` attribute is specified in the filter file, then

All groups (defined in the filter tags file) with no “name” attribute are displayed

All groups (defined in the filter tags file) with a “name” attribute matching the `tagsGroup` attribute are displayed. If the tagsGroup contains a list of comma-separated group names, all groups (defined in the filter tags file) matching one value in the list will be displayed.

The below example displays the tags from all groups defined in the *`my_tags.xml`* file because the filter condition of the filter file currently being edited at the UCA for EBC Admin GUI is part of a topFilter that doesn’t have an associated `tagsGroup` attribute:



[✓] SubAlarm
[ ] ProblemAlarm
[✓] Trigger
CORR_KEY : Node
CORR_KEY2 : null
TIME_LIMIT_SECOND :
TRIGGER_PRIORITY :
CORR_NAME :
CORR_PRIORITY :
Result : Trigger,SubAlarm,CORR_KEY=Node,CORR_KEY2=null

Figure 8 - Editing Filter Tags with UCA for EBC Admin GUI

If you use parameter tags in your filter file(s), you can retrieve the key/value pairs of the parameter tags associated with an Alarm that passed the filters by using the `getPassingFiltersParams()` method on the Alarm object.

You can also edit the filter tags using the Filter Builder utility within UCA for EBC Admin GUI.

☞[R10] *Unified Correlation Analyzer for Event Based Correlation – User Interface Guide*, chapter Troubleshooting UCA for event based Correlation

3.5.5 Mappers definition file

The Mappers are mainly used in the context of topology aware Value packs.

This configuration file offers mainly two different functionalities:

A way to establish the correspondence between some data contained in the received alarm and a topology instance during the Alarm Enrichment process (see Chapter 3.5.6 Alarm Enrichment).

A way to define Cypher Queries for retrieving information in the graph database

The Scenario mappers file is defined using the **<mapperFile>** tag in the scenario configuration file.

The mappers definition file is stored in the scenario directory: *${UCA_EBC_INSTANCE}/deploy/<myValuepack>-<version>/<myScenario>/mappers.xml*

Where:

- *myValuepack* is the Value pack name
- *version* is the Value pack version
- *myScenario* is the scenario name.

3.5.5.1 Mapper definitions

The mappers' configuration file can define several mappers. Usually a mapper is dedicated to a certain type of Alarm meaning that one mapper should be defined per alarm type.

A mapper is identified with a name and is made of either:

- One or a set of 'extract' instructions
- A 'pattern' instruction

3.5.5.1.1. The 'extract' mapper

Each extract instruction allows extracting all or a portion of the specified alarm attribute thanks to a regular expression.

The alarm attribute is defined using the <fieldName> tag.

This regular expression is defined thanks to two tags: the <matcher> tag and the <mappedTo> tag.

The <matcher> Tag defines the string subsets to extract, i.e. the part of the regular expression that is within parenthesis.

Whereas the <mappedTo> Tag indicate how to use the extracted substring(s).

Here below is an example:

```
<mapper name='retrieveSwitchName'>
        <extract>
            <fieldName>originatingManagedEntity</fieldName>
            <matcher>SWITCH (.*)$</matcher>
            <mappedTo>$1</mappedTo>
        </extract>
</mapper>
```

The <extract> element has also two optional attributes:

Attribute	**Description**
replaceAll	This attribute of Boolean type is used to specify what to do when multiple patterns are matching. When false, only first matching pattern will be applied. When true, all matching patterns will be applied
unchangedValuetIfNoMatch	This attribute of Boolean type is used to specify what to do when no pattern is matching. When false, it means that the result value will be empty. When true, the value of fieldName is left unchanged.

Finally, all the extracted substrings are concatenated together in order to make the final mapping string.

3.5.5.1.2. The ‘pattern’ mapper

A pattern mapper is used to build a mapping result from other mappers.

The regular expression defining which other mappers are to be used is the <expression> tag.

The result is defined thanks to two tags: the <matcher> tag and the <mappedTo> tag.

The <matcher> tag defines the string subsets to extract, i.e. the part of the regular expression that is within parenthesis.

Whereas the <mappedTo> tag indicate how to use the extracted substring(s).

Let's suppose we have defined mappers named 'btsID' and 'location' to extract values from corresponding fields and we would like to have a mapper that concatenates the results of those 2 ‘extract’ mappers. A pattern can be defined as per example below to achieve this:

```
<mapper name='NodeB_UniqueID'>
   <pattern>
      <expression>[btsID]~[location]</expression>
      <matcher>(.*)</matcher>
      <mappedTo>$1</mappedTo>
   </pattern>
</mapper>
```

A pattern mapper has few optional attributes:

Attribute	**Description**
regex	This attribute defines the regular expression to match a mapper name
removed	This attribute defines the regular expression to be ignored from the matcher in order to form the mapper name
replaceAll	This attribute of Boolean type is used to specify what to do when multiple patterns are matching. When false, only first matching pattern will be applied. When true, all matching patterns will be applied
bestEffortIfNoMatch	This attribute of Boolean type is used to specify what to do when not all patterns are matching. When false, it means that the result value will be empty. When true, the mappers are executed on best effort basis.

Let's suppose you want to use ${x} instead of [] to define your mapper names.
So you would set the two first attributes as per example below:

<pattern regex="\$\{\w*\}" removed="\$\{|\}">

3.5.5.1.3. Mapper Example

This is an example of mapper usage.

With this example the mappers file is as follow:

```
<?xml version="1.0" encoding="UTF-8" ?>
<mappers xmlns="http://hp.com/uca/expert/instancemapper">
   <mapper name="TopoInstance">
        <extract>
            <fieldName>originatingManagedEntity</fieldName>
            <matcher>(.*) (.*)</matcher>
            <mappedTo>$1_$2</mappedTo>
        </extract>
        <extract>
            <fieldName>additionalInformation</fieldName>
            <matcher>.*:([0-9]*):.*</matcher>
            <mappedTo>:$1</mappedTo>
        </extract>
    </mapper>
</mappers>
```

This file is defining one Mapper Called “TopoInstance”.

In the previous example, this mapper is made of the concatenation of two pieces of information. The first one coming from the originatingManagedEntity attribute of the alarm, and the second one from the additionalInformation attribute.

From the ‘originatingManageEntity’ attribute we expect to have a string made of two words. The regular expression (<matcher>) defines two substrings (one for each word of the ‘originatingManageEntity’ attribute). The <mappedTo> section tells how to put these two words together in the final extract substring. In our case the two words are concatenated with a ‘_’ character between the two.

From the ‘additionalInformation’ attribute, the <matcher> Tag expression extracts a number located between two semicolon characters. The <mappedTo> string produces a string starting with a semicolon and followed by the extracted number.

Finally the two strings are concatenated together.

Now when the following alarm is received:

```
<Alarms xmlns="http://hp.com/uca/expert/x733Alarm">
    <AlarmCreationInterface>
        <sourceIdentifier>src</sourceIdentifier>
        <identifier>12300</identifier>
        <originatingManagedEntity>BOX B1</originatingManagedEntity>
        <alarmType>PROCESSING_ERROR_ALARM</alarmType>
        <probableCause>Fire</probableCause>
        <specificProblem>Fire</specificProblem>
        <perceivedSeverity>CRITICAL</perceivedSeverity>
        <alarmRaisedTime>2009-09-16T12:00:00.000+02:00</alarmRaisedTime>
        <additionalText>Alarm text</additionalText>
        <additionalInformation>network info:6502: status
failed</additionalInformation>
        <proposedRepairActions>action</proposedRepairActions>
        <customFields>
            <customField name="field1" value="v1"/>
            <customField name="field2" value="v2"/>
            <customField name="field3" value="100"/>
        </customFields>
    </AlarmCreationInterface>
</Alarms>
```

The produced mapping is : BOX_B1:6502

3.5.5.1.4. Using the mappers

Mappers are typically used during alarm enrichment process (see Chapter 3.5.6 Alarm Enrichment).

The resulting string of a given mapper can be retrieved using the static `doMapping()` method below with the alarm object as input parameter.

Example:

String topoIdentifier = MapperUtils.doMapping(alarm, “TopoInstance”);

Doing this way the mapping between the alarm and the corresponding topology instance is fully configurable through configuration file and does not require any code change.

The mappers can also be retrieved from the scenario object. This is done using the method `getMappers()`. To retrieve a specific mapper (for example a mapper named “NetworkInstance”), the following must be done:

theScenario.getMappers.getMapper("NetworkInstance");

3.5.5.2 Cypher queries definitions

The mappers’ configuration file can define several cypher queries. A cypher query is identified by a name and defines a cypher request.

Example of definitions of two queries: ‘GetPortLink’ and ‘GetRemotePort’

```
<?xml version="1.0" encoding="UTF-8" ?>
<mappers xmlns="http://hp.com/uca/expert/instancemapper">
   <cypherQuery name='GetPortLink'>
        <query><![CDATA[START n=node:PortsByUniqueId(uniqueId = {portName})
MATCH (n)<-[link:LINK]->() RETURN link]]></query>
   </cypherQuery>
   <cypherQuery name='GetRemotePort'>
        <query><![CDATA[START n=node:PortsByUniqueId(uniqueId = {portName})
MATCH (n)<-[:LINK]->(m) RETURN m]]></query>
   </cypherQuery>
</mappers>
```

3.5.5.2.1. Using the cypher queries

Cypher queries can be used during alarm enrichment process or any other process that would required getting topology information within a scenario.

To use a cypher query, it first must be retrieved from the scenario and then executed thanks to the com.hp.uca.expert.topology.CypherQuery utility class.

The result of the query is then analysed to extract the required information.

Example of use:

```
        // Get the Scenario associated to the current thread
        Scenario theScenario = ScenarioThreadLocal.getScenario();

        // retrieve the Cypher query from configuration
        String query =
theScenario.getMappers().getCypherQuery("GetPortLink");

        Map<String,Object> params = new HashMap<String, Object>();
        params.put("portName", portName);

         // execute the query
        ExecutionResult result = CypherQuery.executeAndreturnResult(query,
params);

        // get information from the result (in this case the ‘LINK’ relation)
        ResourceIterator<Relationship> links = result.columnAs("link");
        if (links.hasNext()) {
            link  = links.next();
        }
```

3.5.5.3 Combining the use of Filter Tags and Mappers

As said above, a mapper may be very specific to each alarm type. On solutions dealing with alarms coming from several different sources, several mappers will be used.

More over the mapping between the alarm content and the targeted instance may evolve over time depending on the network elements versions. It is therefore very useful to adapt the instance mapping through configuration instead of going back to the code to support the changes.

For such solution, the association between Filter Tags and Mappers is used:

A specific Top Filter is used to characterize the Alarm Type (returning a Tag representing the Alarm Type)

The returned Tag is used to select the correct Instance Mapper

As both Filters and Mappers are configuration file, this solution can evolve without Java code modification

Example of Alarm Identification Top filter:

```
<?xml version="1.0" encoding="UTF-8" ?>
<filters xmlns="http://hp.com/uca/expert/filter">
    <topFilter name="AlarmType">
        <anyCondition>
            <anyCondition tag="Ciena_EVC”>
                <stringFilterStatement>
                    <fieldName>sourceIdentifier</fieldName>
                    <operator>contains</operator>
                    <fieldValue>ciena</fieldValue>
                </stringFilterStatement>
                <stringFilterStatement>
                    <fieldName>originatingManagedEntity</fieldName>
                    <operator>matches</operator>
                    <fieldValue>EVC_SEGMENT.*</fieldValue>
                </stringFilterStatement>
            </anyCondition>
            <anyCondition tag="2IP_LAG”>
                <stringFilterStatement>
                    <fieldName>sourceIdentifier</fieldName>
                    <operator>contains</operator>
                    <fieldValue>2IP</fieldValue>
                </stringFilterStatement>
                <stringFilterStatement>
                    <fieldName>originatingManagedEntity</fieldName>
                    <operator>matches</operator>
                    <fieldValue>LAG .*</fieldValue>
                </stringFilterStatement>
            </anyCondition>
        </anyCondition>
    </topFilter>
</filters>
```

This Top Filters identifies two types of alarms: *Ciena_EVC* and *2IP_LAG*

Then a mapper is written for each of these alarms:

```
<?xml version="1.0" encoding="UTF-8" ?>
<mappers xmlns="http://hp.com/uca/expert/instancemapper">
   <mapper name="Ciena_EVC">
        <extract>
            <fieldName>originatingManagedEntity</fieldName>
            <matcher>EVC_SEGMENT (.*)</matcher>
            <mappedTo>Ciena_EVC_$1</mappedTo>
        </extract>
    </mapper>
   <mapper name="2IP_LAG">
        <extract>
            <fieldName>originatingManagedEntity</fieldName>
            <matcher>LAG (.*)</matcher>
            <mappedTo>2IP_LAG_$1</mappedTo>
        </extract>
    </mapper>
</mappers>
```

The generic Alarm enrichment Code would be the following:

```
public EnrichedAlarm(Alarm alarm) {
    super(alarm);

    // retrieve the Alarm Type From Filter Tags
    Set<String> tags = alarm.getPassingFiltersTags().get("AlarmType");

    if ((tags != null) && (tags.size() != 0 )) {

    // get the instance mapping from mappers
    InstanceId =alarm.doMapping(tags.toArray()[0]);

    // use the mapped instance (topology query for instance)
    …

}
```

3.5.6 Alarm Enrichment

Alarm Enrichment is implemented by a Java Class that extends the `com.hp.uca.expert.lifecycle.alarm.AlarmLifeCycle` Java Class and overrides the following methods:

```
// ----------------------------------------------------------------------
// Default override processing on Alarm creation at Global system layer
// ----------------------------------------------------------------------
@Override
public Event onAlarmCreationProcess(Alarm alarm) {
        // by def, nothing
        return alarm;
}

// ----------------------------------------------------------------------
// Default override processing on Alarm deletion at Global system layer
// ----------------------------------------------------------------------
@Override
public Event onAlarmDeletionProcess(AlarmDeletion deletion) {
        return deletion;
}

// --------------------------------------------------------------------------
// Default override processing on Alarm state change at Global system layer
// ---------------------------------------------------------------------------
@Override
public Event onAlarmStateChangeProcess(AlarmStateChange stChange) {
        return stChange;
}

// ------------------------------------------------------------------
// Default override processing on Alarm AVC at Global system layer
// ------------------------------------------------------------------
@Override
public Event onAlarmAttributeValueChangeProcess(
                AlarmAttributeValueChange avc) {
        return avc;
}
```

The use of this new class must be declared in the `ValuePack configuration.xml` file in the proper scenario section. This is done by setting the <customLifeCycleClass> tag as follows:

```
<customLifeCycleClass>
  com.hp.uca.ebc.myVp.myScenarioLifeCycle
</customLifeCycleClass>
```

Where `com.hp.uca.ebc.myVp.myScenarioLifeCycle` is the Extended Scenario LifeCycle Class.

Alarm Enrichment information can be stored in any of the standard alarm attributes. The 'additionalText' attribute is usually a good candidate for storing this extra information. Another alternative can be to use a 'LocalVariable' associated with the Alarm.

However, in order to simplify the process of writing rules, it can be preferable to store enrichment information directly in some specific Alarm Object attributes. For this purpose, standard UCA for EBC alarm objects (Alarm, AlarmDeletion, AlarmAttributeValueChange and AlarmStateChange) can be extended.

In such a case, 'onAlarm*XXXX*Process()' methods from the extended AlarmLifeCycle Class can be written to use extended alarm objects:

```
@Override
public Event onAlarmCreationProcess(Alarm alarm) {
    LogHelper.enter(log, "onAlarmCreationProcess()");

    ExtendedAlarm extendedAlarm = new ExtendedAlarm (alarm);

    // do the Alarm enrichment from external source here!
    …

    LogHelper.exit(log, "onAlarmCreationProcess()");
    return extendedAlarm;
}
```

3.5.7 Scenario Initialization class

It is usually necessary setting up an environment or performing some initialization before starting the normal scenario processing.

This can be achieved by configuring the spring context in order to instanciate Java classes that perform this initialization.

An alternative to the use of the Spring context is to develop an initialization Class. This class must extend the `DefaultScenarioInitialization` class as follow:

```
package my.package;
public class MyScenarioInitialization extends DefaultScenarioInitialization {
    public MyScenarioInitialization(Scenario scenario,
            ValuePackApplicationContext valuePackApplicationContext) {
        super(scenario, valuePackApplicationContext);
    }
    @Override
    public void initializeScenario() throws UcaException {
        //put your code here
    }
    @Override
    public void disposeScenario() throws UcaException {
        //put your code here
    }
}
```

This initialization Class is declared in the Scenarion configuration section of the ValuepackConfiguration.xml file as follow:

```
<customInitializationClass>my.package.MyScenarioInitialization</customInitializationClass>
```

3.5.8 Rules files

Rules files can be either template rules files or standard rules files.

Each Scenario can have multiple Rule Files. The rules files are defined in the Valuepack configuration file (`ValuePackConfiguration.xml`). All rule files will be loaded at scenario initialization unless the **disabledAtStartup="true"** attribute is set for a specific `<rulesFile>...</rulesFile>` XML entity in the Valuepack configuration file. For example:

```
...
<rulesFiles>
  <rulesFile disabledAtStartup="true">
  <filename>file:./deploy/vp-1.0/scenario/rules.drl</filename>
  <name>alarmforwarder rules</name>
  <ruleFileType>DRL</ruleFileType>
  </rulesFile>
  ...
</rulesFiles>
...
```

When rules file are disabled at startup, they can be loaded/unloaded dynamically using Java code thanks to a couple of methods on the RuleSession object:

void	**loadRulesFile**(String rulesFileName) load or re-load the RulesFile given as parameter.
void	**unLoadRulesFile**(String rulesFileName) unload the RulesFile given as parameter

A specific rule can also be removed from the knowledgebase with the following method:

void	**removeRule**(String packageName, String ruleName) Remove the rule from the knowledge base.

Note

In order to preserve working memory integrity and to avoid interfering with the locking mechanism implemented by the UCA for EBC framework, Drools keywords MUST NOT be used directly when developping UCA for EBC rules.

In particular, all timer based keywords should be avoided: **duration, timer, calendar**.

3.5.8.1 Standard rules files

Standard Rules files are standard Drools '.drl' file.

These files are loaded by the Drools engine at Scenario start-up.

Then, the Drools Builder dynamically compiles the Rule Files to transparently generate Java classes representing the righ hand-side part of the rule (the 'then' part).

When firing a rule, the Drools engine will call the Java generated method(s).

Standard rules files are defined as follow in the ValuePackConfiguration.xml file:

```
<rulesFiles>
    <rulesFile>
        <filename>file:./src/test/resources/myScenarioRules.drl</filename>
        <name>rulesFileName</name>
        <ruleFileType>DRL</ruleFileType>
    </rulesFile>
</rulesFiles>
```

Where :

`<filename>`	defines the rules Files filename
`<name>`	is the rules file given name
`<ruleFileType>`	This is the RuleFile Type (DRL) for standard rule.

☞ See [R7] *Unified Correlation Analyzer for Event Based Correlation – Value Pack Development Guide*, for more information on How to write a Rule.

3.5.8.2 Template rule files

Template rules files are similar to standard rules files except that template rules file contain parameters. Template rules files are always associated with template parameters files wich is defined with an additional tag in the RulesFile definition:

```
<paramsFilename>file:./src/test/resources/myScenarioRules.xml</paramsFilename>
```

It is useful to use template rules files when a same rule must be 'duplicated' in order to just change one or two parameters like a threshold value, a time window, etc...

The concept of 'Template rule' exists in Drools and is fully integrated in UCA for EBC product. Indeed, instead of defining a rule with all specific values, it is possible to define a rule in a template section with some 'variables' included.

When a template rules file is loaded in the engine, it is compiled with its associated parameters file that provides values for the different variables. The same rule can be instantiated several times depending on the number of definitions in the parameter file

Template file parameters can be modified without modifying the template rules file itself. Updating the parameters and reloading the rule or scenario is enough to make the Drools engine use the new parameters (no UCA for EBC restart is needed).

3.5.8.2.1. Basics

A template rules file may contain one or multiple rules. The file suffix of a template rule file is '.xdrl'

Here are the different parts that compose a template rules file:

```
template header
var-name1
var-name2

package package-name

imports

globals

functions

template "name"

queries

rules (that contain $var-name1 and $var-name2)

end template
```

Rule Sample 3 - Template rules file main components

Template header

This corresponds to the list of variables used in the template rule. When the template is compiled, each variable defined in the 'template header' section is replaced by its actual value (from the template parameters file).

Template “name”

This defines the beginning of the template’s scope. From this point on, the compilation of the file will replace any variable by its associated value. If several values exist for variable, the rules that use this variable will be duplicated. The Template section ends by the ‘end template’ tag.

Note

The import, global and functions sections of the file can not be in the scope of the template, i.e. variables cannot be used in these sections.

3.5.8.2.2. Sample of Templates rules file

Template rules files can be used for scenarios that are either in STREAM or CLOUD modes. The definition of a template rules file is almost the same as the definition of a standard rules file in the *ValuePackConfiguration.xml* file:

```
<?xml version="1.0" encoding="UTF-8"?>
<valuePackConfiguration
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       name="myValuePack" version="1.0">

<scenarios>
    <scenario name="myScenario">
        <filterFile>deploy/myValuePack-
1.0/myScenario/myfilter.xml</filterFile>
        <fireAllRulesPolicy>EACH_ACCESS</fireAllRulesPolicy>
        <globals></globals>
        <processingMode>STREAM</processingMode>
        <rulesFiles>
            <rulesFile>
                <filename>file:./deploy/myValuePack-
1.0/myScenario/myTemplate.drl</filename>
                <name>myRules</name>
                <paramsFilename>file:./deploy/myValuePack-
1.0/myScenario/myParams.xml</paramsFilename>
                <ruleFileType>XDRL</ruleFileType>
            </rulesFile>
        </rulesFiles>
    </scenario>
</scenarios>

</valuePackConfiguration>
```

XML Configuration 5 - Template definition in Scenario

The template rule is defined as ‘XDRL’ type and requires an extra parameter that corresponds to the file that contains the list of variables with associated values.

Let’s imagine a case where you want to count each time you received x fault alarms over a certain period of time. The following template rules file illustrates this example:

```
template header
thresholdValue
action
timewindow
```

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.PerceivedSeverity;

declare Alarm
  @role( event )
  @timestamp( dateTimestamp )
  @expires( 60s )
end

template "my scenario template"

rule "Count ${thresholdValue} faults"
when
  a: Alarm(perceivedSeverity != PerceivedSeverity.CLEAR)

  $alarms : ArrayList(size == (${thresholdValue} - 1))
      from collect(
         Alarm(this != $a, this after [0, ${timewindow}] $a) )
then
  ${action}
end
end template
```

Rule Sample 4 - Template Rules file example

Now, let’s see the associated parameters file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<RuleParametersCollection
xmlns="http://hp.com/uca/expert/engine/template">
        <RuleParameters>
                <parameter name="thresholdValue">
                        <value>30</value>
                </parameter>
                <parameter name="action">
                        <value> System.out.println("########## WARNING
(30 alarms) received ###########");</value>
                </parameter>
                <parameter name="timewindow">
                        <value>30s</value>
                </parameter>
        </RuleParameters>
        <RuleParameters>
                <parameter name="thresholdValue">
                        <value>600</value>
                </parameter>
                <parameter name=" action ">
                        <value> System.out.println("########## CRITICAL
(600 alarms) received ###########");</value>
                </parameter>
                <parameter name=" timewindow ">
                        <value>60s</value>
                </parameter>
        </RuleParameters>
<RuleParametersCollection>
```

XML Configuration 6 - Rule Parameter file example

Important Note

Although the Drools engine supports Template Parameters Files in either XML File or Properties File format, UCA for EBC only supports XML File format.

In real time, when such a template rules file is compiled, the following two rules (notice the duplication of the same rule twice) are generated and loaded in the engine memory:

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.PerceivedSeverity;

declare Alarm
  @role( event )
  @timestamp( dateTimestamp )
  @expires( 60s )
end

rule "Count 30 faults"
when
  a: Alarm(perceivedSeverity != PerceivedSeverity.CLEAR)

  $alarms : ArrayList(size == (30 - 1))
      from collect(
         Alarm(this != $a, this after [0, 30s] $a) )
then
  System.out.println("########## WARNING (30 alarms) received
###########");
End

rule "Count 600 faults"
when
  a: Alarm(perceivedSeverity != PerceivedSeverity.CLEAR)

  $alarms : ArrayList(size == (600 - 1))
      from collect(
         Alarm(this != $a, this after [0, 60s] $a) )
then
  System.out.println("########## CRITICAL (60 alarms) received
###########");
End
```

Rule Sample 5 - Rule generated from Template example

The usage of template rules files avoids the duplication of rules that would have same structure and where "generic" rule can be written instead.

3.6 Value Pack Life Cycle

The following picture explains the Value pack life cycle within the UCA for EBC product:

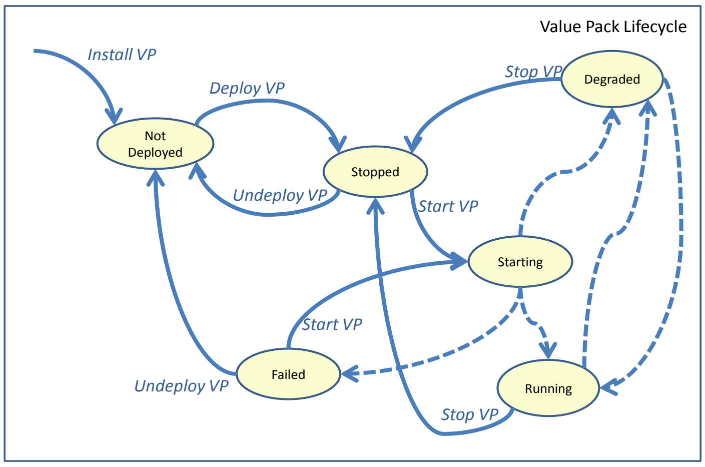


Figure 9 - Value Pack Life Cycle

Bold lines transitions indicate a specific action on a Value Pack (deploy, start, stop, etc…).

Dotted lines transitions indicate either internal processing or a problem.

Running state: all scenarios are in “Running” state and so is the mediation.

Failed state: in case of XML file configuration problem, or when all scenarios of the Value Pack are in a “Failed” or “Degraded” state.

Degraded state: when the state of one or more scenario is “Degraded”, and/or the mediation is not available.

3.6.1 Installing a Value Pack

An UCA for EBC Value pack is a zip file generated using the UCA for EBC Development toolkit.

To install a Value Pack, you need to copy the zip file in the *${UCA_EBC_INSTANCE}/valuepacks* directory.
Or you can use the UCA GUI Dashboard (UCA for EBC > Application > Monitoring) to directly upload your Value Pack from your development station, of course being logged with admin or developper rights on GUI.

No other action is needed to install a value pack. The UCA for EBC server will automatically detect the newly installed Value pack. This value pack will then be visible from the UCA GUI Dashboard (UCA for EBC > Application > Monitoring).

☞ Refer to [R10] *Unified Correlation Analyzer for Event Based Correlation – User Interface Guide* for all GUI Administration features.

3.6.2 Deploying a Value Pack

Deploying a value pack can be done in two ways:

From the command line, by executing the following commands (executed as the "uca" user):

```
$ cd ${UCA_EBC_HOME}/bin
$ uca-ebc-admin --deploy -vpn valuepackName -vpv valuepackVersion
```

Where *valuepackName* and *valuepackVersion* are the name and version of the value pack to deploy (example: llef-example 1.0)

From the Web GUI.

By clicking on the "deploy" button from the Value pack Monitoring view.

3.6.2.1 File organization

At the end of the deployment step, the files delivered by the Value Pack are deployed in the

${UCA_EBC_INSTANCE}/deploy/{valuepackName}-{valuepackVersion} directory.

Directories	**Description**
`lib/`	This directory contains the jar files needed by the Value Pack
`conf/`	This directory contains the configuration files that defines the Value Pack and its scenarios
`<Scenario Name>/`	There is one *<Scenario Name>/* for each scenario of the Value Pack. Each directory is named after the scenario and contains all the rules files (including filter and parameter files) for the scenario.

Table 8 - File structure of a deployed Value Pack

3.6.3 Starting a Value Pack

Starting a value pack can be done in two ways:

From the command line, by executing the following commands (executed as the "uca" user):

```
$ cd ${UCA_EBC_HOME}/bin
$ uca-ebc-admin --start -vpn valuepackName -vpv valuepackVersion
```

Where *valuepackName* and *valuepackVersion* are the name and version of the value pack to start (example: llef-example 1.0)

From the Web GUI

By clicking on the "start" button from the Value pack Monitoring view.

Starting a Value Pack will also create all the mediation flows defined for this Value Pack in the mediation flows section of the *ValuePackConfiguration.xml* file.

3.6.4 Stopping a Value Pack

Stopping a value pack can be done in two ways:

From the command line, by executing the following commands (executed as the "uca" user)

```
$ cd ${UCA_EBC_HOME}/bin
$ uca-ebc-admin --stop -vpn valuepackName -vpv valuepackVersion
```

Where *valuepackName* and *valuepackVersion* are the name and version of the value pack to stop (example: llef-example 1.0)

From the Web GUI

By clicking on the “stop” button from the Value pack Monitoring view

Stopping a Value Pack will also delete the mediation flow(s) associated with this Value Pack.

3.6.5 Un-deploying a Value Pack

Un-deploying a value pack can be done in two ways:

From the command line, by executing the following commands (executed as the “uca” user)

```
$ cd ${UCA_EBC_HOME}/bin
$ uca-ebc-admin --undeploy -vpn valuepackName -vpv
valuepackVersion
```

Where *valuepackName* and *valuepackVersion* are the name and version of the value pack to un-deploy (example: llef-example 1.0)

From the Web GUI

By clicking on the “undeploy” button from the Value pack Monitoring view.

Undeploying a value pack performs the following actions:

- It removes the Value Pack from the ${UCA_EBC_INSTANCE}/deploy directory
- It make an archive ZIP file of the Value Pack and stores it in the *${UCA_EBC_INSTANCE}/valuepacks* directory (so that it can be deployed back later on). The zipped value pack that was previously present in the *${UCA_EBC_INSTANCE}/valuepacks* directory is moved to the *${UCA_EBC_INSTANCE}/archive* directory and a timestamp is added to the file name.

Once the value pack has been undeployed, it can be deployed back again by using the -deploy, --deploy option.

3.6.6 Removing a Value Pack

Once undeployed, a Value Pack can be fully removed either through a remove shell command:

```
$ rm ${UCA_EBC_INSTANCE}/valuepacks/vpName-vp-vpVersion.zip
```

Or from the Web GUI

By clicking on the “remove” button from the Value pack Monitoring view.

3.7 Scenario Life Cycle and Status

The following picture explains the Scenario life cycle within the UCA for EBC product:

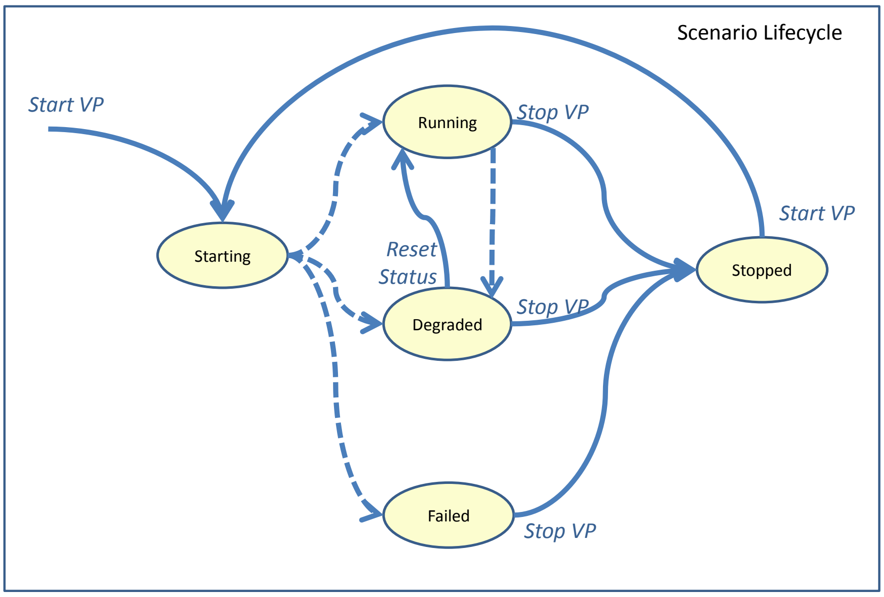


Figure 10 - Scenario Lifecycle

As shown in the figure above, most of the bold lines transitions are driven by the Value Pack itself.

Dotted lines transitions indicate internal processing or a problem:

- Running: all rules files have been loaded successfully.
- Failed: rules have not loaded successfully
- Degraded: a problem has been detected at run-time (exception in the right hand side (‘then’) part of the rule), usually a customer code problem.

Chapter 4

Scenario Policies

The behavior of a UCA for EBC scenario is driven by a set of configurable properties. These properties are called Policies.

The scenario properties modify the default behavior of a scenario to better suit your needs.

The following is the list of configurable Scenario Policies:

Processing policies:

- processingMode
- fireAllRulesPolicy
- fireAllRulesDuringResynchronization
- retractOnResyncPolicy
- compressionMode
- Time-related settings:
- fireAllRulePeriod
- tickPeriod
- asyncActionPeriod
- garbageCollectionPeriod

Filtering:

- alarmEligibilityPolicy

Garbage Collection:

- actionRetractedAutomaticallyWhenCompleted
- eligibleForBroadcast

☞ See *Chapter 3.5.2* "Scenario definition file" to learn how to apply these policies to a Scenario.

4.1.1 Processing policies

4.1.1.1 Processing Mode

This policy defines the Alarm processing Mode as defined in the section above (3.3.4 Alarm lifecycle).

The value can be **STREAM** or **CLOUD**.

4.1.1.2 fireAllRulesPolicy

This policy can take two possible values: EACH_ACCESS or WATCHDOG.

When EACH_ACCESS is chosen, each 'insert', 'update' or 'retract' operation on the Scenario's working memory will trigger the rules evaluation.

When WATCHDOG is chosen, the rule evaluation is done periodically. It can be used to optimize the rules processing. The evaluation period is set using the 'fireAllRulePeriod' policy.

Warning: Using the WATCHDOG policy will prevent the rule engine from firing the rules when some transient changes occur on objects in Working Memory. For instance, in CLOUD mode, a transient change of the 'aboutToBeRetracted' alarm attribute from 'false' to 'true' will be ignored and will not trigger the rules evaluation.

4.1.1.3 fireAllRulesDuringResynchronization

This policy is a boolean policy. Its value can either be "true" or "false".

Resynchronization is the phase where the Channel Adapter resends all alarms (except "closed" alarms) from its backing store. This phase comes when the Channel Adapter collection is established or upon operator's specific request.

During the re-synchronization an important alarm flow is received within a short period of time which leads to a great number of rule evaluations, implying a potential impact on performance.

The fireAllRulesDuringResynchronization policy - when set to false – prevents rule evaluation during resynchronization.

A single 'fireAllRule' is performed when the resynchronization terminates.

Note: When this policy is not defined in the XML file, **the policy is false by default**: the rules will not be triggered during a resynchronization.

4.1.1.4 retractOnResyncPolicy

This policy can take three possible values: NONE, PER_FLOW or ALL.

As stated above, resynchronization is the phase where the Channel Adapter resends all alarms (except "closed" alarms) from its backing store. This phase comes when the Channel Adapter collection is established or upon operator's specific request.

To have a consistent state in Working Memory, it is usually a good idea to retract the alarms from Working Memory prior to resynchronizing a Mediation Flow. This prevents alarms from potentially ending up being duplicated in Working Memory (in STREAM mode for example), rules not being fired as expected (update rules being triggered instead of insert rules in CLOUD mode for example), and basically not having a predictable state of the Working Memory. If you don't want to write rules that will specifically deal with the resynchronization, it is usually advisable to enable the retractOnResyncPolicy (by setting it to a value other than NONE).

The following values are possible for the retractOnResyncPolicy:

- **NONE:** No object is retracted from Working Memory upon resynchronization of a Mediation Flow
- **PER_FLOW:** All Alarm(1) objects associated with the resynchronizing Mediation Flow (as indicated by the targetValuePack(2) property of the Alarm(1) objects) are retracted from Working Memory upon resynchronization of a Mediation Flow (this is the default value if the retractOnResyncPolicy property is omitted)

- **ALL:** All objects are retracted from Working Memory upon resynchronization of a Mediation Flow

Note: When this policy is not defined in the XML file, **the policy is set to PER_FLOW by default**: when a resynchronization is initiated on a Mediation Flow, all alarms associated with this specific Mediation Flow will be retracted from Working Memory.

Notes

(1) Alarm objects are all objects that implement the AlarmCommon interface (either directly or indirectly by extending an object that implements this interface: see Chapter 3.5.6 Alarm Enrichment), i.e.:

- Alarm (and derived objects)
- AlarmAttributeValueChange (and derived objects)
- AlarmDeletion (and derived objects)
- AlarmStateChange (and derived objects)
- Any object that implements the AlarmCommon interface or extends an object that does

The AlarmCommon interface, and the Alarm, AlarmAttributeValueChange, AlarmStateChange, and AlarmDeletion classes are located in the com.hp.uca.expert.alarm package.

(2) The format of the "TargetValuePack" property is the following: <value pack name>-<value pack version>##<mediation flow name>, for example: pd-example-2.1##temipFlow.

4.1.1.5 compressionMode

This policy is a Boolean policy. Its value can either be "true" or "false". By default **compressionMode** is set to "false", i.e. compression is disabled.

When set to true, compression is enabled. When this is the case, alarm Attribute Value Change and alarm State Change events are compressed before being sent to the scenario.

Alarm Attribute Value Change events that target the same Alarm (identified by the identifier field) over a period of time (called compressionPeriod) are grouped together and replaced by a single Alarm Attribute Value Change event that contains the aggregated attribute changes contained in all the Alarm Attribute Value Change events. The same mechanism applies to alarm State Change events.

Compression limits the number of Alarm Attribute Value Change and State Change events being sent to the Scenario, thus improving the overall performance of the Drools engine of the Scenario because fewer events need to be processed.

It is interesting to enable the compression policy when a lot of Alarm Attribute Value Change and State Change events on a small number of distinct Alarms are expected to be received by a Scenario.

When the compression policy is enabled, you can also set the **compressionPeriod** property that defines the period of time over which to compress Alarm Attribute Value Change and State Change events. The default value for the compressionPeriod is set to 1000 milliseconds (1 second).

Below is an example of how to set the **compressionMode** and **compressionPeriod** properties for a Scenario in the `ValuePackConfiguration.xml` file of a Value Pack:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<valuePackConfiguration
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     name="myValuePack"
     version="1.0">

     <scenarios>
        <scenario name="com.acme.dummy.Scenario1">

<actionRetractedAutomaticallyWhenCompleted>true</actionRetractedAutomaticallyWhenCompleted>
          <asyncActionPeriod>1000</asyncActionPeriod>
          <clockTypeMode>NORMAL</clockTypeMode>
          <filterFile>deploy/myValuePack-1.0/scenario1/myFilters.xml</filterFile>
          <fireAllRulePeriod>1000</fireAllRulePeriod>
          <fireAllRulesDuringResynchronization>true</fireAllRulesDuringResynchronization>
          <fireAllRulesPolicy>WATCHDOG</fireAllRulesPolicy>
          <globals></globals>
          <processingMode>CLOUD</processingMode>
          <rulesFiles>
              <rulesFile>
                    <filename>file:./deploy/myValuePack-1.0/scenario1/myRules.drl</filename>
                    <name>My Rules</name>
                    <ruleFileType>DRL</ruleFileType>
              </rulesFile>
          </rulesFiles>
          <tickPeriod>30000</tickPeriod>
          <compressionMode>true</compressionMode>
          <compressionPeriod>1000</compressionPeriod><!-- in milliseconds -->
       </scenario>
    </scenarios>
</valuePackConfiguration>
```

XML Configuration 7 – Enabling the CompressionMode for a Scenario

4.1.2 Time-related settings

Note: In this section, all timer values are defined in **milliseconds**.

4.1.2.1 fireAllRulePeriod

This timer configures the fireAllRules period. This is the time interval between two executions of the evaluation of the rules, in case the “WATCHDOG” fireAllRulePolicy is selected.

Default value: 1000 (milliseconds), so 1 second.

4.1.2.2 tickPeriod

This property configures the tickFlag timer period. This is the time interval between two updates of the TickFlag object present in WorkingMemory.

This object can be used in rule conditions to implement recurrent rule executions.

Default value: 30000 (milliseconds), so 30 seconds.

4.1.2.3 asyncActionPeriod

This timer configures the asynchronous action management period. This is the time between two calls of the Asynchronous Action Management agent. The Asynchronous Action Management agent calls the Asynchronous Actions callbacks (see Chapter 5.3.3 Action callback) and updates the Asynchronous Actions status in Working Memory.

Default value: 1000 (milliseconds), so 1 second.

4.1.2.4 garbageCollectionPeriod

This timer configures the garbage collection period. This is the time interval between two garbage collections. The garbage collection will retract both:

Completed actions (if the actionRetractedAutomaticallyWhenCompleted Policy is set to “true”)

Alarms with a ‘aboutToBeRetracted’ attribute equals to ‘true’.

Default value: 10000 (milliseconds), so 10 seconds.

4.1.3 Alarm eligibility

The following alarm eligibility policies filter alarms coming into a scenario. They are used in addition to the scenario filter file described in section 2.4.1 “Filters”.

4.1.3.1 eligibleForBroadcast

This policy is a boolean policy. Its value can either be “true” or “false”.

When set to “true”, the scenario is able to receive incoming alarms from the mediation layer (then these alarms are filtered or not by the scenario filter).

When set to “false”, the scenario is not able to receive incoming alarms from the mediation layer. The scenario can only receive alarms from other scenarios (either in the same Value Pack or another Value Pack) using scenario cascading capabilities (See chapter 6.1.6 Orchestration API).

The default value for this *eligibleForBroadcast* property is “true”. By default, all scenarios can receive incoming alarms from the mediation layer.

4.1.3.2 alarmEligibilityPolicy

The alarmEligibilityPolicy is a Boolean expression. Its value is an expression that is evaluated and can either be “true” or “false”. This policy determines whether an Alarm is eligible or not to be inserted into Working Memory or to remain in Working Memory.

If this expression is evaluated to “true”, the alarm will be inserted in the scenario’s Working Memory. In the same way, if the expression results in a value of “false” the alarm will not be inserted or will be automatically retracted from the Working Memory.

A usual alarmEligibilityPolicy Expression can be based on the combination of the following alarm statuses: NetworkState, OperatorState, and ProblemState.

The possible values for a NetworkState are: "NOT_CLEARED", "CLEARED".

The possible values for an OperatorState are: "NOT_ACKNOWLEDGED", "ACKNOWLEDGED", "TERMINATED".

The possible values for a ProblemState are: "NOT_HANDLED", "HANDLED", "CLOSED".

The syntax of the alarmEligibilityPolicy expression is similar to a Java condition expression.

The following are examples of valid alarmEligibilityPolicy expressions:

NetworkState=="NOT_CLEARED"

NetworkState=="NOT_CLEARED" && OperatorState!="TERMINATED"

NetworkState=="NOT_CLEARED" && OperatorState!="TERMINATED" && ProblemState!="CLOSED"

NetworkState!="CLEARED" && (OperatorState=="TERMINATED" || OperatorState=="ACKNOWLEDGED") && ProblemState!="CLOSED"

Warning:

The alarmEligibility expression is specified within an XML file and must respect the XML grammar. This means that particular characters such as &, <, >, ", ' are not allowed and must be replaced by their XMl representation counterpart.

The expression : OperatorState!="TERMINATED"

Should then be represented as :

```
<alarmEligibilityPolicy>
        OperatorState!=&quot;TERMINATED&quot;
</alarmEligibilityPolicy>
```

Which is not really readable as soon as the expression becomes a little bit more complex.

Another way the set the expression in a more readable way is to use the XML CDATA Tag which allows using the special characters in the expression string as follow:

```
<alarmEligibilityPolicy><![CDATA[
            OperatorState!="TERMINATED"
]]></alarmEligibilityPolicy>
```

Default value: The default value for the alarmEligibilityPolicy expression is "true"; meaning that each alarm will be systematically inserted into Working Memory and will never be automatically retracted later on.

4.1.4 Garbage Collection

4.1.4.1 actionRetractedAutomaticallyWhenCompleted

This policy is a boolean policy. Its value can either be "true" or "false".

When an action completes, the object associated to the executed action remains in WorkingMemory with a status set to "Completed".

By setting the actionRetractedAutomaticallyWhenCompleted boolean policy to "true", the UCA for EBC framework will automatically retract any Action object that has a status of "Completed" from Working Memory. By setting this policy to "false", all Action objects will remain in Working Memory until they are explicitely removed (using a retract statement).

Default Value: true

4.1.5 Automatic handling of configuration files modifications

4.1.5.1 automaticRefreshOnConfigurationChange

This policy is a boolean policy. Its value can either be "true" or "false".

The Scenario Configuration files (filters, rule files, mapping files, specific configuration Files) are read at scenario startup. Such files can be modified from the GUI and a GUI button can be used to make the scenario re-read the file and take the new values into account.

There automaticRefreshOnConfigurationChange policy is a way to make the system automatically re-read the configuration files without any operation from the GUI.

This can be useful if the configuration files are automatically generated from an external application.

When setting the automaticRefreshOnConfigurationChange to ‘true’ the configurations files are periodically checked and reloaded if detected as changed.

Default Value: false

Chapter 5

Common Objects

This chapter defines the common Java objects provided by the UCA for EBC framework and gives information on how to use these objects in the rules.

☞ Refer to [R3] *Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Engine* for detailed usage of the API.

5.1 Events

Event class was introduced in V3.1. It is a generic class to define all incoming objects within UCA-EBC. More specifically, Alarm class is a specialization of Event class.

In the rules, the following imports can be used to deal with event content:

```
import com.hp.uca.expert.event.Event;
```

Event attributes can be used in the rules "condition" part, and Event methods can be called in the rules "action" part.

5.1.1 Identification attributes

Attribute	Method	Type
identifier	getIdentifier()	String
sourceIdentifier	getSourceIdentifier()	String

Table 9 - Event identification attributes and methods

5.1.2 Orchestration attributes

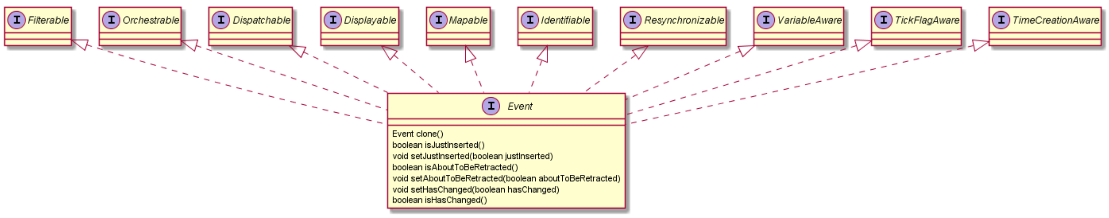
Attribute	Method	Type
orchestraData	getOrchestraData()	Map<String, Serializable>
convergenceComplete	isConvergenceComplete()	boolean
eventUUID	getEventUUID()	UUID

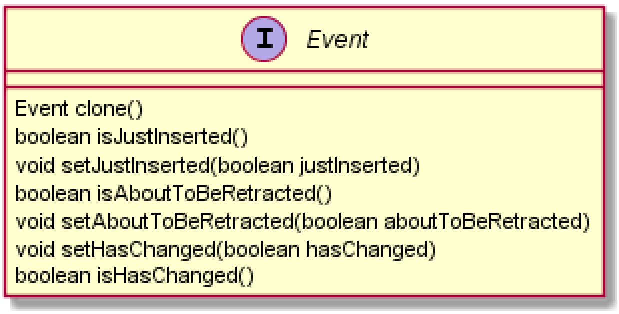
Table 10 - Event orchestration attributes and methods

Note

The Orchestration attributes and methods are detailed in ☞ 6.1.6 Orchestration API.

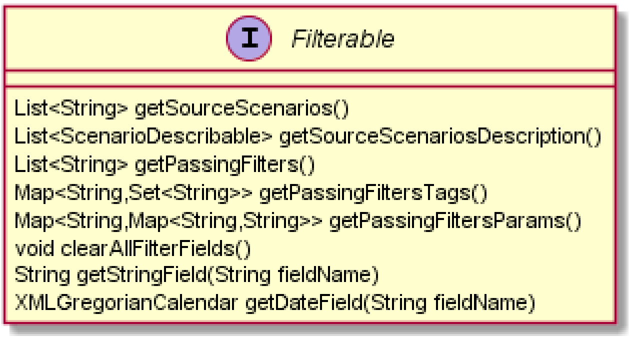
5.1.3 Interfaces hierarchy



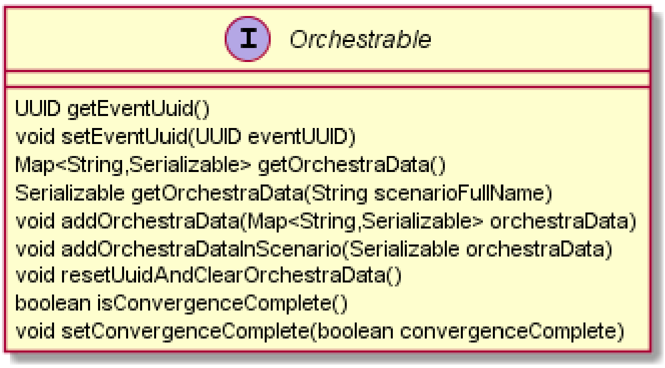


```
Event clone()
boolean isJustInserted()
void setJustInserted(boolean justInserted)
boolean isAboutToBeRetracted()
void setAboutToBeRetracted(boolean aboutToBeRetracted)
void setHasChanged(boolean hasChanged)
boolean isHasChanged()
```

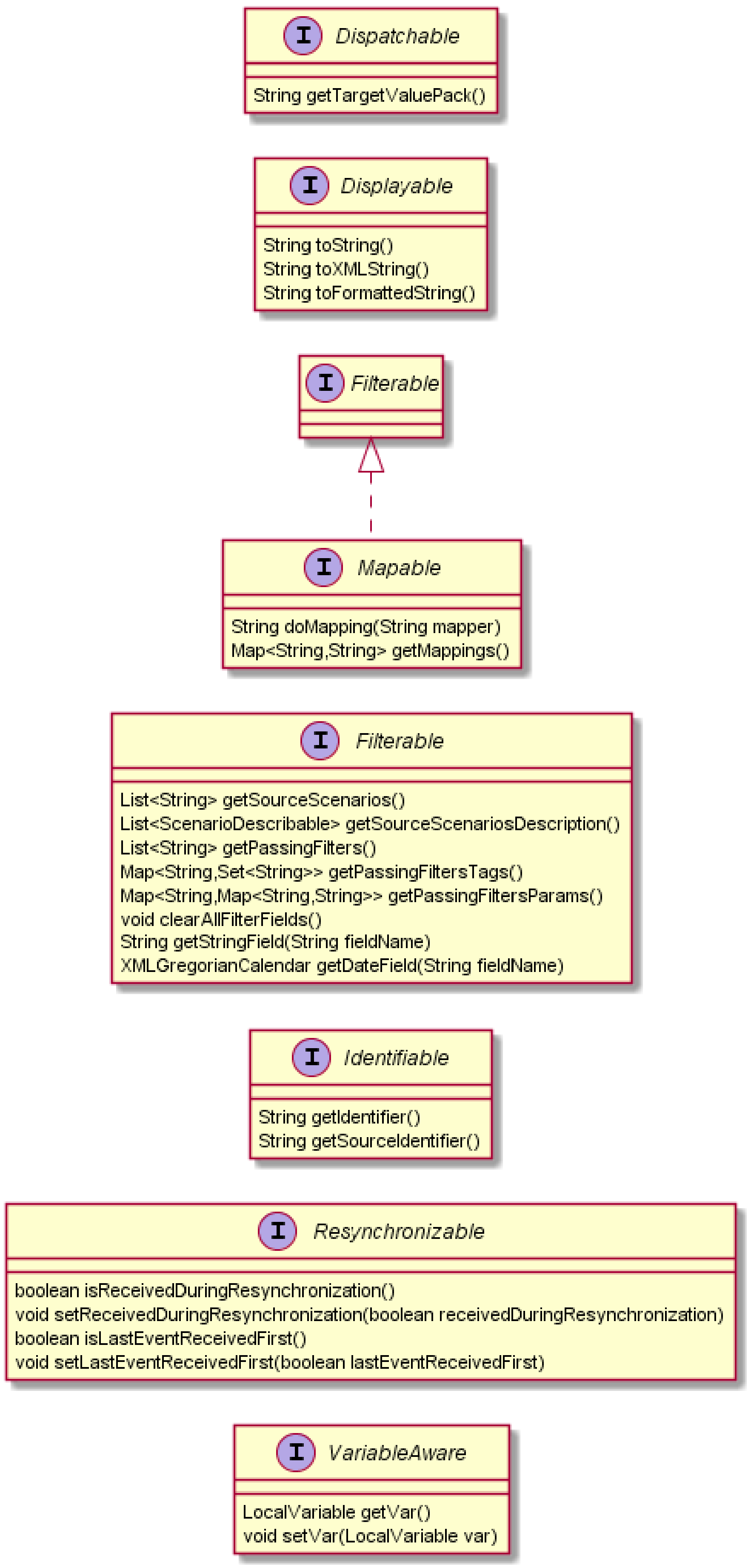
With following super interfaces:



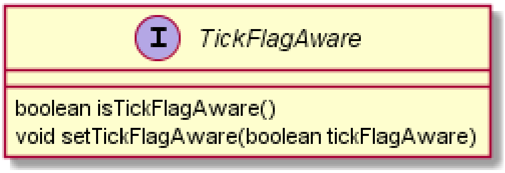
```
List<String> getSourceScenarios()
List<ScenarioDescribable> getSourceScenariosDescription()
List<String> getPassingFilters()
Map<String,Set<String>> getPassingFiltersTags()
Map<String,Map<String,String>> getPassingFiltersParams()
void clearAllFilterFields()
String getStringField(String fieldName)
XMLGregorianCalendar getDateField(String fieldName)
```

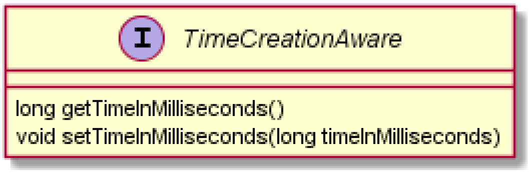


```
UUID getEventUuid()
void setEventUuid(UUID eventUUID)
Map<String,Serializable> getOrchestraData()
Serializable getOrchestraData(String scenarioFullName)
void addOrchestraData(Map<String,Serializable> orchestraData)
void addOrchestraDataInScenario(Serializable orchestraData)
void resetUuidAndClearOrchestraData()
boolean isConvergenceComplete()
void setConvergenceComplete(boolean convergenceComplete)
```



```
I Dispatchable
String getTargetValuePack()

I Displayable
String toString()
String toXMLString()
String toFormattedString()

I Filterable

I Mapable
String doMapping(String mapper)
Map<String,String> getMappings()

I Filterable
List<String> getSourceScenarios()
List<ScenarioDescribable> getSourceScenariosDescription()
List<String> getPassingFilters()
Map<String,Set<String>> getPassingFiltersTags()
Map<String,Map<String,String>> getPassingFiltersParams()
void clearAllFilterFields()
String getStringField(String fieldName)
XMLGregorianCalendar getDateField(String fieldName)

I Identifiable
String getIdentifier()
String getSourceIdentifier()

I Resynchronizable
boolean isReceivedDuringResynchronization()
void setReceivedDuringResynchronization(boolean receivedDuringResynchronization)
boolean isLastEventReceivedFirst()
void setLastEventReceivedFirst(boolean lastEventReceivedFirst)

I VariableAware
LocalVariable getVar()
void setVar(LocalVariable var)
```


I TickFlagAware
boolean isTickFlagAware()
void setTickFlagAware(boolean tickFlagAware)


I TimeCreationAware
long getTimeInMilliseconds()
void setTimeInMilliseconds(long timeInMilliseconds)

5.2 Alarms models used in the rules

In the rules, the following imports can be used to deal with alarm content:

```
import com.hp.uca.expert.event.Event;
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.alarm.AlarmStateChange;
import com.hp.uca.expert.alarm.AlarmAttributeValueChange;
import com.hp.uca.expert.alarm.AlarmDeletion;

import com.hp.uca.expert.x733alarm.PerceivedSeverity;
import com.hp.uca.expert.x733alarm.AlarmType;
import com.hp.uca.expert.x733alarm.CustomFields;
import com.hp.uca.expert.x733alarm.CustomField;
```

All flavors of Alarms objects inherit from Event class.

5.2.1 Alarm

```
import com.hp.uca.expert.alarm.Alarm;
```

Alarm attributes can be used in the rules “condition” part, and Alarm methods can be called in the rules “action” part.

5.2.1.1 General attributes

Attribute	Method	Type
originatingManagedEntity	getOriginatingManagedEntity()	String
originatingManagedEntityStructure	getOriginatingManagedEntityStructure()	OriginatingManagedEntityStructure
alarmType	getAlarmType()	AlarmType
perceivedSeverity	getPerceivedSeverity()	PerceivedSeverity
probableCause	getProbableCause()	String
alarmRaisedTime	getAlarmRaisedTime()	XMLGregorianCalendar
timeInMilliseconds	getTimeInMilliseconds()	long
specificProblem	getSpecificProblem()	String
additionalInformation	getAdditionalInformation()	String
additionalText	getAdditionalText()	String
customFields	getCustomFields()	CustomFields
	getCustomFieldValue(String field)	String

	setCustomFieldValue(String field, String value)	boolean
proposedRepairActions	getProposedRepairActions()	String
alarmAdditionalData	getAlarmAdditionalData()	String

Table 11 - Alarm attributes and methods

Note

The alarmRaisedTime field is automatically updated when using the setTimeInMilliSecond() method.

5.2.1.2 Status attributes

Attribute	Method	Type
networkState	getNetworkState()	NetworkState
operatorState	getOperatorState()	OperatorState
problemState	getProblemState()	ProblemState
problemInformation	getProblemInformation()	String

Table 12 - Alarm status attributes and methods

5.2.1.3 Correlation purpose attributes

Attribute	Method	Type
notificationIdentifier	getNotificationIdentifier()	String
correlationsNotificationIdentifiers	getCorrelationsNotificationIdentifiers()	String

Table 13 - Alarm Correlation attributes and methods

5.2.1.4 Association purpose attributes

Attribute	Method	Type
parents	getParents()	String
children	getChildren()	String

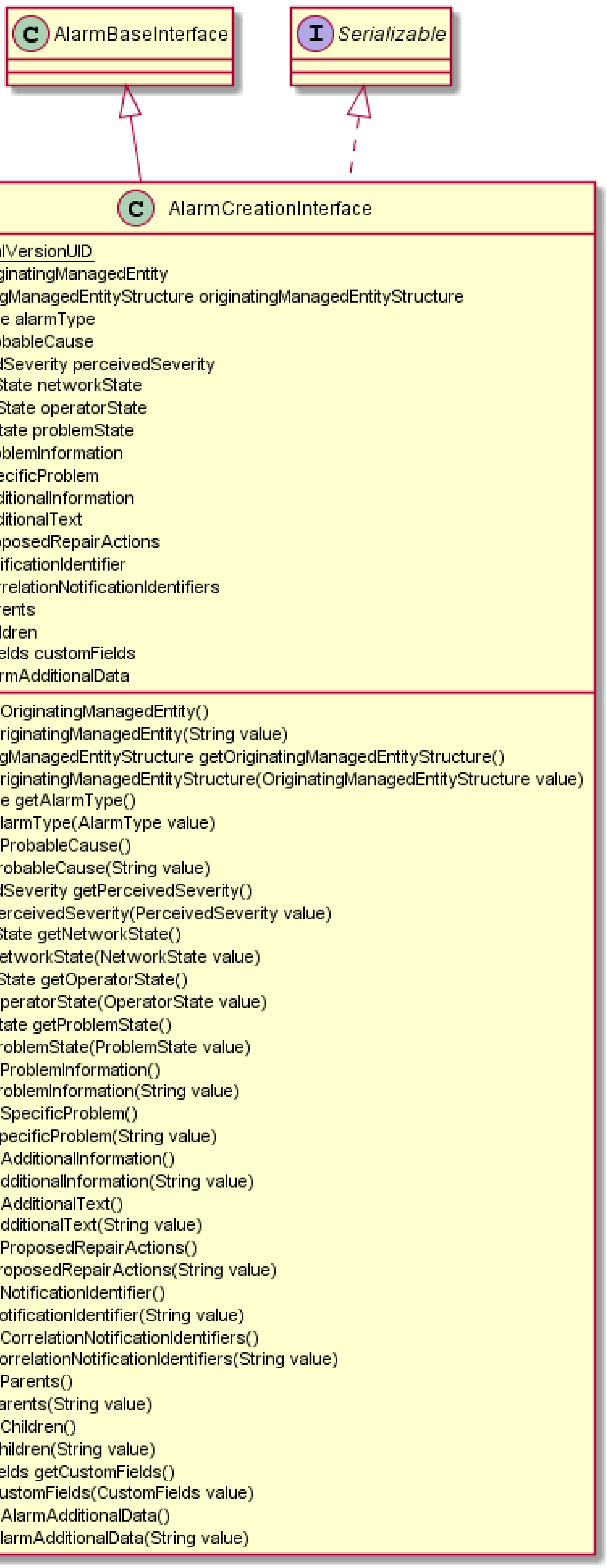
Table 14 - Alarm Association attributes and methods

5.2.1.5 Example

```
import com.hp.uca.expert.alarm.Alarm;
rule "Sample"
when
  a: Alarm(originatingManagedEntity == "BOX .b1")
then
  System.out.println(a.getOriginatingManagedEntity());
  a.setAdditionalText(a.getAdditionalText() + " Append user defined");
end
```

Rule Sample 6 - Alarm attributes

5.2.1.6 AlarmCreationInterface detailed



AlarmBaseInterface
Serializable
AlarmCreationInterface
long serialVersionUID
String originatingManagedEntity
OriginatingManagedEntityStructure originatingManagedEntityStructure
AlarmType alarmType
String probableCause
PerceivedSeverity perceivedSeverity
NetworkState networkState
OperatorState operatorState
ProblemState problemState
String problemInformation
String specificProblem
String additionalInformation
String additionalText
String proposedRepairActions
String notificationIdentifier
String correlationNotificationIdentifiers
String parents
String children
CustomFields customFields
String alarmAdditionalData
String getOriginatingManagedEntity()
void setOriginatingManagedEntity(String value)
OriginatingManagedEntityStructure getOriginatingManagedEntityStructure()
void setOriginatingManagedEntityStructure(OriginatingManagedEntityStructure value)
AlarmType getAlarmType()
void setAlarmType(AlarmType value)
String getProbableCause()
void setProbableCause(String value)
PerceivedSeverity getPerceivedSeverity()
void setPerceivedSeverity(PerceivedSeverity value)
NetworkState getNetworkState()
void setNetworkState(NetworkState value)
OperatorState getOperatorState()
void setOperatorState(OperatorState value)
ProblemState getProblemState()
void setProblemState(ProblemState value)
String getProblemInformation()
void setProblemInformation(String value)
String getSpecificProblem()
void setSpecificProblem(String value)
String getAdditionalInformation()
void setAdditionalInformation(String value)
String getAdditionalText()
void setAdditionalText(String value)
String getProposedRepairActions()
void setProposedRepairActions(String value)
String getNotificationIdentifier()
void setNotificationIdentifier(String value)
String getCorrelationNotificationIdentifiers()
void setCorrelationNotificationIdentifiers(String value)
String getParents()
void setParents(String value)
String getChildren()
void setChildren(String value)
CustomFields getCustomFields()
void setCustomFields(CustomFields value)
String getAlarmAdditionalData()
void setAlarmAdditionalData(String value)

5.2.2 AlarmStateChange

Import

```
import com.hp.uca.expert.alarm.AlarmStateChange;
```

Attributes and methods

AlarmStateChange attributes can be used in the rules “condition” part, and AlarmStateChange methods can be called in the rules “action” part:

5.2.2.1 Identification attributes

Attribute	Method	Type
Identifier	getIdentifier()	String

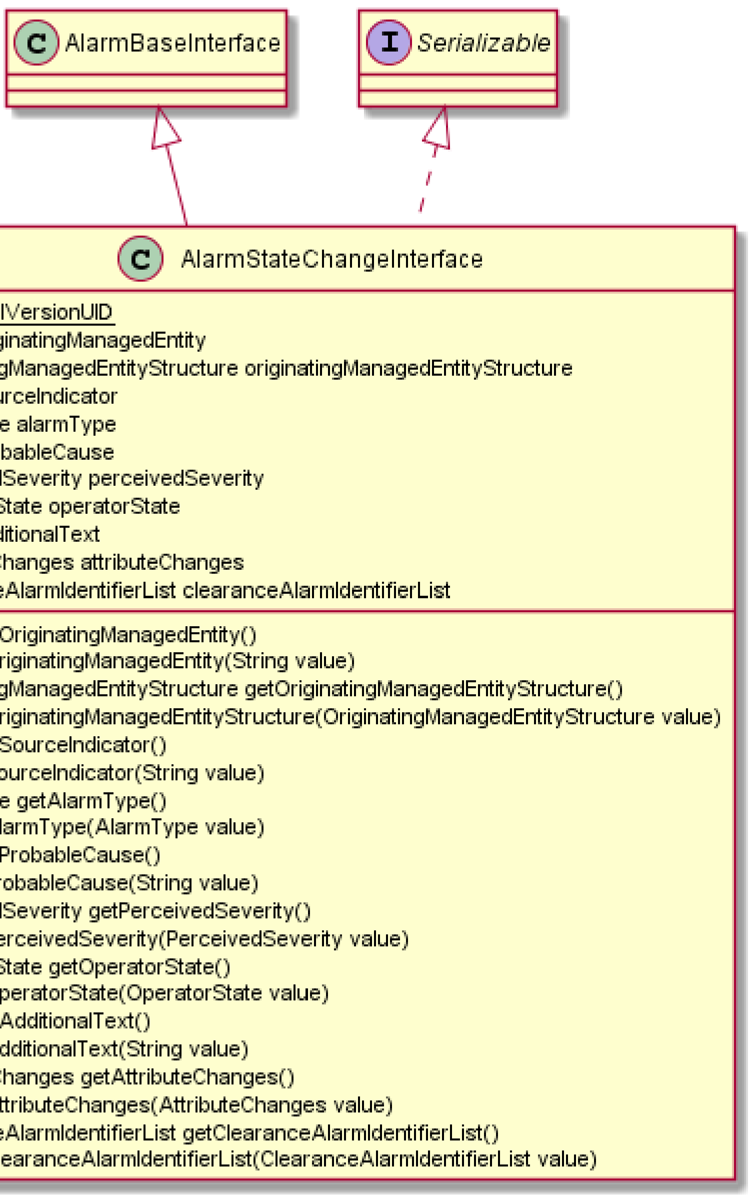
Table 15 - AlarmStateChange identifier attributes and methods

5.2.2.2 General attributes

Attribute	Method	Type
originatingManagedEntity	getOriginatingManagedEntity()	String
originatingManagedEntityStructure	getOriginatingManagedEntityStructure()	OriginatingManagedEntityStructure
sourceIndicator	getSourceIndicator()	String
additionalText	getAdditionalText()	String
attributeChanges	getAttributeChanges()	AttributeChanges

Table 16 - AlarmStateChange general attributes and methods

5.2.2.3 AlarmStateChangeInterface detailed


AlarmBaseInterface
Serializable
AlarmStateChangeInterface
long serialVersionUID
String originatingManagedEntity
OriginatingManagedEntityStructure originatingManagedEntityStructure
String sourceIndicator
AlarmType alarmType
String probableCause
PerceivedSeverity perceivedSeverity
OperatorState operatorState
String additionalText
AttributeChanges attributeChanges
ClearanceAlarmIdentifierList clearanceAlarmIdentifierList
String getOriginatingManagedEntity()
void setOriginatingManagedEntity(String value)
OriginatingManagedEntityStructure getOriginatingManagedEntityStructure()
void setOriginatingManagedEntityStructure(OriginatingManagedEntityStructure value)
String getSourceIndicator()
void setSourceIndicator(String value)
AlarmType getAlarmType()
void setAlarmType(AlarmType value)
String getProbableCause()
void setProbableCause(String value)
PerceivedSeverity getPerceivedSeverity()
void setPerceivedSeverity(PerceivedSeverity value)
OperatorState getOperatorState()
void setOperatorState(OperatorState value)
String getAdditionalText()
void setAdditionalText(String value)
AttributeChanges getAttributeChanges()
void setAttributeChanges(AttributeChanges value)
ClearanceAlarmIdentifierList getClearanceAlarmIdentifierList()
void setClearanceAlarmIdentifierList(ClearanceAlarmIdentifierList value)

5.2.3 AlarmAttributeValueChange

```
import com.hp.uca.expert.alarm.AlarmAttributeValueChange;
```

AlarmAttributeValueChange attributes can be used in the rules “condition” part, and AlarmAttributeValueChange methods can be called in the rules “action” part:

5.2.3.1 Identification attributes

Attribute	Method	Type
Identifier	getIdentifier()	String

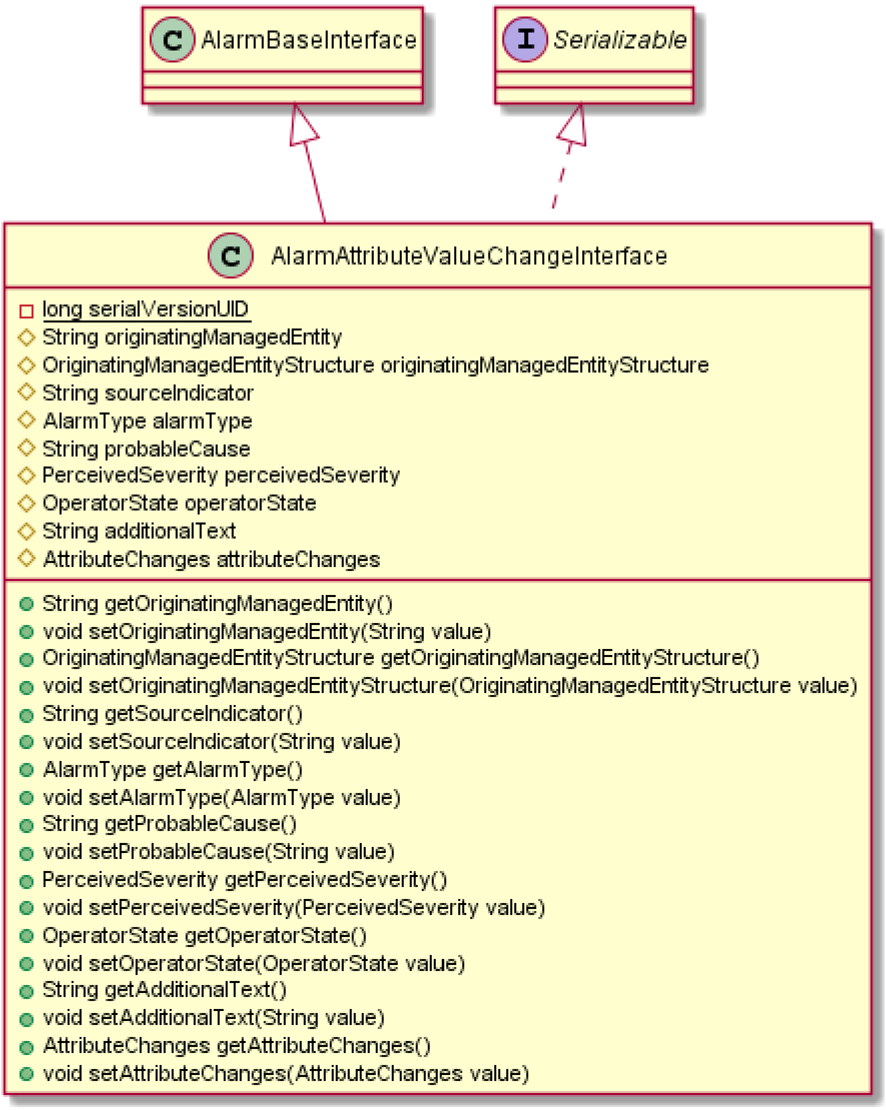
Table 17 - AlarmAttributeValueChange identifier attributes and methods

5.2.3.2 General attributes

Attribute	Method	Type
originatingManagedEntity	getOriginatingManagedEntity()	String
originatingManagedEntityStructure	getOriginatingManagedEntityStructure()	OriginatingManagedEntityStructure
sourceIndicator	getSourceIndicator()	String
additionalText	getAdditionalText()	String
attributeChanges	getAttributeChanges()	AttributeChanges

Table 18 - AlarmAttributeValueChange general attributes and methods

5.2.3.3 AlarmAttributeValueChangeInterface detailed



C AlarmBaseInterface
I Serializable
C AlarmAttributeValueChangeInterface
long serialVersionUID
String originatingManagedEntity
OriginatingManagedEntityStructure originatingManagedEntityStructure
String sourceIndicator
AlarmType alarmType
String probableCause
PerceivedSeverity perceivedSeverity
OperatorState operatorState
String additionalText
AttributeChanges attributeChanges
String getOriginatingManagedEntity()
void setOriginatingManagedEntity(String value)
OriginatingManagedEntityStructure getOriginatingManagedEntityStructure()
void setOriginatingManagedEntityStructure(OriginatingManagedEntityStructure value)
String getSourceIndicator()
void setSourceIndicator(String value)
AlarmType getAlarmType()
void setAlarmType(AlarmType value)
String getProbableCause()
void setProbableCause(String value)
PerceivedSeverity getPerceivedSeverity()
void setPerceivedSeverity(PerceivedSeverity value)
OperatorState getOperatorState()
void setOperatorState(OperatorState value)
String getAdditionalText()
void setAdditionalText(String value)
AttributeChanges getAttributeChanges()
void setAttributeChanges(AttributeChanges value)

5.2.4 AlarmDeletion

```
import com.hp.uca.expert.alarm.AlarmDeletion;
```

AlarmDeletion attributes can be used in the rules “condition” part, and AlarmDeletion methods can be called in the rules “action” part:

5.2.4.1 Identification attributes

Attribute	Method	Type
identifier	getIdentifier()	String

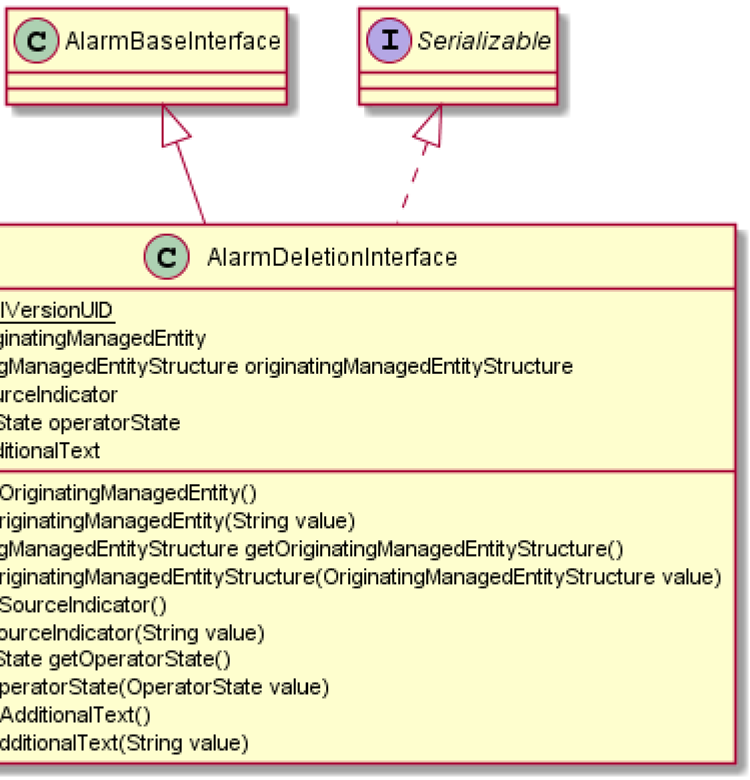
Table 19 - AlarmDeletion identifier attributes and methods

5.2.4.2 General attributes

Attribute	Method	Type
originatingManagedEntity	getOriginatingManagedEntity()	String
originatingManagedEntityStructure	getOriginatingManagedEntityStructure()	OriginatingManagedEntityStructure
sourceIndicator	getSourceIndicator()	String
additionalText	getAdditionalText()	String

Table 20 - AlarmDeletion general attributes and methods

5.2.4.3 AlarmDeletionInterface detailed



AlarmBaseInterface
Serializable
AlarmDeletionInterface
long serialVersionUID
String originatingManagedEntity
OriginatingManagedEntityStructure originatingManagedEntityStructure
String sourceIndicator
OperatorState operatorState
String additionalText
String getOriginatingManagedEntity()
void setOriginatingManagedEntity(String value)
OriginatingManagedEntityStructure getOriginatingManagedEntityStructure()
void setOriginatingManagedEntityStructure(OriginatingManagedEntityStructure value)
String getSourceIndicator()
void setSourceIndicator(String value)
OperatorState getOperatorState()
void setOperatorState(OperatorState value)
String getAdditionalText()
void setAdditionalText(String value)

5.2.5 Properties of alarms

5.2.5.1 AlarmType

Import

```
import com.hp.uca.expert.x733alarm.AlarmType;
```

Values

The following table shows the possible values for the AlarmType attribute:

AlarmType values
AlarmType.UNKNOWN_ALARM_TYPE
AlarmType.COMMUNICATIONS_ALARM
AlarmType.PROCESSING_ERROR_ALARM
AlarmType.ENVIRONMENTAL_ALARM
AlarmType.QUALITY_OF_SERVICE_ALARM
AlarmType.EQUIPMENT_ALARM
AlarmType.INTEGRITY_VIOLATION
AlarmType.OPERATIONAL_VIOLATION
AlarmType.PHYSICAL_VIOLATION
AlarmType.SECURITY_VIOLATION
AlarmType.TIME_DOMAIN_VIOLATION

Table 21 - AlarmType values

Example

Below is an example of how to use the AlarmType attribute in rules:

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.AlarmType;

rule "Sample"
when
  a: Alarm(originatingManagedEntity == "BOX .b1" &&
   alarmType==AlarmType.EQUIPMENT_ALARM)
then
  // Do Something
end
```

Rule Sample 7 - AlarmType example

5.2.5.2 PerceivedSeverity

Import

```
import com.hp.uca.expert.x733alarm.PerceivedSeverity;
```

Values

The following table shows the possible values for the PerceivedSeverity attribute:

PerceivedSeverity values
PerceivedSeverity.CLEAR
PerceivedSeverity.CRITICAL
PerceivedSeverity.INDETERMINATE
PerceivedSeverity.MAJOR
PerceivedSeverity.MINOR
PerceivedSeverity.WARNING

Table 22 - PerceivedSeverity values

Example

Below is an example of how to use the PerceivedSeverity attribute in rules:

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.PerceivedSeverity;

rule "Sample"
when
   a: Alarm(perceivedSeverity == PerceivedSeverity.CRITICAL )
then
   // Do Something
end
```

Rule Sample 8 - PerceivedSeverity example

5.2.5.3 NetworkState

Import

```
import com.hp.uca.expert.x733alarm.NetworkState;
```

Values

The following table shows the possible values for the NetworkState attribute:

NetworkState values
NetworkState.NOT_CLEARED
NetworkState.CLEARED

Table 23 - NetworkState values

Example

Below is an example of how to use the NetworkState attribute in rules:

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.NetworkState;

rule "Sample"
when
   a: Alarm(networkState == NetworkState.NOT_CLEARED )
then
   // Do Something
end
```

Rule Sample 9 - NetworkState example

5.2.5.4 OperatorState

Import

```
import com.hp.uca.expert.x733alarm.OperatorState;
```

Values

The following table shows the possible values for the OperatorState attribute:

OperatorState values
OperatorState.NOT_ACKNOWLEDGED
OperatorState.ACKNOWLEDGED
OperatorState.TERMINATED

Table 24 - OperatorState values

Example

Below is an example of how to use the OperatorState attribute in rules:

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.OperatorState;

rule "Sample"
when
   a: Alarm(operatorState == OperatorState.NOT_CLEARED )
then
   // Do Something
end
```

Rule Sample 10 - OperatorState example

5.2.5.5 ProblemState

Import

```
import com.hp.uca.expert.x733alarm.ProblemState;
```

Values

The following table shows the possible values for the ProblemState attribute:

ProblemState values
ProblemState.NOT_HANDLED
ProblemState.HANDLED
ProblemState.CLOSED

Table 25 - ProblemState values

Example

Below is an example of how to use the ProblemState attribute in rules:

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.ProblemState;

rule "Sample"
when
  a: Alarm(problemState == ProblemState.NOT_HANDLED )
then
  // Do Something
end
```

Rule Sample 11 - ProblemState example

5.2.5.6 CustomFields

Import

```
import com.hp.uca.expert.x733alarm.CustomFields;
import com.hp.uca.expert.x733alarm.CustomField;
```

CustomFields attributes and methods

CustomFields attributes can be used in the rules “condition” part, and CustomFields methods can be called in the rules “action” part:

Attribute	Method	Type
customField	getCustomField()	List<CustomField>

Table 26 - CustomFields attributes and methods

There is no “set” method on “CustomField”, but the list of custom fields can be modified in the rules “action” part.

CustomField attributes and methods

CustomField attributes can be used in the rules “condition” part, and CustomField methods can be called in the rules “action” part:

Attribute	Method	Type
name	getName() / setName()	String
value	getValue() / setValue()	String

Table 27 - CustomField attributes and methods

Alternative method with Alarm Object

The Alarm object also supports the following methods that are useful to get or set a CustomField:

- `String getCustomFieldValue(String fieldName)`
- `boolean setCustomFieldValue(String fieldName, String value)`

Example

```
import com.hp.uca.expert.x733alarm.CustomFields;
import com.hp.uca.expert.x733alarm.CustomField;
import com.hp.uca.expert.alarm.AlarmObjectFactory;

rule "Sample"
```

```
when
  a: Alarm(originatingManagedEntity == "BOX .b1")
then

 // Usage 1
 CustomField newCF = new AlarmObjectFactory().createCustomField();
 newCF.setName("Origin");
 newCF.setValue("UCA EBC system");
 a.getCustomField().add(newCF);

 // Alternatively, Usage 2
 a.setCustomFieldValue("Origin", "UCA EBC system");
end
```

Rule Sample 12 - CustomField example

5.2.5.7 AttributeChanges

Import

```
import com.hp.uca.expert.x733alarm.AttributeChanges;
import com.hp.uca.expert.x733alarm.AttributeChange;
```

AttributeChanges attributes and methods

AttributesChanges attributes can be used in the rules “condition” part, and AttributesChanges methods can be called in the rules “action” part:

Attribute	Method	Type
attributeChange	getAttributeChange()	List<AttributeChange>

Table 28 - AttributeChanges attributes and methods

There is no “set” method on “attributeChange”, but the list can be modified in the rules “action” part.

AttributeChange attributes and methods

AttributeChange attributes can be used in the rules “condition” part, and AttributeChange methods can be called in the rules “action” part:

Attribute	Method	Type
name	getName() / setName()	String
newValue	getNewValue() / setNewValue()	String
oldValue	getOldValue() / setOldValue ()	String

Table 29 - AttributeChange attributes and methods

5.3 Actions

UCA for EBC Actions are a means to trigger actions on external systems by going through the mediation layer: OSS Open Mediation V7.1 (NOM V7.1).

Such actions can be

Alarm Object actions routed to and executed by a Network Management System (alarm creation, alarm termination, alarm grouping etc…)

Script execution (execution of executables or command-line scripts) on the system hosting the mediation layer

Actions are routed to the mediation using the information stored in the *`ActionRegistry.xml`* file.

When new Action Java objects are created in the rules, the constructor method for the Action Java object must reference a valid actionReference defined in the *`ActionRegistry.xml`* file. ActionReferences define how to route actions to the mediation layer, and to the proper Channel Adapter (the Channel Adapter able to process the action request) on the mediation layer.

5.3.1 Action Class

☞ Please refer to [R4] *Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Actions* for more information on Actions.

Import

```
import com.hp.uca.mediation.action.client.Action;
```

Description

The proper way to create and execute actions is to perform the following 4 steps in sequence:

Create the action object using the `Action(String actionReference)` constructor.

The `actionReference` parameter references an `<Action actionReference="…">…</Action>` XML entity in the *`ActionRegistry.xml`* file that contains information for routing actions to the proper Channel Adapter on the mediation layer depending on the type of action (identified by the actionReference).

Please refer to chapter 5.3.2 “Action registry” for more information on how to define `actionReferences` that will target specific Channel Adapters on the mediation layer.

Here’s an example of how to create an Action object:

```
Action action = new Action("TeMIP_AO_Directives_localhost");
```

Specify what action to execute by using the `addCommand(String key, String value)` method.

This method defines what action to execute by defining key/value pairs that will be interpreted by the proper Channel Adapter on the mediation layer.

Here’s an example of how to define an Action object:

```
action.addCommand("directiveName", "TERMINATE");
action.addCommand("entityName", a.getIdentifier());
action.addCommand("UserId", "UCA for EBC");
```

Associate the newly created Action object to the current Scenario using the `addAction(Action action)` method.

Failure to associate the Action object with the current Scenario will result in the action being improperly processed.

Here’s an example of how to associate an Action object to the current scenario:

```
theScenario.addAction(action);
```

Request the execution of the action by using either the `executeSync()` method or the `executeAsync(String synchronizationKey)` method.

Here’s an example of how to request synchronous execution of an Action object:

```
action.executeSync();
```

Here’s an example of how to request asynchronous execution of an Action object:

```
action.executeAsync(Action.NO_SYNCHRONIZATION_KEY);
```

Example

The following chapters show how to use an action in a UCA for EBC rule for terminating an alarm. Actions can be executed either synchronously using the `executeSync()` method or asynchronously using the `executeAsync(String synchronizationKey)` method.

The difference between synchronous and asynchronous actions is that synchronous action requests are blocking while asynchronous action requests are not.

Notes

Actions have a default timeout defined in the `${UCA_EBC_INSTANCE}/conf/uca-ebc.properties` file. The default timeout is specified by the `action.timeout` property. The value of this property is in milliseconds. For example:

action.timeout = 60000

In case you need to, this default action timeout can be either changed in the `${UCA_EBC_INSTANCE}/conf/uca-ebc.properties` file or overwritten for any single action by using the `public void setActionTimeout(int actionTimeout)` method of any `Action` object. The `actionTimeout` parameter is also in milliseconds. For example:

```
// Sets this action timeout to 10 seconds
action.setActionTimeout(10000);
```

When the timeout is reached, the action fails. The status explanation of the action indicates that the action has timed out.

5.3.1.1 Synchronous actions

```
rule "Any Acknowledged Alarm with Add text request for termination
(Action)"
when
    a: Alarm(operatorState == OperatorState.ACKNOWLEDGED &&
additionalText matches "to terminate")
then
```

```
    theScenario.getLogger().info("[RULE " + drools.getRule().getName()
+ "] Found acknowledged alarm: identifier = " + a.getIdentifier() +
":");
    theScenario.getLogger().debug(a.toFormattedString());

    // Terminating the Alarm
    Action action = new Action("TeMIP_AO_Directives_localhost");
    action.addCommand("directiveName", "TERMINATE");
    action.addCommand("entityName", a.getIdentifier());
    action.addCommand("UserId", "UCA for EBC");
    theScenario.addAction(action); // Associate the action with the
scenario

    theScenario.getLogger().info("Executing synchronous TERMINATE
directive on alarm: " + a.getIdentifier());

    action.executeSync(); // This call is blocking. The execution of
the rule will continue only after the action has been executed.

    theScenario.getLogger().debug(action);
    if (action.getActionStatus() ==
com.hp.uca.mediation.action.client.ActionStatus.Completed)
    {
      theScenario.getLogger().info("Action successful");
    } else {
      theScenario.getLogger().error("Action failed:
"+action.getActionStatusExplanation());
    }
end
```

Rule Sample 13 - Simple Action example (synchronous)

When a call to `executeSync()` is made, the execution of the rule is stopped until the synchronous action terminates (either successfully or not).

5.3.1.2 Asynchronous actions

```
rule "Any Acknowledged Alarm with Add text request for termination
(Action)"
  when
    a: Alarm(operatorState == OperatorState.ACKNOWLEDGED &&
additionalText matches "to terminate")
  then
    theScenario.getLogger().info("[RULE " + drools.getRule().getName()
+ "] Found acknowledged alarm: identifier = " + a.getIdentifier() +
":");
    theScenario.getLogger().debug(a.toFormattedString());

    // Terminating the Alarm
    Action action = new Action("TeMIP_AO_Directives_localhost");
    action.addCommand("directiveName", "TERMINATE");
    action.addCommand("entityName", a.getIdentifier());
    action.addCommand("UserId", "UCA for EBC");
    theScenario.addAction(action); // Associate the action with the
scenario

    theScenario.getLogger().info("Executing synchronous TERMINATE
directive on alarm: " + a.getIdentifier());
```

```
    action.executeAsync(Action.NO_SYNCHRONIZATION_KEY); // This call is
non-blocking. The execution of the rule will continue right away and
not wait for the action to be executed

end
```

Rule Sample 14 - Simple Action example (asynchronous)

When a call to `executeAsync(String synchronizationKey)` is made, the execution or the rule continues right way. Rule execution is not stopped.

The executeAsync(String synchronizationKey) method has a mandatory parameter called synchronizationKey.

The purpose of this parameter is to preserve the order of groups of asynchronous actions that have the same `synchronizationKey` value. Because asynchronous actions are executed in parallel by a pool of threads, the order is thus not preserved. For example, asynchronous action A requested before asynchronous action B can end up being executed after asynchronous action B. This can be a problem in some use cases, when you need groups of asynchronous actions affecting the same object (an Alarm for example) to be executed in the order they are requested. In such a case, you could use, for example, the identifier of the alarm as a `synchronizationKey` for all asynchronous actions related to alarms.

The `synchronizationKey` must match the name of a key you have used in an `addCommand(String key, String value)` call for the same Action object.

For example, in order to use the identifier of the alarm as `synchronizationKey`, you have to write:

```
import com.hp.uca.mediation.action.client.Action;
import com.hp.uca.temip.mvp.aodirective.mapper.AODirective;
import com.hp.uca.temip.mvp.aodirective.mapper.AODirectiveKey;
…
#declare any global variables here
global Scenario theScenario;
…
rule "Rule 1"
  when
    a: Alarm(…)
  then
    …

    // Setting the Alarm Additional Text
    Action action = new Action("TeMIP_AO_Directives_localhost");
    action.addCommand(AODirectiveKey.DIRECTIVE_NAME, AODirective.SET);

    action.addCommand(AODirectiveKey.ENTITY_NAME, a.getIdentifier());
    action.addCommand(AODirectiveKey. ADDITIONAL_TEXT, "New Text");
    theScenario.addAction(action); // Associate the action with the
scenario
    // Execute the directive asynchronously, using the entity name as
Synchronization Key

    action.executeAsync(AODirectiveKey.ENTITY_NAME);
  …
End
```

```
rule "Rule 2"
  when
    a: Alarm(…)
  then
    …

    // Terminating the Alarm
    Action action = new Action("TeMIP_AO_Directives_localhost");
    action.addCommand(AODirectiveKey.DIRECTIVE_NAME,
AODirective.TERMINATE);
    action.addCommand(AODirectiveKey.ENTITY_NAME, a.getIdentifier());
    action.addCommand(AODirectiveKey.USERID, "UCA for EBC");
    theScenario.addAction(action); // Associate the action with the
scenario
    // Execute the directive asynchronously, using the entity name as
Synchronization Key
    action.executeAsync(AODirectiveKey.ENTITY_NAME);
    …
end
```

In the above example, the SET and TERMINATE directives are both executed asynchronously. In both cases, the entity name (alarm identifier) is used as the synchronization key. You can notice that the `synchronizationKey` matches the name of a key (`AODirectiveKey.ENTITY_NAME`) you have used in an `addCommand(String key, String value)` call for the same Action object.

If both SET and TERMINATE directives are executed on the same alarm object (i.e. same alarm identifier), the directives will be executed in the order they are requested thanks to the synchronization key.

If you don't want to use any synchronization key, please use an empty string or the `Action.NO_SYNCHRONIZATION_KEY` constant:

```
    …

    // Execute the directive asynchronously, using no Synchronization
Key
    action.executeAsync(Action.NO_SYNCHRONIZATION_KEY);
```

You can execute custom code when the asynchronous action terminates (either successfully or not) by using action callbacks as described in chapter 5.3.3 “Action callbacks”.

5.3.2 Action registry

The Action Registry is a configuration file used to define routing information for any Action processed by the rules.

This file is located at: ${UCA_EBC_INSTANCE}/conf/ActionRegistry.xml

The <**MediationValuePack**> tags of the Action Registry define a list of services called <Action>.

Each of these <Action> entries defines one action reference that can be used in the rules. Each action reference defines a specific “action service” provided by the mediation layer.

The file is a **sequence** of XML tags (the order is important).

The <**MediationValuePack**> tag defines the identification of a target Channel Adapter (the Channel Adapter that will be targeted by the actions).

Type	Name	Value
Attribute	MvpName	Channel Adapter name
Attribute	MvpVersion	Channel Adapter version
Attribute	url	URL of the Open Mediation entry point. For UCA for EBC, the entry point is the UCA for EBC Channel Adapter. This URL is the URL of the Action web service of the UCA for EBC Channel Adapter, used for processing UCA for EBC external actions.
Attribute	brokerURL	URL of the Open Mediation entry point. For UCA for EBC, the entry point is the UCA for EBC Channel Adapter. This URL is the URL of the JMS Broker service of the UCA for EBC Channel Adapter, used for forwarding Alarms to OSS Open Mediation V7.1.
Property	Action	List of Channel Adapter services

Table 30 - ActionRegistry - MediationValuePack properties

Each <**Action**> tag defines a separate action service implemented by the target Channel Adapter defined by the <**MediationValuePack**> tag:

Type	Name	Value
Attribute	actionReference	Unique reference that will be used in the rule to define the routing information of an Action
Property	ServiceName	Identifier of the service supported by the Channel Adapter. This information is available in the Channel Adapter specification guide.
Property	NmsName	The identifier of the NMS that will receive the Action. This information can be used by the Channel Adapter to know on which physical machine to send the action.

Table 31 - ActionRegistry – Action properties

Example

Here’s an example of the ActionRegistry.xml file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ActionRegistryXML xmlns="http://registry.action.mediation.uca.hp.com/">

        <MediationValuePack MvpName="temip"
                            MvpVersion="1.0"

        url=http://localhost:26700/uca/mediation/action/ActionService?WSDL
brokerURL="failover://tcp://localhost:10000">

                <Action actionReference="TeMIP_AO_Directives_localhost">
                        <ServiceName>aoDirective</ServiceName>
                        <NmsName>localTeMIP</NmsName>
                </Action>
                <Action actionReference="TeMIP_TT_Directives_localhost">
                        <ServiceName>ttDirective</ServiceName>
                        <NmsName>localTeMIP</NmsName>
                </Action>
                <Action actionReference="TeMIP_FlowManagement">
                        <ServiceName>subscriptionManagement</ServiceName>
                        <NmsName>localTeMIP</NmsName>
                </Action>
        </MediationValuePack>

        <MediationValuePack MvpName="exec"
                            MvpVersion="1.0"
url=http://localhost:26700/uca/mediation/action/ActionService?WSDL
brokerURL="failover://tcp://localhost:10000">
                <Action actionReference="Exec_localhost">
                        <ServiceName>commandsExecution</ServiceName>
                        <NmsName>localhost</NmsName>
                </Action>
        </MediationValuePack>

</ActionRegistryXML>
```

XML Configuration 8 - ActionRegistry.xml example

In the example above, the Channel Adapter 'temip' Version '1.0' available through the URL 'http://localhost:26700/uca/mediation/action/ActionService?WSDL' (URL of the UCA Channel Adapter entry point, used by any Action processed by UCA for EBC) is able to manage the following services:

aoDirective: Action service to process TeMIP Alarm Object directives. This action service can be referenced in the rules by using the "TeMIP_AO_Directives_localhost" actionReference.

ttDirective: Action service to process TeMIP Trouble Ticket directives. This action service can be referenced in the rules by using the "TeMIP_TT_Directives_localhost" actionReference.

subscriptionManagement: Action service to process TeMIP Channel Adapter flow creation/deletion. This service is used by the UCA for EBC framework to create Alarm Flows at Value Pack start-up (See Chapter *3.4.2.2 Defining Collection flows*)

Note

☞ Please see [R2] *Unified Correlation Analyzer for Event Based Correlation – Administration, Configuration and Troubleshooting Guide* for more information on how to configure the ActionRegistry.xml file.

☞ Please refer to [R9] *OSS Open Mediation V7.1 Installation and Configuration Guide* for more information on how to configure the OSS Open Mediation V7.1 (NOM V7.1) to support the execution of Actions.

5.3.3 Action callbacks

It is possible to define action callbacks when using asynchronous actions. The action callback is a Java method that will be automatically called when the asynchronous action terminates.

This callback is called when the action is effectively processed. The callbacks are called at the next execution of the Asynchronous Action Management agent. This agent is called at regular intervals. The interval at which this agent is run depends on the value of the “asyncActionPeriod” Scenario policy. *See chapter 4.1.2.3 asyncActionPeriod*

Import

```
import com.hp.uca.common.callback.Callback;
```

Callback Constructor

The callback Object constructor is:

public Callback(java.lang.reflect.Method method, Object theObject, Object[] arguments)

Example

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.expert.flag.Flag;
import com.hp.uca.expert.vp.sample.ActionCallback;
import com.hp.uca.temip.mvp.aodirective.mapper.AODirective;
import com.hp.uca.temip.mvp.aodirective.mapper.AODirectiveKey;
import com.hp.uca.mediation.action.client.Action;
import com.hp.uca.mediation.action.exception.UcaActionExecutionException;
import com.hp.uca.mediation.action.exception.UcaActionInitializationException;


#declare any global variables here
global Scenario theScenario;

rule "Clear Alarm"
when
        alarm : Alarm()
then
        Action action = null;
        try {
                action = new Action("TeMIP_AO_Directives_localhost");
                action.addCommand(AODirectiveKey.DIRECTIVE_NAME,
                                AODirective.CLEARALARM);
                action.addCommand(AODirectiveKey.ENTITY_NAME, alarm.getIdentifier());
                action.addCommand(AODirectiveKey.USERID, "UCA EBC - ActionId: "
                                + action.getActionId());
                theScenario.addAction(action);
                try {
                        action.setCallback(ActionCallback
        .buildClearAlarmValidationCallback(action));
                } catch (SecurityException e) {
                // Manage the Exception
                } catch (NoSuchMethodException e) {
                // Manage the Exception
                }
                action.executeAsync(Action.NO_SYNCHRONIZATION_KEY);
        } catch (UcaActionInitializationException e) {
                // Manage the Exception
        }
end
```

You can define the following Java Class to define an Action callback.

```
package com.hp.uca.expert.vp.sample;

import java.lang.reflect.Method;

import org.apache.log4j.Logger;

import com.hp.uca.common.callback.Callback;
import com.hp.uca.common.trace.LogHelper;
import com.hp.uca.common.trace.SpecificLogManager;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.mediation.action.client.Action;

public final class ActionCallback {

        private static final int ARGUMENT_1 = 0;

        private static final int NB_CALLBACK_ARGUMENTS = 1;

        private static final String CLEAR_ALARM_CALLBACK_NAME =
"clearAlarmValidationCallback";

        /**
         * Log used to trace this component
         */
        private static final Logger LOG = Logger.getLogger(ActionCallback.class);

        /**
         * Hide Default constructor
         */
        private ActionCallback() {
        }

        /**
         * @param action
         * @return the {@linkplain Callback}
         *
         * @throws SecurityException
         * @throws NoSuchMethodException
         */
        public static Callback buildClearAlarmValidationCallback(Action action)
                        throws SecurityException, NoSuchMethodException {
                if (LOG.isTraceEnabled()) {
                        LogHelper.method(LOG, "buildClearAlarmValidationCallback()");
                }

                Class<?> partypes[] = new Class[NB_CALLBACK_ARGUMENTS];
                partypes[ARGUMENT_1] = Action.class;

                Object arglist[] = new Object[NB_CALLBACK_ARGUMENTS];
                arglist[ARGUMENT_1] = action;

                Method method = PD_ActionCallback.class.getMethod(
                                CLEAR_ALARM_CALLBACK_NAME, partypes);

                Callback callback = new Callback(method, null, arglist);

                return callback;
        }

        /**
         * @param action
         */
        public static void clearAlarmValidationCallback(Action action) {
                if (LOG.isTraceEnabled()) {
                        LogHelper.enter(LOG, "clearAlarmValidationCallback()");
                }

                // ActionStatus: Failed
                // ActionStatusExplanation: Specialized Exception: Generic Exception
                // (Source Of The Error = ClearAlarm)

                switch (action.getActionStatus()) {
                case Failed:
                        if (action.getActionStatusExplanation().contains(
                                        "Generic Exception (Source Of The Error =
ClearAlarm)")) {

                                LOG.warn("Was already cleared, forcing Action Status
to Completed");
                                action.acknowledgeActionFailure();
                        }
                        break;
                default:
                        break;
                }
```

```
                    if (LOG.isTraceEnabled()) {
                            LogHelper.exit(LOG, "clearAlarmValidationCallback()");
                    }
            }

    }
```

Rule Sample 15 - Action Callback example

5.4 Flags

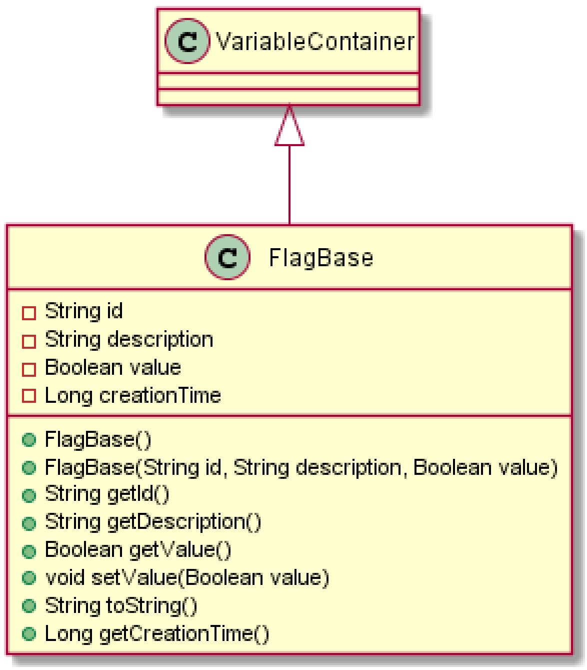
The UCA for EBC product provides a set of Flag Java object. These objects are useful to trigger rule execution in complex use cases or to trigger internal processing (Synchronization, etc…).

☞ Pleas refer to [R3] *Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Engine* for more information on Flag objects.

All Flag objects inherit from FlagBase class and have the following list of attributes and methods:

Attribute	Method	Type
`id`	`getId()`	`String`
`description`	`getDescription()`	`String`
`value`	`getValue()`	`Boolean`
`creationTime`	`getCreationTime()`	`Long`

Table 32 - Flags attributes and methods

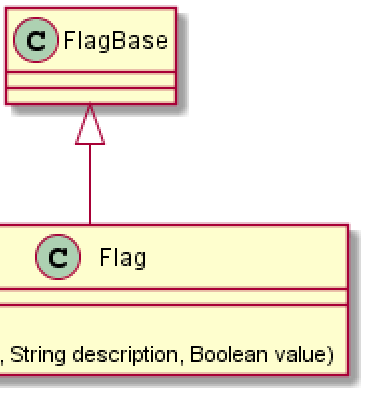


Flags attributes can be used in rules conditions, and Flag methods can be called in rules actions.

5.4.1 Flag

This kind of Flag can be explicitely inserted in Working Memory by some rules and be used to trigger other rules on demand.

```
import com.hp.uca.expert.flag.Flag;
```



Example

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.expert.flag.Flag;

#declare any global variables here
global Scenario theScenario;

rule "First Alarm => Insert Flag Object to start a Time Window"
no-loop
when
        firstAlarm : Alarm()
        not Flag(description  matches
firstAlarm.getOriginatingManagedEntity() )
then
    LogHelper.enter(theScenario.getLogger(), drools.getRule().getName());

    Flag flag=new Flag(firstAlarm.getIdentifier(),
firstAlarm.getOriginatingManagedEntity() , false);
    theScenario.getLogger().info("Inserting Flag for Context: " +
flag.getDescription());
        theScenario.getSession().insert(flag);

        theScenario.getLogger().info("Enabling the triggering of Context: " +
flag.getDescription());
        flag.setValue(true);

        theScenario.getLogger().debug("Add Watchdog item: ");
        theScenario.createWatchdogItem(5000, flag, false, "Flag:" +
flag.getDescription(),true);

        LogHelper.exit(theScenario.getLogger(), drools.getRule().getName());
end


rule "End of TimeWindow => Do Something once"
when
        firstAlarm : Alarm()
        myFlag : Flag(value == true,
                               id  matches firstAlarm.identifier)
then
    // Do Something once

                end
```

Rule Sample 16 - Flag example

In the rule example above, the Flag is used to trigger rule after a delay.

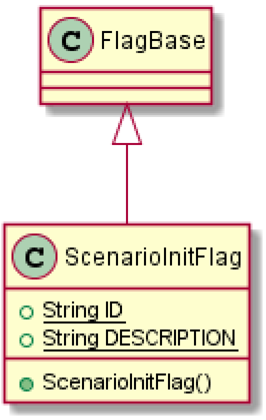
5.4.2 ScenarioInitFlag

This kind of Flag can be inserted by the rules in the Working Memory and used to trigger any rules on demand.

The rule developer can use this object to let multiple rules be triggered when the Scenario is started for instance.

No ScenarioInitFlag is automatically inserted by the Scenario.

```
import com.hp.uca.expert.flag.ScenarioInitFlag;
```



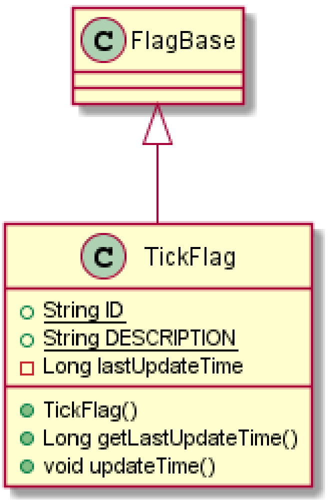
5.4.3 TickFlag

This Flag is automatically inserted in the Scenario's working memory at start-up.

It is convenient way to trigger rules when regular rule evaluation is needed.

The TickFlag is automatically updated in Working Memory with a periodicity configured by the "tickPeriod" Scenario property (See Chapter 4.1.2.2 tickPeriod).

```
import com.hp.uca.expert.flag.TickFlag;
```



Example

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.expert.flag.TickFlag;

#declare any global variables here
global Scenario theScenario;

rule "Rule - Regular tick processing for Alarm"
salience 10
no-loop
 when
        TickFlag( )
        alarm : Alarm( tickFlagAware == true)
 then
     // Do Something

                end
```

Rule Sample 17 - TickFlag example

The above rule will be regularly evaluated at TickFlag period, only for Alarms that are subject to regular evaluation (in this example we select Alarms based on the value of the tickFlagAware field).

The tickFlagAware field is available for following Objects:

- Alarm
- Group

5.4.4 SynchronizationFlag

This Flag is automatically inserted in the Scenario's working memory at start-up.

This flag can be used in rule conditions to trigger rules based on the Scenario's synchronization state. This flag indicates when the alarm resynchronization initiated at scenario startup or by request after the initial start of the scenario is complete.

```
import com.hp.uca.expert.flag.SynchronizationFlag;
```

Example

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.expert.flag.SynchronizationFlag;

#declare any global variables here
global Scenario theScenario;

rule "Store Mediation Resynchronization Start"
  when
    flag: SynchronizationFlag(value == false)
  then
    // Do Something when the Scenario Working Memory has been cleared, and
that the mediation is about to send historical alarms.

               end

rule "Store Mediation Resynchronization End"
  when
    flag: SynchronizationFlag(value == true)
  then
    // Do Something when the Scenario is fully synchronized
end
```

Rule Sample 18 - SynchronizationFlag example

The value of the SynchronizationFlag is automatically updated by the UCA for EBC framework:

- **false**: when at least one Mediation Flow associated with the Scenario's Value Pack is in a resynchronizing state. The Working Memory is automatically cleared or not or only partially, depending on the value of the Scenario's retractOnResyncPolicy(1), and a Resynchronization request is sent to the OSS Open Mediation Channel Adapter associated with the Mediation Flow.
- **true**: the Scenario is fully synchronized, i.e. all Mediation Flows associated with the Scenario's Value Pack are fully synchronized (no Mediation Flow associated with the Scenario's Value Pack is currently resynchronizing).

Notes

(1) ☞ Please see Chapter 4.1.1.4 retractOnResyncPolicy for more information.

5.4.5 Internal Flags

The following list of administration Flags are used by the UCA for EBC framework to trigger internal processing.

These flags are public for reference, but we do not recommend them to be used in customer rules.

Flag class	Usage
AsyncActionFlag	Recurrent processing of Asynchronous Actions. Update the status of completed Action in Working Memory when they are effectively completed See *Chapter 4.1.2.3* asyncActionPeriod
FireAllRuleFlag	Recurrent rule firing when Scenario is configured as 'WATCHDOG'. See *Chapter 4.1.1.2 fireAllRulesPolicy* and *Chapter 4.1.2.1 fireAllRulePeriod*
GarbageCollectionFlag	Recurrent rule firing when Alarm & Action garbage collection is issued. See *Chapter 4.1.2.4 garbageCollectionPeriod*

Table 33 - Value Pack properties

5.5 Groups

The UCA for EBC framework provides Java object to collect Alarms.

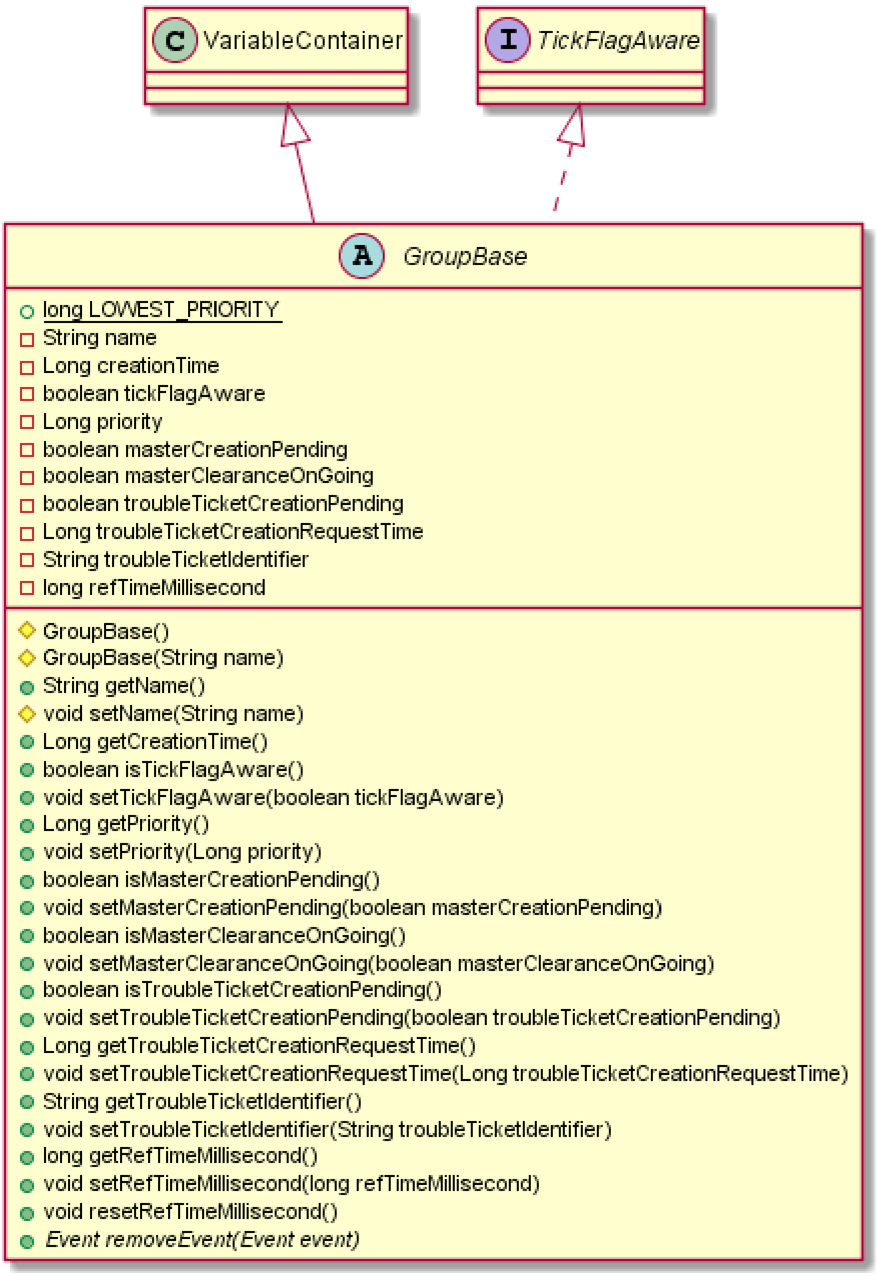
These are:

- Group
- PropagationGroup

These Group objects are useful to group Alarms and States. They act as Alarm containers but they also store additional information associated with the Group (creation time, LocalVariable, etc…).

☞Refer to [R3] Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Engine for detailed information.

Both Grouping classes inherit from GroupBase Java class:



C VariableContainer
I TickFlagAware
A GroupBase
long LOWEST_PRIORITY
String name
Long creationTime
boolean tickFlagAware
Long priority
boolean masterCreationPending
boolean masterClearanceOnGoing
boolean troubleTicketCreationPending
Long troubleTicketCreationRequestTime
String troubleTicketIdentifier
long refTimeMillisecond
GroupBase()
GroupBase(String name)
String getName()
void setName(String name)
Long getCreationTime()
boolean isTickFlagAware()
void setTickFlagAware(boolean tickFlagAware)
Long getPriority()
void setPriority(Long priority)
boolean isMasterCreationPending()
void setMasterCreationPending(boolean masterCreationPending)
boolean isMasterClearanceOnGoing()
void setMasterClearanceOnGoing(boolean masterClearanceOnGoing)
boolean isTroubleTicketCreationPending()
void setTroubleTicketCreationPending(boolean troubleTicketCreationPending)
Long getTroubleTicketCreationRequestTime()
void setTroubleTicketCreationRequestTime(Long troubleTicketCreationRequestTime)
String getTroubleTicketIdentifier()
void setTroubleTicketIdentifier(String troubleTicketIdentifier)
long getRefTimeMillisecond()
void setRefTimeMillisecond(long refTimeMillisecond)
void resetRefTimeMillisecond()
Event removeEvent(Event event)

5.5.1 Group

```
import com.hp.uca.expert.group.Group;
```

Group attributes can be used in rules conditions, and Group methods can be called in rules actions:

Attribute	Method	Description	Type
`name`	`getName()`	Name of the Group	`String`
`trigger`	`getTrigger()`	Alarm at the origin of the group creation	`Alarm`
`alarms`	`getAlarms()`	Map of Alarm to group	`Map<String ,Alarm>`

problemAlarm	getProblemAlarm()	Optional: used to store a Problem Alarm if needed. Usually, a problem alarm is a new Alarm generated by the rules to represent a group	Alarm
creationTime	getCreationTime()	Creation Time of the group. Automatically set at Group instantiation.	Long
refTimeMillisecond	getRefTimeMillisecond()	Usually the trigger Alarm reference time.	Long
var	getVar()	Optional. Used to store any Object needed in the rule processing	LocalVariable

Table 34 - Group attributes and methods

Example

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.group.Group;

#declare any global variables here
global Scenario theScenario;


rule "Correlation - Correlate & Attach to the Group"
  when
    a: Alarm(justInserted == true)
    group:  Group(trigger.getVar().getString("Site") matches
a.getVar().getString("Site"),
        trigger != a)
  then
    a.setJustInserted(false);

    // Do Something for instance
    group.getAlarms().put(a.getIdentifier(), a);
end
```

Rule Sample 19 - Group example

5.5.2 PropagationGroup

PropagationGroup class was introduced in V3.2

```
import com.hp.uca.expert.group.PropagationGroup;
```

PropagationGroup attributes can be used in rules conditions, and PropagationGroup methods can be called in rules actions:

Attribute	Method	Description	Type
name	getName()	Name of the Group	String
dbDomain	getDbDomain()	The DB domain	String
dbId	getDbId()	The DB ID	Long
dbType	getDbType()	The DB type	String
dbUniqueIdReference	getDbUniqueIdReference ()	The DB unique ref.	String
impactingStatesMap	getImpactingStatesList()	The Impacting States list	Collection <State>
rootCauseAlarmsMap	getImpactingStatesList()	The Root Cause Alarms list	Collection <Alarm>

creationTime	getCreationTime()	Creation Time of the group. Automatically set at Group instantiation.	Long
refTimeMillisecond	getRefTimeMillisecond()	Usually the trigger Alarm reference time.	Long
var	getVar()	Optional. Used to store any Object needed in the rule processing	LocalVariable

Table 35 - Group attributes and methods

Example

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.group.Group;

#declare any global variables here
global Scenario theScenario;


rule "Correlation - Correlate & Attach to the Group"
  when
    a: Alarm(justInserted == true)
    group:  Group(trigger.getVar().getString("Site") matches
a.getVar().getString("Site"),
       trigger != a)
  then
    a.setJustInserted(false);

    // Do Something for instance
    group.getAlarms().put(a.getIdentifier(), a);
end
```

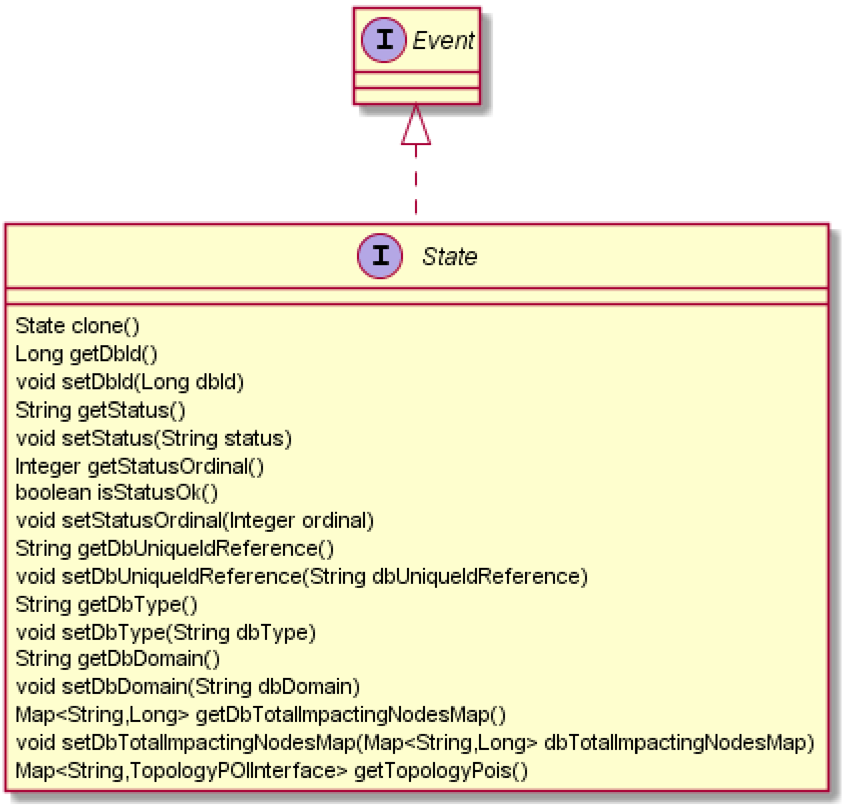
Rule Sample 20 - Group example

5.6 State

State objects were introduced in V3.2. They inherit Event class. They are useful objects to identify the current state of an impacted element stored in the Topology DB.

```
import com.hp.uca.expert.state.State;
```

State attributes can be used in rules conditions, and States methods can be called in rules actions


Event
State
State clone()
Long getDbId()
void setDbId(Long dbId)
String getStatus()
void setStatus(String status)
Integer getStatusOrdinal()
boolean isStatusOk()
void setStatusOrdinal(Integer ordinal)
String getDbUniqueIdReference()
void setDbUniqueIdReference(String dbUniqueIdReference)
String getDbType()
void setDbType(String dbType)
String getDbDomain()
void setDbDomain(String dbDomain)
Map<String,Long> getDbTotalImpactingNodesMap()
void setDbTotalImpactingNodesMap(Map<String,Long> dbTotalImpactingNodesMap)
Map<String,TopologyPOIInterface> getTopologyPois()

5.7 LocalVariable

The UCA for EBC framework provides a LocalVariable Java object. Such an object is useful to associate any Java Object to an Alarm, a Group, a Flag, or an Action.

☞Refer to [R3] Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Engine for detailed information.

LocalVariable methods

Method	Description	Type
`put(String key, Object value)`	Add an object in the LocalVariable using an identifier (key)	`void`
`get(String key)`	Get an Object stored in the LocalVariable using its identifier	`Object`
`remove(String key)`	Remove one Object defined by its identifier	`Object`
`getMap()`	Returns the Map where all Object are stored	`Map<String,Object>`
`getKeys()`	Returns all identifiers of the Object stored	`Set<String>`
`clear()`	Clear all entries from the LocalVariable	`void`
`getString(String key)`	Returns a String if the Object corresponding to the identifier is a String.	`String`
`getBoolean(String key)`	Returns a Boolean if the Object corresponding to the identifier is a Boolean.	`Boolean`

Table 36 - LocalVariable methods

Example

```
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.group.Group;

#declare any global variables here
global Scenario theScenario;


rule "LocalVariable example"
  when
    a: Alarm(justInserted == true)
    group:  Group(trigger.getVar().getString("Site") matches
a.getVar().getString("Site"),
        trigger != a)
  then
       AcmeObject myCustomizedObject=new AcmeObject();
       a.getVar().put("myDummyObject", myCustomizedObject);

       group.getVar().put("endOfProcess", new Boolean(true));

end
```

Rule Sample 21 - LocalVariable example

5.8 Watchdog Item

You can use Watchdog Items (associated to your scenario) to update any object in Working Memory after an initial delay or at regular intervals.

Then the update of the object in Working Memory after an initial delay or at regular intervals can be used to trigger other rules.

This feature is available through the Scenario interface:

```
public int createWatchdogItem(long timeDelayMs, Object object,
        boolean recurrent, String description,
        boolean cancelIfNotInWorkingMemory)
```

This method returns the id of the Watchdog Item that was just created.

☞Refer to [R3] Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Engine for detailed information.

WatchdogItem usage

Argument	Description	Type
`timeDelayMs`	The period or the delay before the 'update' is issued on the object in Working Memory. This time is in Millisecond	`long`
`object`	The Object that is 'updated' in Working Memory	`Object`
`recurrent`	True: the object will be recurrently updated. False: the object will be 'updated' only once after delay.	`boolean`
`description`	A textual description to qualify this Watchdog Item.	`String`
`cancelIfNotInWorkingMemory`	**True**: the 'object' will be updated only if it is still present in Working Memory. If the 'object' is not in Working Memory anymore, the WatchdogItem will be cancelled altogether (no more recurrence). **False**: the 'object' will be updated regardless of whether it is still present in Working Memory or not	`boolean`

Table 37 - WatchdogItem usage

Example

In the example below, the Watchdog Item feature is used to trigger a rule after a delay:

```
package com.hp.uca.expert.vp.cascading.communication;


#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.alarm.AlarmStateChange;
import com.hp.uca.expert.alarm.AlarmAttributeValueChange;
import com.hp.uca.expert.alarm.AlarmDeletion;
import com.hp.uca.expert.x733alarm.CustomFields;
import com.hp.uca.expert.x733alarm.CustomField;
import com.hp.uca.expert.x733alarm.PerceivedSeverity;
import com.hp.uca.expert.x733alarm.NetworkState;        // NOT_CLEARED, CLEARED
import com.hp.uca.expert.x733alarm.OperatorState;       // NOT_ACKNOWLEDGED,
ACKNOWLEDGED, TERMINATED
import com.hp.uca.expert.x733alarm.ProblemState;        // NOT_HANDLED, HANDLED, CLOSED
import com.hp.uca.expert.util.MessageFileHandler;
import com.hp.uca.mediation.action.client.Action;
import com.hp.uca.mediation.action.client.ActionStatus;
import com.hp.uca.mediation.action.jaxws.ActionResponseItem;
import com.hp.uca.temip.mvp.aodirective.mapper.AODirective;
import com.hp.uca.temip.mvp.aodirective.mapper.AODirectiveKey;
import com.hp.uca.temip.mvp.aodirective.mapper.Partition;
import com.hp.uca.temip.mvp.aodirective.mapper.EventID;
import java.util.ArrayList;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.common.trace.LogHelper;
import com.hp.uca.expert.flag.Flag;
import com.hp.uca.expert.vp.cascading.common.Actions;

#declare any global variables here
global Scenario theScenario;

declare Alarm
@role( event )
@timestamp( dateTimestamp )
@expires( 60s )
end

declare Flag
@role( event )
@timestamp( creationTime )
@expires( 60m )
end

rule "First Alarm => Insert Flag Object to start a Filter Time Window"
no-loop
when
        firstAlarm : Alarm()
        not Flag(description  matches firstAlarm.getOriginatingManagedEntity() )
then
    LogHelper.enter(theScenario.getLogger(), drools.getRule().getName());

    Flag flag=new Flag(firstAlarm.getIdentifier(),
firstAlarm.getOriginatingManagedEntity() , false);
    theScenario.getLogger().info("Inserting Flag for Context: " +
flag.getDescription());
        theScenario.getSession().insert(flag);

        theScenario.getLogger().info("Enabling the triggering of Context: " +
flag.getDescription());
        flag.setValue(true);

        theScenario.getLogger().debug("Create Watchdog item: ");
        int watchdogItemId = theScenario.createWatchdogItem(5000, flag, false, "Flag:"
+ flag.getDescription(),true);

        LogHelper.exit(theScenario.getLogger(), drools.getRule().getName());
end


rule "End of TimeWindow => Send first alarm and group similar alarms"
when
        firstAlarm : Alarm()
        myFlag : Flag(value == true,
                                 id  matches firstAlarm.identifier)
        alarms : ArrayList()
             from collect(
                Alarm( identifier != firstAlarm.identifier,
                         originatingManagedEntity ==
firstAlarm.getOriginatingManagedEntity(),
                            this after [0, 20s ] firstAlarm) )
then
```

```
        LogHelper.enter(theScenario.getLogger(), drools.getRule().getName());

        Flag newFlag=myFlag;

        ArrayList<Alarm> newAlarms=new ArrayList<Alarm>();
        if (alarms != null) {
                theScenario.getLogger().info("Grouping Alarm: " +
firstAlarm.getIdentifier());

                for (Object o: alarms) {
                        newAlarms.add((Alarm) o);
                }
                Actions.associateAlarms(theScenario,firstAlarm,newAlarms);

        } else {
                theScenario.getLogger().debug("No additional Alarms");
        }

        theScenario.getLogger().debug("delegateAlarm: " +firstAlarm.getIdentifier());
        theScenario.getSession().retract(firstAlarm);

        theScenario.delegateAlarmToScenario("myVP", "1.0",
"com.hp.uca.expert.vp.cascading.Enrichment", firstAlarm);

        for (Alarm a: newAlarms) {
                theScenario.getSession().retract( a);
        }

        theScenario.getSession().retract(newFlag);
        LogHelper.enter(theScenario.getLogger(), drools.getRule().getName());

                end
```

Rule Sample 22 - WatchdogItem example

In case you need to, Watchdog Items can be removed by calling the `deleteWatchdogItem(…)` method available through the Scenario interface:

```
public boolean deleteWatchdogItem(int watchdogItemId)
```

You have to pass the Watchdog Item Id of the Watchdog Item you want to remove to the `deleteWatchdogItem(…)` method.
You can retrieve the Watchdog Item Id of a Watchdog Item when you create it. It is returned by the call to the `createWatchdogItem(…)` method.

5.9 Watchdog Item callbacks

The UCA for EBC framework provides a temporal feature called “Watchdog Item callback” to call a Java method after an initial delay or at regular intervals.

This feature is available through the Scenario interface:

```
public int createCallbackWatchdogItem(long timeDelayMs,
        Callback callback,
        boolean recurrent, String description,
        boolean cancelIfNotInWorkingMemory, Object objectInWM)
```

This method returns the id of the Callback Watchdog Item that was just created.

```
import com.hp.uca.common.callback.Callback;
```

Callback Constructor

The callback Object constructor is:

public Callback(java.lang.reflect.Method method, Object theObject, Object[] arguments)

The Callback object is shared with the Action Callback (see chapter 5.3.3 Action callback)

☞Refer to [R3] Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Engine for detailed information.

WatchdogItem Callback usage

Argument	Description	Type
`timeDelayMs`	The period or the delay before the callback is called. This time is in Millisecond	`long`
`callback`	The callback to invoke at timer expiration	`Callback`
`recurrent`	true: the callback will be recurrently invoked. false: the callback will be invoked only once after delay.	`boolean`
`description`	A textual description to qualify this Callback.	`String`
`cancelIfNotInWorkingMemory`	**True**: the callback will be executed only if the 'objectInWM' is still present in Working Memory. If the 'objectInWM' is not in Working Memory anymore, the WatchdogItem Callback will be cancelled altogether (no more recurrence). **False**: the callback will be executed regardless of whether the 'objectInWM' is still present in Working Memory or not	`boolean`
`objectInWM`	The Object to verify in Working Memory	`Object`

Table 38 - WatchdogItem Callback usage

Example

In the example below, the Watchdog Item Callback feature is used to call a navigation processing after a while (update some Alarm field in TeMIP Client GUI, for instance)

```
package com.hp.uca.expert.vp.cascading.communication;


#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.expert.flag.Flag;
import com.hp.uca.expert.sample.Navigation;


#declare any global variables here
global Scenario theScenario;

rule "Do some Java processing after a while"
when
        alarm : Alarm()
then

        Navigation.addCallbackForNavigation(// Add needed arguments);

end
```

And the associated Java processing:

```
package com.hp.uca.expert.vp.sample;

import java.lang.reflect.Method;
import java.util.Set;

import org.apache.log4j.Logger;

import com.hp.uca.common.callback.Callback;
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.group.Group;
import com.hp.uca.expert.group.Qualifier;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.expert.vp.sample.interfaces.CustomInterface;
import com.hp.uca.expert.vp.sample.services.Service_Util;


public final class Navigation {

        private static final String CALLBACK_FOR_NAVIGATION_DESCRIPTION = "Callback
for navigation";

        private static final String EXCEPTION_RECEIVED_BY_ADDCALLBACK_FOR_NAVIGATION =
"Exception received by addcallbackForNavigation()";

        private static final int ARGUMENT_4 = 3;

        private static final int ARGUMENT_3 = 2;

        private static final int ARGUMENT_2 = 1;

        private static final int ARGUMENT_1 = 0;

        private static final int NB_CALLBACK_ARGUMENTS = 4;

        private static final String CALLBACK_FOR_NAVIGATION_METHOD_NAME =
"callbackForNavigation";

        private static final int INFINITE_DELAY = -1;

        /**
         * Log used to trace this component
         */
        private static final Logger LOG = Logger.getLogger(Navigation.class);

        /**
         * Hide Default constructor
         */
        private PD_Navigation() {
        }
        /**
         * @param scenario
         * @param alarm
         * @param qualifier
         * @param custom
         * @param delay
         */
        public static void addCallbackForNavigation(Scenario scenario,
                         Alarm alarm, Qualifier qualifier, CustomInterface custom,
long delay) {

                Class<?> partypes[] = new Class[NB_CALLBACK_ARGUMENTS];
                partypes[ARGUMENT_1] = Scenario.class;
                partypes[ARGUMENT_2] = Alarm.class;
                partypes[ARGUMENT_3] = Qualifier.class;
                partypes[ARGUMENT_4] = CustomInterface.class;

                Object arglist[] = new Object[NB_CALLBACK_ARGUMENTS];
                arglist[ARGUMENT_1] = scenario;
                arglist[ARGUMENT_2] = alarm;
                arglist[ARGUMENT_3] = qualifier;
                arglist[ARGUMENT_4] = custom;

                try {
                        Method method = Navigation.class.getMethod(
                                        CALLBACK_FOR_NAVIGATION_METHOD_NAME,
partypes);

                        Callback callback = new Callback(method, null, arglist);
                        int watchdogItemId =
scenario.createCallbackWatchdogItem(delay, callback, false,
                                        CALLBACK_FOR_NAVIGATION_DESCRIPTION, true,
alarm);

                } catch (SecurityException e) {
                     // Manage the exception
                } catch (NoSuchMethodException e) {
                     // Manage the exception
                }

        }

        /**
         * @param scenario
         *            the current scenario where this processing takes place
         * @param alarm
         * @param qualifier
         * @param custom
         */
        public static void callbackForNavigation(Scenario scenario,
                         Alarm alarm, Qualifier qualifier, CustomInterface custom) {

                  // Do something depending on arguments of this method
        }
}
```

Rule Sample 23 - WatchdogItem callback example

In case you need to, Watchdog Items can be removed by calling the `deleteWatchdogItem(…)` method available through the Scenario interface:

```
public boolean deleteWatchdogItem(int watchdogItemId)
```

You have to pass the Watchdog Item Id of the Watchdog Item you want to remove to the `deleteWatchdogItem(…)` method.
You can retrieve the Watchdog Item Id of a Watchdog Item when you create it. It is returned by the call to the `createCallbackWatchdogItem(…)` method.

5.10 Rule Session

When writing rules, the Working Memory should only be accessed through the `com.hp.uca.expert.rulesession.RuleSessionInterface.`

The Rule Session is available using the following method on the Scenario:

```
theScenario.getSession()
```

Important Note

Drools **insert**, **update**, and **retract** statements MUST NOT be used directly. It is mandatory, for working memory integrity to use the RuleSessionInterface to perform these Working Memory actions, by using the following statements instead:

theScenario.getSession().insert(...)

theScenario.getSession().update(...)

theScenario.getSession().retract(...)

☞Refer to [R3] *Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Engine* for detailed information.

RuleSession usage

Method	Description	Return Type
`insert(Object object)`	Insert an object in the Working Memory. Insertion is the act of telling the WorkingMemory about a fact, which you do by *theScenario.getSession().insert(yourObject).* When you insert a fact, it is examined for matches against the rules. This means all of the work for deciding about firing or not firing a rule is done during insertion; no rule, however, is executed until the Scenario calls fireAllRules() depending on Scenario's policies. See *[R5] JBoss* Drools Expert guide for more information about WorkingMemoryEntryPoint insertion	`FactHandle`
`insert(Collection<Object> objectCollection)`	Insert a collection of object in the Working Memory in "bulk mode" (the rules are not fired until the whole collection is inserted). This method is similar to the `insert(Object object)` method except that it inserts a collection of object in the Working Memory instead of just one object. This method can be called by writing the following code in your rules file: *theScenario.getSession().insert(yourObjectCollection).* The insertion of the collection of objects passed as parameter to the method into Working Memory is done in "bulk mode": the rules are not fired during the insert. Instead, the rules are fired at the end of the insertion of the collection of objects, depending on the Scenario's policies, provided the Scenario is in FireAllRulesPolicy.EACH_ACCESS mode.	`Collection<FactHandle>`
`update(Object object)`	Update an object in the Working Memory. The Rule Engine must be notified of modified facts, so that they can be reprocessed. Internally, modification is actually a retract followed by an insert (automatic processing of Drools); the Rule Engine removes the fact from the WorkingMemory and inserts it again. You must use the *theScenario.getSession().update()* method to notify the WorkingMemory of changed objects. See *[R5] JBoss* Drools Expert guide for more information about WorkingMemoryEntryPoint update	`FactHandle`
`retract(Object object)`	Retract an object from the Working Memory. Retraction is the removal of a fact from Working Memory, which means that it will no longer track and match that fact, and any rules that	`void`

	are activated and dependent on that fact will be cancelled. Note that it is possible to have rules that depend on the nonexistence of a fact, in which case retracting a fact may cause a rule to activate. See *[R5] JBoss* Drools Expert guide for more information about WorkingMemoryEntryPoint retraction	
`getQueryResults(String query, Object... arguments)`	Retrieves the QueryResults of the specified Drools query and arguments. See *[R5] JBoss* Drools Expert guide for more information about Drools Queries	`QueryResults`
`openLiveQuery(String query, Object[] arguments, ViewChangedEventListener listener);`	Opens a Live Query, which has a listener attached to it that listens to changes in the result set of the LiveQuery (instead of returning an iterable result set as is the case with the getQueryResults(String, Object...) method) See [R5] JBoss Drools Expert guide for more information about Drools Live Queries	`LiveQuery`

Table 39 - Rule Session usage

Example

```
package com.hp.uca.expert.vp.sample;


#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.scenario.Scenario;


#declare any global variables here
global Scenario theScenario;

rule "Example of Rule Session access"
when
        alarm : Alarm()
then

        Flag flag=new Flag(firstAlarm.getIdentifier(),
                         firstAlarm.getOriginatingManagedEntity() , false);

        // Do not use insert() directly, use the getSession() instead
        theScenario.getSession().insert(flag);


        // Do not use retract() directly, use the getSession() instead
        theScenario.getSession().retract(alarm);
end
```

Rule Sample 24 - RuleSession (WorkingMemory) mandatory access

5.11 Collection Flows

The Mediation Flows and DB Flows defined in the `ValuePackConfiguration.xml` file of a Value Pack are accessible from the rules.

The list of Mediation Flows associated with a Scenario's Value Pack are available using the following method on the Scenario's Value Pack:

```
theScenario.getValuePack().getValuePackMediationFlows()
```

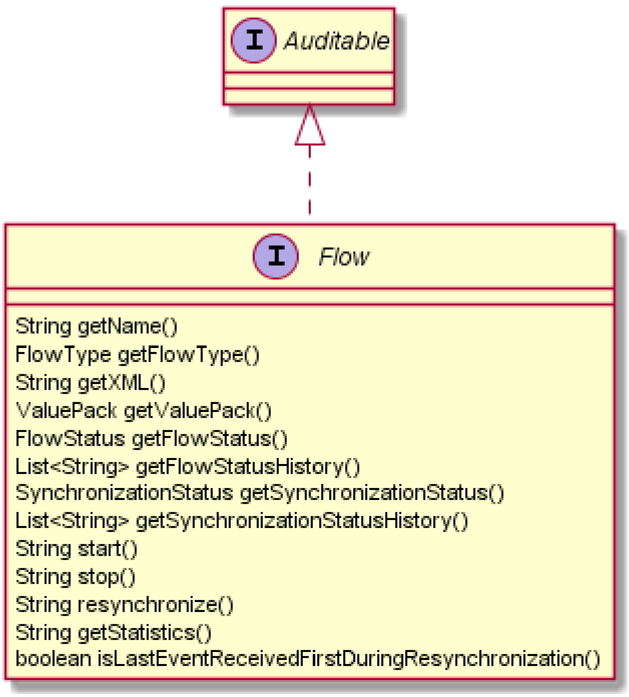
The list of DB Flows associated with a Scenario's Value Pack are available using the following method on the Scenario's Value Pack:

```
theScenario.getValuePack().getDbFlowsMap()
```

You can then get information on each Collection Flow (its Name, Type, Status, and Synchronization Status among other things) and also perform actions on each Mediation Flow, namely start/stop/resynchronize it:

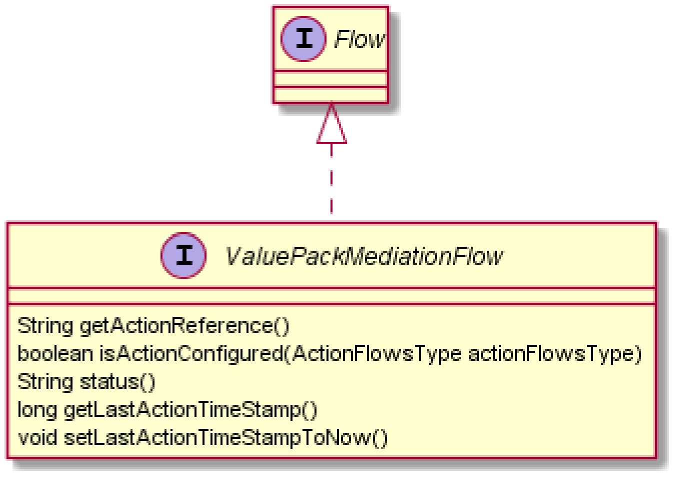
- getName()
- getFlowType()
- getFlowStatus()
- getFlowStatusHistory()
- getSynchronizationStatus()
- start()
- stop()
- resynchronize()

Each Collection Flow inherit the Flow Java interface.

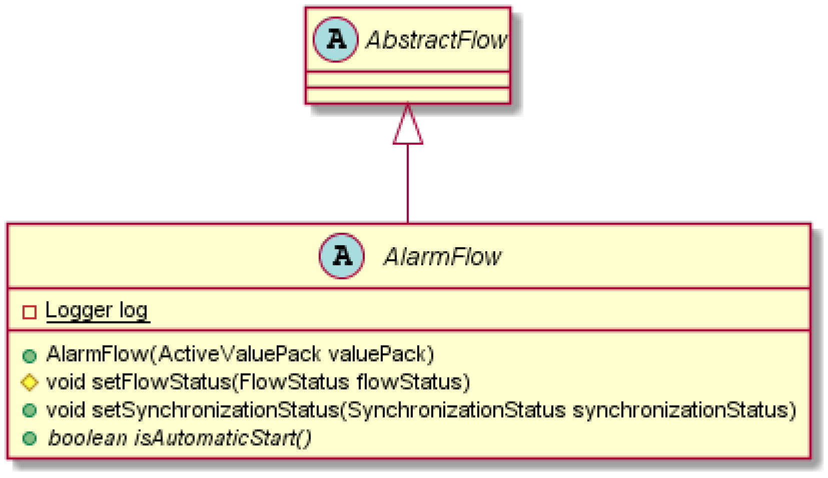


```
String getName()
FlowType getFlowType()
String getXML()
ValuePack getValuePack()
FlowStatus getFlowStatus()
List<String> getFlowStatusHistory()
SynchronizationStatus getSynchronizationStatus()
List<String> getSynchronizationStatusHistory()
String start()
String stop()
String resynchronize()
String getStatistics()
boolean isLastEventReceivedFirstDuringResynchronization()
```

A Mediation Flow is a specialized Flow as per below diagram:


I Flow
I ValuePackMediationFlow
String getActionReference()
boolean isActionConfigured(ActionFlowsType actionFlowsType)
String status()
long getLastActionTimeStamp()
void setLastActionTimeStampToNow()

A DB Flow is a specialized Flow as per below diagram:


A AbstractFlow
A AlarmFlow
Logger log
AlarmFlow(ActiveValuePack valuePack)
void setFlowStatus(FlowStatus flowStatus)
void setSynchronizationStatus(SynchronizationStatus synchronizationStatus)
boolean isAutomaticStart()

☞Please refer to [R3] *Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Engine* for detailed information.

Below is an example of how to work with Mediation Flows in Rules files:

Example

```
package com.hp.uca.expert.vp.sample;

#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.CustomFields;
import com.hp.uca.expert.x733alarm.CustomField;
import com.hp.uca.expert.x733alarm.PerceivedSeverity;
import com.hp.uca.expert.x733alarm.NetworkState;        // NOT_CLEARED, CLEARED
import com.hp.uca.expert.x733alarm.OperatorState;       // NOT_ACKNOWLEDGED,
ACKNOWLEDGED, TERMINATED
import com.hp.uca.expert.x733alarm.ProblemState;        // NOT_HANDLED, HANDLED, CLOSED
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.common.trace.LogHelper;
import com.hp.uca.expert.vp.flow.FlowStatus;
import com.hp.uca.expert.vp.flow.ValuePackMediationFlow;

#declare any global variables here
global Scenario theScenario;


rule "My Rule"
  when
    …
  then
         LogHelper.enter(theScenario.getLogger(), drools.getRule().getName());

        …
        for (ValuePackMediationFlow flow:
             theScenario.getValuePack().getValuePackMediationFlows().values()) {
                …

                if (flow.getFlowStatus().equals(FlowStatus.Inactive)) {
                        flow.start();
                }
                …
                if (flow.getFlowStatus().equals(FlowStatus.Active)) {
                        flow.stop();
                }
                …
                if (flow.getFlowStatus().equals(FlowStatus.Active)) {
                        flow.resynchronize();
                }
                …
        }
        …
        LogHelper.exit(theScenario.getLogger(), drools.getRule().getName());
end
```

Rule Sample 25 – Mediation Flow example

5.12 Scenario Loggers

The UCA for EBC product provides advanced logging mechanism that is able to trace specific rule processing for each Scenario.

The UCA for EBC Administration GUI fully supports this logging mechanism.

☞ [R10] *Unified Correlation Analyzer for Event Based Correlation – User Interface Guide*, chapter Troubleshooting UCA for event based Correlation

To take benefits from this mechanism, the rule developer must use the logger provided by the Scenario:

```
theScenario.getLogger()
```

The getLogger() method provides access to a standard *org.apache.log4j.Logger* object.

All standard Logger methods are available to better qualify the level of information needed (for example info(), debug(), warn(), etc…)

Example

```
package com.hp.uca.expert.vp.sample;

#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.x733alarm.CustomFields;
import com.hp.uca.expert.x733alarm.CustomField;
import com.hp.uca.expert.x733alarm.PerceivedSeverity;
import com.hp.uca.expert.x733alarm.NetworkState;        // NOT_CLEARED, CLEARED
import com.hp.uca.expert.x733alarm.OperatorState;       // NOT_ACKNOWLEDGED,
ACKNOWLEDGED, TERMINATED
import com.hp.uca.expert.x733alarm.ProblemState;        // NOT_HANDLED, HANDLED, CLOSED
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.common.trace.LogHelper;

#declare any global variables here
global Scenario theScenario;


rule "Any new Acknowledged Alarm"
  when
    a: Alarm(operatorState == OperatorState.ACKNOWLEDGED)
  then
         LogHelper.enter(theScenario.getLogger(), drools.getRule().getName());

        theScenario.getLogger().info("[RULE " + drools.getRule().getName() + "] Found
new acknowledged alarm: identifier = " + a.getIdentifier()+ ":");
        theScenario.getLogger().debug(a.toFormattedString());

        LogHelper.exit(theScenario.getLogger(), drools.getRule().getName());
end

rule "Any new Terminated Alarm"
  when
        a: Alarm(operatorState == OperatorState.TERMINATED)
  then
         LogHelper.enter(theScenario.getLogger(), drools.getRule().getName());

        theScenario.getLogger().info("[RULE " + drools.getRule().getName() + "] Found
new terminated alarm: identifier = " + a.getIdentifier() + ":");
        theScenario.getLogger().debug(a.toFormattedString());

        LogHelper.exit(theScenario.getLogger(), drools.getRule().getName());

                  end
```

Rule Sample 26 - Scenario logger example

Chapter 6

Advanced UCA for EBC features

6.1 Orchestration of event cascading between scenarios

The Orchestrating event cascading feature is introduced since UCA-EBC 3.1, as an extension of the alarms cascading between scenarios provided in UCA-EBC 3.0. It is the capability of defining, per UCA EBC server, an event workflow between the scenarios running on the same server (belonging or not to the same Value Pack). This workflow is made of several routes, each of them describing the way in which an event is cascaded to one or more scenarios:

- Copying an event from one scenario to another **(COPY)**
- Aggregating the information (**orchestra data**) added by several scenarios on copies of the same event, and sending the resulted aggregated event to another scenario **(JOIN)**.

This feature is implemented via one method of the Scenario interface:

```
boolean applyOrchestration(Event event);
```

For all types of operations, orchestration is implemented regardless of the scenario's mode (CLOUD or STREAM). This means that **all possible combinations** of pushing events between scenarios defined in the orchestration workflow are accepted:

- cascading from a STREAM scenario to a STREAM scenario
- cascading from a STREAM scenario to a CLOUD scenario
- cascading from a CLOUD scenario to a STREAM scenario
- cascading from a CLOUD scenario to a CLOUD scenario

However, in the case of the **JOIN** operation, there are some limitations for the STREAM to CLOUD cascading, detailed in 6.1.5 "Scenario mode (STREAM/CLOUD) impact on event cascading".

6.1.1 Orchestration Principles

As described above the Orchestration is made thanks to the definitions of **Routes** between scenarios.

The two Route types available for Orchestration are COPY and JOIN, detailed in the following subsections.

Each of the routes has to have at least one **source scenario** and one **destination scenario**.

- **The Source scenario**

  This scenario will provide the event to be cascaded. It is identified by:

  - the full valuepack name (valuePackName-valuePackVersion)
  - the scenario name.

- **The Destination scenario**

  It is composed of:

  - An **Orchestra Filter** (Optional). It is the name of a TopFilter part of the Orchestra Filters.

    Orchestra Filters are defined in a specific Filter Configuration File: *${UCA_EBC_INSTANCE}/conf/OrchestraFilters.xml*. The schema for this file is the same as the one used for defining the scenario's Filters.

    A filter is specified by its name (TopFilter's name).

    The Orchestra Filter is an optional parameter. When specified, the Event will have to pass this filter prior to be cascaded to the target scenario.When the filter is not specified, the event is systematically cascaded to the target scenario.

  - A **Target scenario**. To this scenario the event is cascaded. It is identified by:

    - the full valuepack name (valuePackName-valuePackVersion)
    - the scenario name.

6.1.1.1 The COPY route

This Route is used to copy an Event from **a source scenario** to **one or several destination scenarios**.

The orchestrated event is duplicated and a cloned event is sent to each destination scenario. Cloned events are independent Java Objects that can be updated by all scenarios without interactions between each other.

A COPY Rule example is shown in Figure 11:

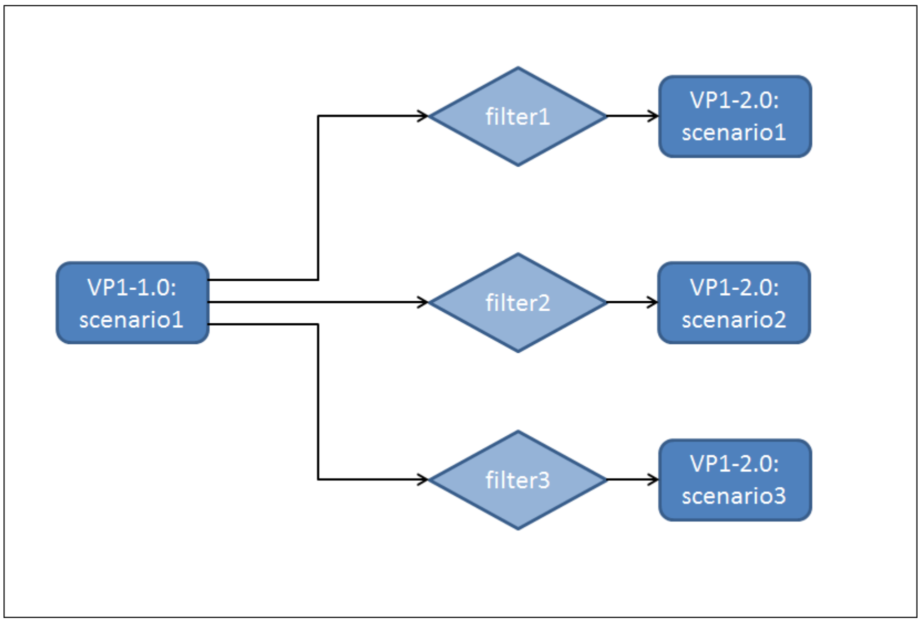


Figure 11 - COPY Route example

Once the event is cascaded to the target scenario, it is injected to the scenario in a way that the scenario's filters (filters defined at the scenario level) are evaluated. The event will then be pushed to the target scenario's working memory only if it passes both the Orchestra filter (if specified) and at least one of the scenario's top filters.

During the copy of the Event, the clone() overridable Java method is invoked. If the cascaded object's class extends the Event class, make sure that you have overridden the clone() method.

A good practice, in the case of a COPY route, is to define the target scenario's policy 'eligibleForBroadcast' as 'false'. This is to ensure that events entering the scenario are coming only from the source Scenario and not also from the network.

See chapter 4.1.3.1 eligibleForBroadcast.

6.1.1.2 The JOIN route

This Route is used to aggregate information coming from **several source scenarios** into an enriched event sent to **a destination scenario**.

In the case of JOIN Route the destination scenario is sometimes called **"convergence scenario"**.

The JOIN method supposes that:

- The source scenario is either eligible to broadcast or receives cascaded events.
- There is at least another scenario (on the same UCA-EBC server) that receives the same event (2 source scenarios at least for one target scenario). If not, it is advised to use COPY instead.

As for the COPY Route, the Destination is defined by an Orchestra Filter (optional) and a Target scenario. And as for the COPY Route, the Event will have to pass the filter (when defined) prior to be cascaded to the target scenario.

The JOIN route has an extra tag that must be specified: the **expireTime** (milliseconds). This is used to limit (timeout) the wait time of the join operation.

A timer is activated at the receipt of the first event contributing to the JOIN route. Then,

- If events are received from all source scenarios before the expireTime is reached, the aggregated event will contain enrichement data for all these source scenarios and the boolean `convergenceComplete` is set to true.
- If the expireTime is reached before all events are received from the source scenarios, the aggregated event will contain only the enrichment data received so far and the boolean `convergenceComplete` is set to false.

Join with timer example example:

3 source scenarios contribute to a join with an expiration time of 1000 milliseconds.

If only 2 event copies are received from the source scenarios and 1000 milliseconds passed since the reception of the first event, then the event containing aggregated information from the two contributing events is sent to the convergence scenario and the convergenceComplete boolean is set to false.

When the third event is received later on (if any), it is also sent to the convergence scenario, having the convergenceComplete tag set to false.

Therefore, the convergence scenario has to manage the case of receiving several events with the boolean convergenceComplete set to false or one event with the boolean set to true.

Note: Even if the third event will be the only one to participate in the second JOIN, the JOIN mechanism waits again for an expireTime period before sending this event to the target scenario.

A JOIN Rule example is shown in Figure 12:

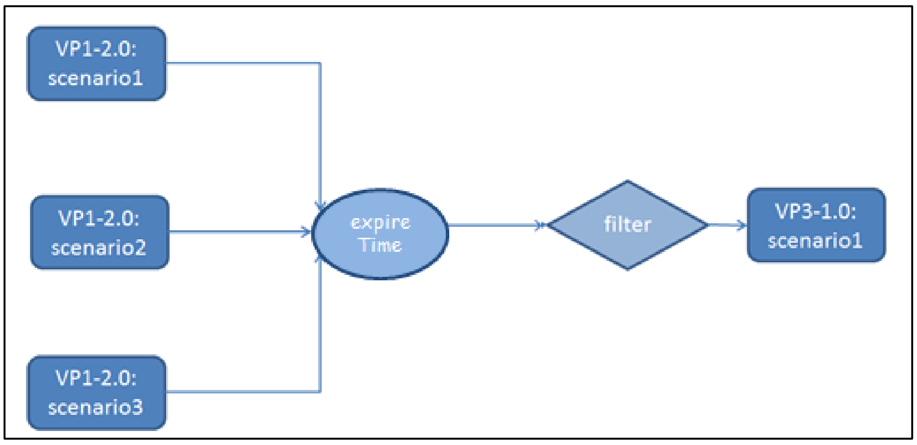


Figure 12 - JOIN Rule example

Note that the same source scenario can contribute to different “convergence scenarios”.

Events are cloned Java Objects that can be updated by all Scenarios without interactions with each other.

The JOIN route source scenarios can enrich the event by calling the method of the *Orchestrable* interface:

```
void addOrchestraDataInScenario(Serializable orchestraData);
```

The data information has to be a **Serializable object**. It will be internally attached to the event in a Map using the **full scenario name.** The full scenario name is composed of the scenario name, its corresponding value pack name and version) of the calling scenario as key.

All primitive types are Serializable. If wanting to add a custom object, it has to implement the *java.io.Serializable* interface.

The JOIN route implementation sends an aggregated event containing the orchestra data added by each of the source scenarios source to the destination (convergence) scenario.

The orchestra data added by all the source scenarios to the event can be retrieved by the convergence scenario by calling the method of the *Orchestrable* interface:

```
Map<String, Serializable> getOrchestraData();
```

Also, retrieve only the orchestra Data added by a specific scenario on the event can be done by calling the method of the *Orchestrable* interface:

```
Serializable getOrchestraData(String scenarioFullName);
```

The key the full scenario name is in the form of **ValuePackName-ValuePackVersion:ScenarioName**.

Note: As for the COPY Route the event cascaded to the target scenario, is injected to the scenario in a way that the scenario’s filter (filters defined at the scenario level) are evaluated. The event will then be pushed to the target scenario’s working memory only if it passes both the Orchestra filter (if specified) and at least one of the scenario’s top filter.

6.1.2 Orchestra Routes Configuration File

For the events to be routed between Value Packs, Orchestration Routes have to be defined in the `OrchestraConfiguration.xml` file of the UCA-EBC server instance, in the *${UCA_EBC_INSTANCE}/conf* folder. This file is only loaded at UCA-EBC server instance start (**static loading**), so **if this file is modified, the server has to be restarted so that the new Orchestration configuration can be taken into consideration**.

This configuration files complies to the schema definition: *${UCA_EBC_HOME}/schemas/OrchestraConfiguration.xsd*

The <**OrchestraWorkflow**> tag of the OrchestraConfiguration.xml file contains the list of all the routes defined between the scenarios, as shown in the simplified schema below:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<OrchestraWorkflow>
    <Routes>
        <Route>
            <COPY>
                <Source>
                    <ValuePackNameVersion></ValuePackNameVersion>
                </Source>
                <Destinations>
                    <Destination>
                        <Filter>
                            <filterName></filterName>
                        </Filter>
                        <Target>
                            <ValuePackNameVersion></ValuePackNameVersion>
                        </Target>
                    </Destination>
                </Destinations>
            </COPY>
          or
            <JOIN>
                <Sources>
                    <Source>
                        <ValuePackNameVersion></ValuePackNameVersion>
                    </Source>
                </Sources>
                <ExpireTime></ExpireTime>
                <Destination>
                    <Filter>
                        <filterName></filterName>
                    </Filter>
                    <Target>
                        <ValuePackNameVersion></ValuePackNameVersion>
                    </Target>
                </Destination>
            </JOIN>
        <Route>
    </Routes>
</OrchestraWorkflow>
```

Each of the tags of the Orchestra schema is described in the following tables:

OrchestraWorkflow Tag		
Defines the Orchestra workflow		
Type	**Name**	**Value**

Element	Routes	Container for each Route definition

Table 40 - OrchestraWorkflow Tag

Routes Tag		
Defines the Routes container		
Type	**Name**	**Value**
Element	List<Route>	Defines a list of <Route>

Table 41 - Routes Tag

Route Tag		
Define a route (either COPY or JOIN)		
Type	**Name**	**Value**
Choice	COPY	Defines a COPY Route
Choice	JOIN	Defines a JOIN Route

Table 42 - Route Tag

COPY Tag		
Defines a COPY Route		
Type	**Name**	**Value**
Element	Source	Identifies the source scenario requesting the event routing. See ☞ Table 48 for detailed definition of scenario identification.
Element	Destinations	Identifies a list of destination scenarios which will eventually receive a copy of the routed event.

Table 43 - Orchestra COPY route properties

JOIN Tag		
Defines a JOIN route		
Type	**Name**	**Value**
Element	Sources	Identifies a list of source scenarios which will contribute to the JOIN route with events.
Element	ExpireTime	Specifies the maximum wait time (in milliseconds and starting from the reception of the first copy of an event contributing to the join route) for the aggregation of an event.
Element	Destination	Identifies the scenario destination which will eventually receive the cascaded event containing all the aggregated orchestra data.

Table 44 - Orchestra JOIN route properties

Destinations Tag		
Define a list of destinations		
Type	**Name**	**Value**
Element	List<Destination>	Defines a list of <Destination>

Table 45 - Destinations Tag

Destination Tag		
Define a list of destinations		
Type	**Name**	**Value**
Element	Filter	**(Optional)** An Orchestra Top Filter name the Event has to pass to be sent to the target Scenario
Element	Target	Identifies the target scenario the event is sent to. See ☞ Table 48 for detailed definition of scenario identification.

Table 46 - Destination Tag

Sources Tag		
Define a list of Sources		
Type	**Name**	**Value**
Element	List<Source>	Defines a list of <Source> Source identifies the source scenario requesting the event routing. See☞Table 48 for detailed definition of scenario identification.

Table 47 - Sources Tag

Scenario identification (<Source> or <Target> tags)		
Type	**Name**	**Value**
Element	ValuePackNameVersion	Identifies the Value Pack Name and version with the form: *valuepackName-valuePackVersion*
Element	ScenarioName	Identifies the Scenario Name.

Table 48 - Orchestra Scenario identification (<Source> or <Target> tags)

6.1.3 Orchestration definition example

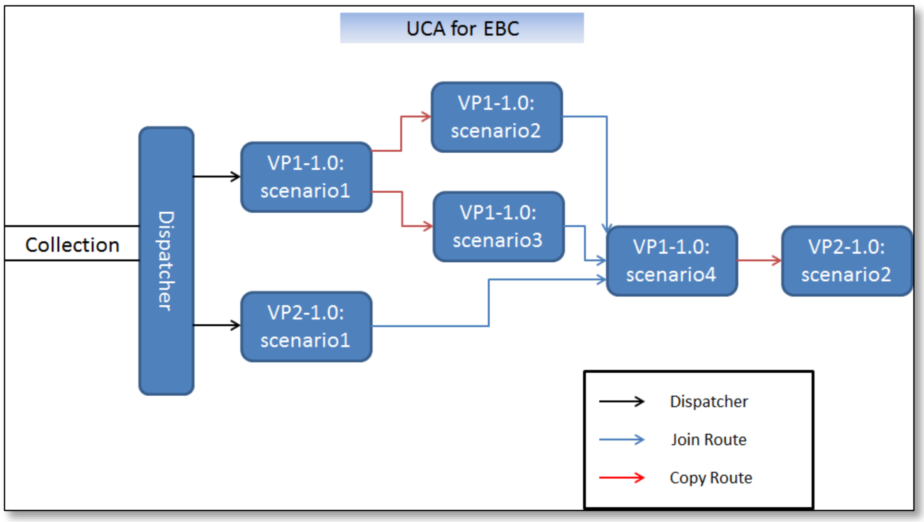
The following is a sample *OrchestraConfiguration.xml* file (assuming this file is deployed in the ${UCA_EBC_INSTANCE}/conf directory). It describes an Orchestration Workflow with the following routes (assuming only scenario1 in VP1 and scenario1 in VP2 are receiving events from the network (eligibleToBroadcast is set to true)):

- 1 route to copy the event from one scenario (scenario1 from VP1) to other 2 scenarios (scenario2 and scenario3 from VP1)
- 1 route to converge (join) the event copies of the same event from 3 scenarios (scenario2 and scenario3 from VP1 and scenario1 from VP2) into one event to send to another scenario (scenario4 in VP1), and having the expire time for the convergence set to 1000 milliseconds
- 1 route to copy the event from a scenario (scenario4 in VP1) to another (scenario2 in VP2).

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<OrchestraWorkflow xmlns="http://hp.com/uca/expert/orchestra/config">
    <Routes>
        <Route>
            <COPY>
                <Source>
                    <ValuePackNameVersion>VP1-1.0</ValuePackNameVersion>
                    <ScenarioName>scenario1</ScenarioName>
                </Source>
                <Destinations>
                    <Destination>
                        <Target>
                           <ValuePackNameVersion>VP1-1.0</ValuePackNameVersion>
                           <ScenarioName>scenario2</ScenarioName>
                        </Target>
                    </Destination>
                    <Destination>
                        <Target>
                           <ValuePackNameVersion>VP1-1.0</ValuePackNameVersion>
                           <ScenarioName>scenario3</ScenarioName>
                        </Target>
                    </Destination>
                </Destinations>
            </COPY>
        </Route>
        <Route>
            <JOIN>
                <Sources>
                    <Source>
                        <ValuePackNameVersion>VP1-1.0</ValuePackNameVersion>
                        <ScenarioName>scenario2</ScenarioName>
                    </Source>
                    <Source>
                        <ValuePackNameVersion>VP1-1.0</ValuePackNameVersion>
                        <ScenarioName>scenario3</ScenarioName>
                    </Source>
                    <Source>
                        <ValuePackNameVersion>VP2-1.0</ValuePackNameVersion>
                        <ScenarioName>scenario1</ScenarioName>
                    </Source>
               </Sources>
               <ExpireTime>1000</ExpireTime>
               <Destination>
                    <Target>
                        <ValuePackNameVersion>VP1-1.0</ValuePackNameVersion>
                        <ScenarioName>scenario4</ScenarioName>
                    </Target>
               </Destination>
           </JOIN>
        </Route>
        <Route>
           <COPY>
                <Source>
                    <ValuePackNameVersion>VP1-1.0</ValuePackNameVersion>
                    <ScenarioName>scenario4</ScenarioName>
                </Source>
                <Destinations>
                    <Destination>
                        <Target>
                           <ValuePackNameVersion>VP2-1.0</ValuePackNameVersion>
                           <ScenarioName>scenario2</ScenarioName>
                        </Target>
                    </Destination>
                </Destinations>
            </COPY>
        </Route>
    </Routes>
</OrchestraWorkflow>
```

XML Configuration 9 - Orchestration configuration example

This is the graphical representation of the above example:



6.1.4 Orchestration looping option

In the *OrchestraConfiguration.xml* file several routes between the scenarios can be defined. In most of cases, it is not desired to have loops in the workflow, e.g. to cascade events from one scenario to others and so on, and send back to the initial one. This may lead to an overflow of the memory.

Nevertheless, if the developer wants to define loops in the Orchestra Workflow, it is possible by explicitly setting to true the **uca.ebc.orchestra.loops.allowed** property from the *uca-ebc.properties* file (found in the ${UCA_EBC_INSTANCE}/conf directory). By default, this property is set to false**. If the property is set to false and a loop is detected in the OrchestraConfiguration.xml, an Error is thrown stopping the server**. The error indicates that either the orchestra loops property should be set to false, either loops should be removed from the configuration file.

6.1.5 Scenario mode (STREAM/CLOUD) impact on event cascading

Event cascading is possible whatever the scenario's mode STREAM or CLOUD. In the following, all the combinations of STREAM and CLOUD cascading between two scenarios are presented.

6.1.5.1 Cascading events from STREAM to STREAM or CLOUD

Event cascading from STREAM TO STREAM is supported by both types of routes (COPY and JOIN).

COPY events from STREAM to STREAM or CLOUD

A typical use case for event cascading from STREAM to STREAM or CLOUD is the implementation of a pre-filtering layer scenario. In such a case, a first STREAM scenario is designed to “reduce” incoming traffic. It then focuses only on a specific type of events or on specific technologies (originatingManagedObject filter). This is usually a standard pattern matching like link Down/Up, repeatedEvent, etc...

Then the result of this first filtering scenario is provided to a second scenario which can be STREAM or CLOUD (depending on the needs). This second scenario acts more as a functional block implementing the real business logic, as show in Figure 13.

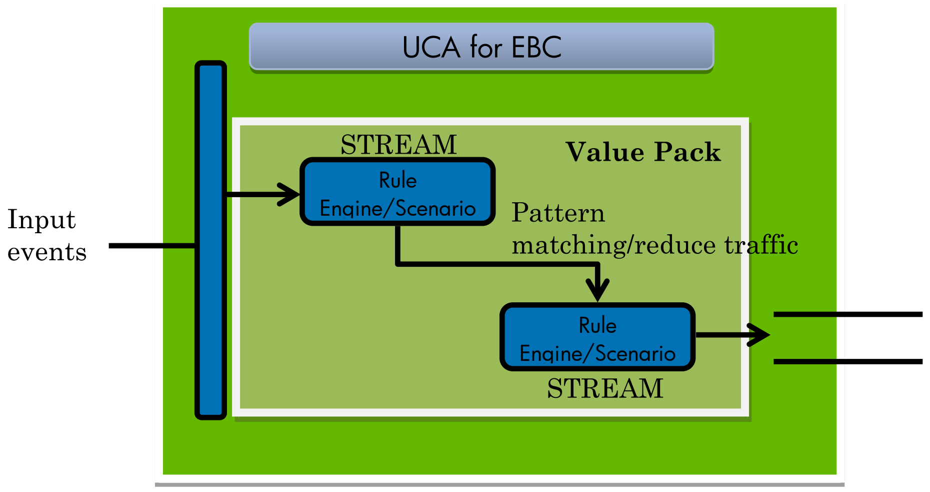


Figure 13 - Cascading from STREAM to STREAM

JOIN events from STREAM to a STREAM or CLOUD

Cascading events through a JOIN route from several **STREAM scenarios to a STREAM** scenario can be done, for example, when a more complex task is divided in multiple layer scenarios. Then, each scenario can add extra information to the event, and the target scenario will receive the event containing all the added information added in the “orchestra data” object.

Cascading events through a JOIN route from several **STREAM scenarios to a CLOUD** scenario is also possible for all event types, but only orchestra data added for the “Alarm” (AlarmCreation type) is conserved when the alarm enters the target scenario’s Working Memory.

When sending other types of event (like AttributeValueChange or AlarmStateChange), the corresponding alarm will be updated in the CLOUD scenario according to the default Alarm lifecycle in CLOUD mode (which is without updating the orchestra data object). If wanting to update the orchestra data added when one of other event types than Alarm are inserted into the working memory, the default Alarm lifecycle in CLOUD mode has to be overwritted.

Depending on the Event instance, the whole lifecycle can be overritted with the method `void doLifecycleProcessing(Event event)` of the **`com.hp.uca.expert.lifecycle.common.LifeCycleExtensionWorkingMemoryAccessAllowed`** interface.

In the case of AttributeValueChange or AlarmStateChange, the following methods of **`com.hp.uca.expert.lifecycle.alarm.AlarmLifeCycleExtensionWorkingMemoryAccessAllowedCloud`** interface have to be overwritted to update also the orchestra data:

- `boolean onUpdateSpecificFieldsFromAttributeValueChange(`

```
AlarmAttributeValueChange alarmAttributeValueChange,
Event alarmInWorkingMemory);
```

-
  ```
  boolean onUpdateSpecificFieldsFromAlarm(Alarm
  newAlarm,Event alarmInWorkingMemory);
  ```

  `boolean onUpdateSpecificFieldsFromStateChange` (AlarmStateChange alarmStateChange, Event alarmInWorkingMemory);

Note

See ☞ 3.3.3 Alarm or Event enrichment and ☞ 3.3.4 Alarm lifecycle for details on the LifeCycle Extension mechanism.

6.1.5.2 Cascading from CLOUD to CLOUD

The CLOUD to CLOUD event cascading use case may be encountered when an important correlation block needs to be split into two to reduce complexity: the second CLOUD scenario provides added-value on top of the first one.

This mode is supported by both types of routes: COPY and JOIN.

6.1.5.3 Cascading from CLOUD to STREAM

The CLOUD to STREAM event cascading use case is supported by both types of routes: COPY and JOIN).

For each Alarm Attribute Value Change (AVC) or Alarm State Change (ASC) cascaded from a CLOUD scenario to a STREAM scenario, the default Alarm Lifecycle is that **the corresponding Alarm creation is created and inserted** into the STREAM scenario's working memory, and only after the AVC or the ASC is inserted.

See ☞ 3.3.4 Alarm lifecycle for details.

For each Alarm Deletion, when the source scenario is in CLOUD mode, **an Alarm Deletion has to be explicity created for continuing applying Orchestration**, whether to a STREAM or to a CLOUD scenario. For this, a

`new AlarmDeletion` has to be created and a call to `com.hp.uca.expert.alarm.AlarmUpdater.replaceAllFields(Alarm,AlarmDeletion)` has to be done in its rule file at the reception of an "aboutToBeRetracted". An example is shown in the following code:

```
rule "Enrichment - [About to be retracted] => Send to
Orchestra"
 when
      a: Alarm (aboutToBeRetracted == true)
 then
    LogHelper.enter(theScenario.getLogger(),
drools.getRule().getName(), a.getIdentifier());

      //have to explicity create an Alarm Deletion for
continuing applying Orchestration
      //when scenario in CLOUD mode
      //for this use AlarmUpdater.replaceAllFields(Alarm,
AlarmDeletion)
      AlarmDeletion generatedAlarmDeletion = new
AlarmDeletion();
      AlarmUpdater.replaceAllFields(a,
generatedAlarmDeletion);
```

```
      theScenario.getLogger().info("Send to Orchestra About
to be retracted\n"+ generatedAlarmDeletion.getIdentifier());
      theScenario.applyOrchestration(generatedAlarmDeletion);
      LogHelper.exit(theScenario.getLogger(),
drools.getRule().getName(),a.getIdentifier());
end
```

This mode is also supported by both types of routes: COPY and JOIN, described in the following sections:

COPY events from CLOUD to STREAM

The CLOUD to STREAM event cascading use case is supported even if it is an unusual use case. Indeed, the CLOUD scenario is generally the end of the correlation chain where the event lifecycle is managed.

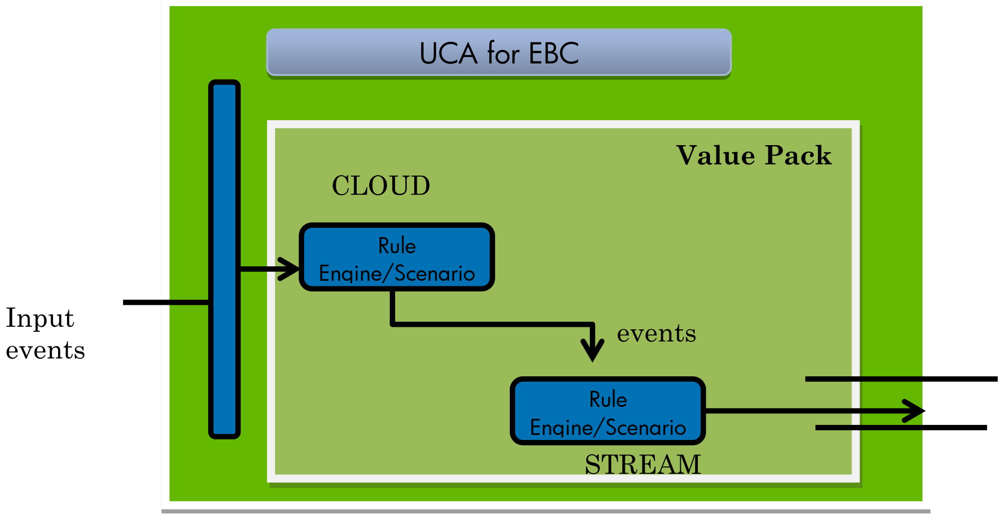


Figure 14 - Cascading from CLOUD to STREAM

JOIN events from CLOUD to STREAM

Cascading events through a JOIN route from **several CLOUD scenarios to a STREAM** scenario is also possible **for all Event types**.

As explained in 6.1.5.1 Cascading events from STREAM to STREAM or CLOUD, for the JOIN events from STREAM to STREAM or CLOUD, the **orchestra data added only for the "Alarm" (AlarmCreation event type) will be updated by default (default Lifecycle)**. Therefore, if the event lifecycle of the source scenarios (in CLOUD mode) is not extended to add orchestra data for other event types then AlarmCreation (like AttributeValueChange, AlarmStateChange, etc.), the STREAM scenario will receive events enriched with orchestra data only for the AlarmCreation type. It will also receive the other events but without the orchestra data enriched by the CLOUD scenario. If the lifecycle in the source CLOUD scenario is updated for the other event types, then the target STREAM scenario will receive enriched events with orchestra data for each event type.

6.1.6 Orchestration API

The UCA for EBC event cascading feature is available through the ***Scenario*** interface:

```
boolean applyOrchestration(Event event);
```

In the case of JOIN routes (Convergence case) there are two methods available through the ***Orchestrable*** interface:

```
void addOrchestraDataInScenario(Serializable orchestraData);
Map<String, Serializable> getOrchestraData();
```

6.1.6.1 Orchestration Methods

Cascading method of the Scenario Interface

Method	Description
`applyOrchestration (Event event)`	Used to cascade an event to other scenarios, according to the workflow defined in the OrchestraConfiguration.xml file. The `applyOrchestration()` method argument must implement the `com.hp.uca.expert.Event` interface. It is the case for the Alarm, AlarmStateChange, AlarmAttributeChange and AlarmDeletion classes. The event argument is cloned by the `applyOrchestration()` method and thus the cascaded object is a completely new Java object that is independent from the one passed as argument. The `applyOrchestration()` method does not perform any action on the passed argument: If this event is in Working Memory it will remain in working memory and will not be retracted by the applyOrchestration method. This method can be called in STREAM or CLOUD mode.

Table 49 - Orchestration method

JOIN (Convergence case) methods of the Orchestrable Interface

Method	Description
`void addOrchestraDataInScenario(Serializable orchestraData)`	This method is to be called by a source scenario of a JOIN route, to add data to the orchestra data object attached to the event. This data must implement the Serializable interface. It is attached to the event in a Map having the calling scenario fullname as key. Note: All primitive types (Integer, String etc …) are Serializable.
`Serializable getOrchestraData()`	This method is to be called from a “convergence scenario” (Destination of a JOIN route) to get the orchestra data. The returned type is a Map<String, Serializable>. Calling this method implies the knowledge Serializable object stored as orchestraData. It must be a contract between the source scenarios and the Destination Scenario of a JOIN Route.
`Serializable getOrchestraData(String scenarioFullName) throws OrchestraException;`	This method, as getOrchestraData(), can be called from a “convergence scenario” (Destination of a JOIN route) to get the orchestraData entry added by a scenario (the key is the scenario full name). It throws an OrchestraException if there is no entry or if orchestraData is empty.
`void resetUuidAndClearOrchestraData();`	This method can be used when wanting to continue applying orchestration on the event, without taking into consideration all the orchestration done before. It resets the internal event UUID (unique identifier used by the Orchestration component) and clears the orchestraData object.
`boolean isConvergenceComplete();`	This method can be used to check if the event (coming from a JOIN route) has the boolean convergence complete true (If all events are received from all source scenarios before the expireTime is reached in the JOIN route) or false if not, as detailed in 6.1.1.2.The JOIN route.

Table 50 - JOIN (Convergence case) methods

6.1.6.2 Example

- **Example of rule using the Orchestra API to apply orchestration**

```
package com.hp.uca.expert.vp.sample;


#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.alarm.AlarmStateChange;
import com.hp.uca.expert.alarm.AlarmAttributeValueChange;
import com.hp.uca.expert.alarm.AlarmDeletion;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.common.trace.LogHelper;

#declare any global variables here
global Scenario theScenario;

declare Alarm
@role( event )
@timestamp( dateTimestamp )
@expires( 60s )
end

rule "Send an Alarm to another Scenario"
when
        # Any alarm inserted in the Working memory
       # has the specific attribute 'justInserted' set to true
        alarm : Alarm (justInserted == true)
then
        # Do something with the alarm, like calling the method defined for
        alarm insertion in Sample.java found in com.hp.uca.expert.vp.sample;

       Sample.newAlarmInsertion(alarm);
       alarm.setJustInserted(false);

        # Retract from  Working Memory
        theScenario.getSession().retract(alarm);

       # Continue the orchestration for this alarm
        theScenario.applyOrchestration(alarm);

        end
```

Rule Sample 27 - Orchestraion API scenario rule example

- **Example of rule using the Orchestra API to add orchestra data**

```
package com.hp.uca.expert.vp.sample;


#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.alarm.AlarmStateChange;
import com.hp.uca.expert.alarm.AlarmAttributeValueChange;
import com.hp.uca.expert.alarm.AlarmDeletion;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.common.trace.LogHelper;

#declare any global variables here
global Scenario theScenario;

declare Alarm
@role( event )
@timestamp( dateTimestamp )
@expires( 60s )
end

rule "Send an Alarm to another Scenario"
when
       # Any alarm inserted in the Working memory
      # has the specific attribute 'justInserted' set to true
       alarm : Alarm (justInserted == true)
then
       # Add some information in the object attached to the alarm
       alarm.addOrchestraDataInScenario ("adding some important info");

       alarm.setJustInserted(false);

       # Continue the orchestrion for this alarm
       theScenario.applyOrchestration(alarm);

       end
```

Rule Sample 28 - Orchestraion API scenario rule for source scenario in JOIN example

- **Example of rule using the Orchestra API to get the orchestra data**

```
package com.hp.uca.expert.vp.sample;


#list any import classes here.
import com.hp.uca.expert.alarm.Alarm;
import com.hp.uca.expert.alarm.AlarmStateChange;
import com.hp.uca.expert.alarm.AlarmAttributeValueChange;
import com.hp.uca.expert.alarm.AlarmDeletion;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.common.trace.LogHelper;

#declare any global variables here
global Scenario theScenario;

declare Alarm
@role( event )
@timestamp( dateTimestamp )
@expires( 60s )
end

rule "Send an Alarm to another Scenario"
when
       # Any alarm inserted in the Working memory
      # has the specific attribute 'justInserted' set to true
       alarm : Alarm (justInserted == true)
then
       # Do something with the modification done by the other scenarios,
        like calling a method using them in Sample.java found in
        com.hp.uca.expert.vp.sample;
       Sample.haveSometingDoneWithOrchestraData(alarm.getOrchestraData());

       alarm.setJustInserted(false);
```

```
    # Retract (or not) the alarm depending if it is still needed(or not)
      in the Scenario Working Memory
    theScenario.getSession().retract(a);

     # Continue the orchestrion for this alarm
    theScenario.applyOrchestration(alarm);

    end
```

Rule Sample 29 - Orchestraion API scenario rule for target scenario in JOIN example

Note

The “Orchestration of Scenarios Cascading” Value Pack delivered with the UCA-EBC 3.2 Server and with the UCA-EBC 3.2 Development Kit provides an example of a Scenario Orchestration with COPY routes.

The “Orchestration of Scenarios Cascading in JOIN Routes” Value Pack delivered with the UCA-EBC 3.2 Development Kit provides an example of a Scenario Orchestration with JOIN routes.

☞ See [R12] Unified Correlation Analyzer for Event Based Correlation – Value Pack Examples for details on each of the Value Packs delivered as examples of the Orchestration of scenarios cascading feature.

6.2 Scenario Specific Configuration

The Scenario Specific Configuration files are optional XML configuration files specific to a given Value pack scenario.

The UCA for EBC product and its Value Pack concept allows any rule developer building a powerful and flexible correlation solution. This flexibility implies a high level of configuration and tunning of the processing of the Value Pack’s scenarios.

A convinient way to configure an application is by using XML files. The JAXB package (part of Java JDK), can help marshalling / unmarshalling the information from XML to Java Object (and vice versa) and make it usable from the Value Pack rules or associated Java code.

The Scenario Specific Configuration feature helps integrating any complex configuration with the UCA for EBC Administration User Interface allowing to read, change and save the configuration from the UCA-EBC Administration Interface.

☞ See [R10] *Unified Correlation Analyzer for Event Based Correlation – User Interface Guide* for more information on how is displayed the Specific Scenario Configuration

A rule developer can use any XML file as a Scenario Specific Configuration file for his Value Pack scenario, provided this file is integrated with JAXB so that the Scenario Specific Configuration can be easily displayed at the UCA for EBC Administration GUI.

To make the UCA-EBC system taking into account a new Scenario Specific Configuration file, the developer needs to give a reference to the JAXB Object representing this configuration (any Java class defining a @XmlRootElement, compatible with JAXB implementation). This is done using the following method from the Scenario interface (usually from a scenario initialization method).

```
void addSpecificConfiguration(Object specificConfiguration);
```

☞ Refer to [R3] *Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Engine* for detailed information.

6.2.1 Setting-up a Value pack specific Configuration file

The following sections explains how to easely create a Specific Configuration file for a sceanrio:

- Defining the XSD of the configuration file.
- Generating the JAXB Binding Java Classes.
- Wraps the Root Binding Java Class to an XmlConfiguration object.
- Integrating these Java Classes into UCA for EBC.

Note

There are multiple ways to use JAXB based information. JAXB allows generating Java code from XSD, or generating an XSD (if needed) from an annoted Java Class. In this example, we are starting from the XSD.

6.2.1.1 Schema definition

This is the starting point when introducing a Scenario Specific Configuration file.

Let's assume that the Scenario behavior must be configured via an XML file compliant with the following XSD file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema xmlns="http://config.pd.vp.expert.uca.hp.com/"
    targetNamespace=http://config.pd.vp.expert.uca.hp.com/
    xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">

    <xs:element name="ProblemPolicies">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="mainPolicy" type="MainPolicy" minOccurs="0"
maxOccurs="1" />
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="MainPolicy">
        <xs:all>
            <xs:element name="candidateVisibilityTimeMode"
                        type="CandidateVisibilityTimeMode"
                minOccurs="1" maxOccurs="1" />
            <xs:element name="candidateVisibilityTimeValue" type="xs:long"
                minOccurs="1" maxOccurs="1" />
            <xs:element name="defaultActionAODirectiveReference" type="xs:string"
                minOccurs="1" maxOccurs="1" />
            <xs:element name="defaultActionScriptReference" type="xs:string"
                minOccurs="1" maxOccurs="1" />
        </xs:all>
    </xs:complexType>

    <xs:simpleType name="CandidateVisibilityTimeMode">
        <xs:restriction base="xs:string">
            <xs:enumeration value="Min" />
            <xs:enumeration value="Max" />
            <xs:enumeration value="Value" />
        </xs:restriction>
    </xs:simpleType>

</xs:schema>
```

XML Configuration 10 - Specific Configuration Schema example

Example of Scenario Specific Configuration XML file complying to this schema:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ProblemPolicies xmlns="http://config.pd.vp.expert.uca.hp.com/">
    <mainPolicy>
        <candidateVisibilityTimeMode>Max</candidateVisibilityTimeMode>
        <candidateVisibilityTimeValue>30000</candidateVisibilityTimeValue>

<defaultActionAODirectiveReference>TeMIP_AO_Directives_localhost</defaultActionAO
DirectiveReference>

<defaultActionScriptReference>Exec_localhost</defaultActionScriptReference>
    </mainPolicy>
</ProblemPolicies>
```

XML Configuration 11 - Specific Configuration XML file example

6.2.1.2 Binding Java classes generation

JAXB is used to generate Binding Java classes from the xsd file.

This can be done in several different ways. We can describe at least two: using maven or the xjc command-line tool.

Using maven plugin:

```
<plugin>
        <groupId>org.jvnet.jaxb2.maven2</groupId>
        <artifactId>maven-jaxb2-plugin</artifactId>
        <version>${maven-jaxb2-plugin.version}</version>
        <configuration>
                <schemaDirectory>src/main/resources/valuepack/conf
                </schemaDirectory>
                <strict>true</strict>
                <verbose>true</verbose>
                <forceRegenerate>true</forceRegenerate>
                <removeOldOutput>true</removeOldOutput>
        </configuration>

                </plugin>
```

Using command line tool:

```
$JAVA_HOME/bin/xjc –p com.hp.mypackage –d src/schema
mySchema.xsd
```

☞ Refer to Oracle XJC user documentation
http://docs.oracle.com/cd/E17802_01/webservices/webservices/docs/1.6/jaxb/xjc.html

Applying the Binding compilation process to the XSD given as example, we obtain a set of Java Classes.

The following Class is the one containing the @XmlRootElement:

```
//
// This file was generated by the JavaTM Architecture for XML Binding(JAXB) Reference
Implementation, vhudson-jaxb-ri-2.1-520
// See <a href="http://java.sun.com/xml/jaxb">http://java.sun.com/xml/jaxb</a>
// Any modifications to this file will be lost upon recompilation of the source schema.
// Generated on: 2012.01.16 at 02:45:50 PM CET
//


package com.hp.uca.example.config;
```

```
import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;


/**
 * <p>Java class for anonymous complex type.
 *
 * <p>The following schema fragment specifies the expected content contained within this
class.
 *
 * <pre>
 * &lt;complexType>
 *   &lt;complexContent>
 *     &lt;restriction base="{http://www.w3.org/2001/XMLSchema}anyType">
 *       &lt;sequence>
 *         &lt;element name="mainPolicy"
type="{http://config.pd.vp.expert.uca.hp.com/}MainPolicy" minOccurs="0"/>
*        &lt;/sequence>
 *     &lt;/restriction>
 *   &lt;/complexContent>
 * &lt;/complexType>
 * </pre>
 *
 *
 */
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "", propOrder = {
    "mainPolicy",
    "problemPolicy"
})
@XmlRootElement(name = "ProblemPolicies")
public class ProblemPolicies {

    protected MainPolicy mainPolicy;
    protected List<ProblemPolicy> problemPolicy;

    /**
     * Gets the value of the mainPolicy property.
     *
     * @return
     *     possible object is
     *     {@link MainPolicy }
     *
     */
    public MainPolicy getMainPolicy() {
        return mainPolicy;
    }

    /**
     * Sets the value of the mainPolicy property.
     *
     * @param value
     *     allowed object is
     *     {@link MainPolicy }
     *
     */
    public void setMainPolicy(MainPolicy value) {
        this.mainPolicy = value;
    }
}
```

XML Configuration 12 - Binding Java Class example

With this set of classes, the rule developer is able to unmarshall an XML file to automatically instanciate these classes with the information stored in the XML document. (It is also possible to generate an XML file from these Java instances).

6.2.1.3 Wraping the JAXB Binding class into a XmlConfiguration object.

In order to make the system taking into account the new Specific Configuration file. A wrapper class implementing the XmlConfiguration object must be provided.

☞ Refer to the XmlConfiguration Javadoc for full details on this class.

This wrapper class can be implemented as follow:

```
package com.hp.uca.example.config;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import com.hp.uca.common.properties.exception.ConfigurationFileException;
import com.hp.uca.common.xml.XmlConfiguration;

public final class MySpecificConfiguration extends XmlConfiguration {

        public static final String RESOURCE_PROBLEM_CONFIG_XML = "valuepack/conf/MyVPConfig.xml";
        private static final Logger LOG = LoggerFactory.getLogger(MySpecificConfiguration.class);

        public MySpecificConfiguration() {
                super();

                setObjectClass(ProblemPolicies.class);

                try {
                        initialize(RESOURCE_PROBLEM_CONFIG_XML);

                        refreshFromFile();

                } catch (ConfigurationFileException e) {
                        LogHelper.logErrorDebug(LOG,
                                        "Invalid Configuration File", e);
                        setWorkFlow(buildDefaultValues());
                }
        }

        public ProblemPolicies getProblemPolicies() {
                return (ProblemPolicies) getTheObject();
        }

        public void setProblemPolicies (ProblemPolicies problemPolicies) {
                setTheObject(problemPolicies);
        }
}
```

XML Configuration 13 – XmlConfiguration Wrapper

6.2.1.4 Scenario Specific Configuration registration

Then the rule developer registers the XmlConfiguration object to the UCA-EBC system. The best location for doing this is from the scenario initialization routine as follow:

```
package com.hp.uca.example.config

import com.hp.uca.common.exception.UcaException;
import com.hp.uca.expert.lifecycle.DefaultScenarioInitialization;
import com.hp.uca.expert.scenario.Scenario;
import com.hp.uca.expert.vp.internal.ValuePackApplicationContext;

public class ScenarioInitialization extends DefaultScenarioInitialization {

        public ScenarioInitialization(Scenario scenario,
                        ValuePackApplicationContext valuePackApplicationContext) {
                super(scenario, valuePackApplicationContext);

                scenario.addSpecificConfiguration(new MySpecificConfiguration() );
        }

        @Override
        public void initializeScenario() throws UcaException {
                Orchestra.initialization(getScenario());

        }
}
```

XML Configuration 14 – Scenario Specific Configuration integration example

At startup, the ValuePack scenario automatically reads the XML file, and the UCA for EBC Administration Interface is able to display the XML Scenario Specific Configuration.

6.3 Persisting alarms into a DB

With UCA-EBC 3.1 it is possible to store alarms in an SQL DB of your choice. This is simply done by some configuration of the value pack.

6.3.1 Configuring the DB persistence feature

You simply need to define multiple Spring beans in the value pack context.xml file:

- The datasource
- The Data Access Objects
- The Alarm forwarder
- The DB notifier

This feature is clearly explained in the Value Pack Development Guide.

Also, you have to define a dbFlow as well (see *Chapter* 3.4.2.2 Defining Collection flows) in the ValuePackConfiguration.xml file.

6.3.2 Accessing DB to retrieve persisted alarms

You need to use the REST API delivered with UCA-EBC 3.1.

This API is explained in appendix.

Appendix A

A.1 UCA for EBC REST API – DB Access

This guide provides technical information about the DB REST API feature introduced in UCA-EBC 3.1. This API is intended to provide easy access to the alarms stored by the DB Alarm Forwarder feature. Most of methods returning alarms handle both XML encoded or JSON encoded alarms. The type used to return alarms depends on the "Accept" header part of the HTTP request.

Indications for reading this guide

keyword	denotes
{baseurl}	the URL to use to access the API.(*)
{identifier}	an identifier of the stored alarm
{field}	a field of an alarm. Can be X733 or custom.
{value}	a value of a field of an alarm.
{session}	an unique session identifier
[1]..[9]	a footnote

(*) This refers to http://your-uca-ebc-server-address:8888/uca-ebc-rest-api/invoke/db

Getting started

To verify that the REST API is up and running, you can do so by connecting to:
http://your-uca-ebc-server-address:8888/uca-ebc-rest-api/

If it returns an error, make sure the REST API is activated on the UCA-EBC server side. This is done by uncommenting the *uca.ebc.rest.api* property in **uca-ebc.properties** file. A UCA-EBC server restart is needed to take the change into account.

Once you are connected to the REST API, a default page will be displayed where you can:
• access to a tests page (where most commands can be executed, except the CREATE and DELETE ones)
• access the WADL (and XML schemas used)

Knowing which are the AlarmDao objects defined

First of all, you need to know which value pack has an AlarmDao object defined. This is done by querying:

GET {baseurl}/list	Return a list of AlarmDao identifiers

An AlarmDao identifier is in the form:
• *name* : in which case it refers to an AlarmDao bean defined in UCA-EBC main applicationcontext.xml
• *name@vpNameVersion* : in which case it refers to an AlarmDao bean defined in the Value Pack context.xml referenced by vpNameVersion

Information retrieved using this command will help end-user to set up correctly the optional parameters [1] and [8] in REST commands below.

Getting stored alarms

Once you know which VP you need to access, you can send following requests:

GET {baseurl}/getids [1] [2] [8]	Return the list of all alarm identifiers stored in the DB
GET {baseurl}/get [1] [2] [8]	Return the list of all alarms along with their fields
GET {baseurl}/get/{identifier} [1] [8]	Return a single alarm
GET {baseurl}/get/{identifier}/{field} [1] [8]	Return a particular field of an alarm

Doing actions on stored alarms

You can do the following actions on a stored alarm by sending following requests:

POST {baseurl}/clear/{identifier} [1] [8]	Set the networkState field to CLEARED
POST {baseurl}/ack/{identifier} [1] [8]	Set the operatorState field to ACKNOWLEDGED
POST {baseurl}/terminate/{identifier} [1] [8]	Set the operatorState field to TERMINATED
POST {baseurl}/set/{identifier}/{field}/{value} [1] [8]	Set any field value
POST {baseurl}/append/{identifier}/{field}/{value} [1] [7] [8]	Append a value to a specific field

Storing new alarms

You can store a new alarm into the DB by sending following request, followed by the AlarmCreationInterface encoded in JSON or XML:

POST {baseurl}/new [1] [8]	Creates the requested alarm

Deleting alarms

You can remove an alarm from the DB by sending following request:

DELETE {baseurl}/del/{identifier} [1] [8]	Deletes the requested alarm

You can also purge alarms that meet a predefined condition. Such a condition is defined in previousy VP context.xml (or in main context)

DELETE {baseurl}/purge [1] [8] [10]	Purges alarms that fill the specified condition.

Registering for DB updates through REST

It is possible to register for DB updates that you will get by polling the DB periodically. No push notification is used in this version.

• First, register your unique session identifier.

• Then, when 200 OK is returned by above method. You can call the poll method 'getupdates' to receive updates. If you have asked for it [5], the first call to this method will send back all alarms stored in DB.

• At the end, do not forget to unregister your session.

GET {baseurl}/register/{session} [3] [5] [6] [8] [12]	Register for notifications service using unique session identifier
GET {baseurl}/getupdates/{session}	Return the notifications (alarms, changes) since last call
GET {baseurl}/unregister/{session} [3] [8]	Unregister your session

Registering for DB updates through NOM adapter

It is possible to register for DB updates that you will through the NOM mediation platform. Alarm updates will be sent as push notifications.

• First, register your NOM mediation flow as defined in the UCA-EBC main configuration.
• Then, when 200 OK is returned by above method. You will receive updates through NOM channel. If you have asked for it [5], the UCA-EBC will send back all alarms stored in DB upon the creation of the flow.
• At the end, do not forget to unregister your NOM mediation flow.

GET {baseurl}/createflow [3] [4] [5] [8] [11] [12]	Create the NOM mediation flow
GET {baseurl}/deleteflow [3] [8] [11]	Delete the NOM mediation flow
GET {baseurl}/resyncflow [3] [8] [11]	Trigger a resynchronization of the NOM mediation flow.

Advanced Requests

The following requests are for advanced users only as they return the fully detailed as stored in the DB, so end-user needs to know exactly what are the meaning of all fields. Those requests return only JSON messages as the internal alarms are not available through JAXB.

GET {baseurl}/getdetails [1] [2] [8]	Return the list of fully detailed alarms
GET {baseurl}/getdetails/{identifier} [1] [8]	Return the fully detailed single alarm
GET {baseurl}/getsessions [9]	Return the list of REST sessions registered

Optional arguments

Footnote	Option	Description	Default
[1]	dao=[name-of-dao]	The DAO bean name as defined in the VP context.xml (or in main context)	"alarmDao"
[2]	since=[when]	A timestamp in milliseconds since alarms have been inserted or updated	"0" meaning that all alarms will be returned
[3]	notifier=[name-of-notifier]	The notifier bean as defined in the VP context.xml	"dbNotifier"
[4]	actionReference=[actionReference]	The action reference as defined in the VP ValuePackConfiguration.xml	"TeMIP_FlowManagement"
[5]	summarize=[true|false]	When true, tells to receive all stored alarms on first poll	"false"
[6]	retention=[time]	Specifies the amount of time in milliseconds for which the updates will be kept. Future getupdates should be done before this time, otherwise the updates will be lost	"600000", meaning 10 minutes
[7]	separator=[string]	Specifies the separator that will be used before appending new value to a field.	" "
[8]	vp=[vp-name-version]	The Value Pack in which the alarmDao is defined	null
[9]	audit=[true|false]	Returns the list of registered sessions identifiers. If true, a full audit of registered sessions will be returned	"false"
[10]	condition=[name-of-condition]	The purge condition bean name as defined in the VP context.xml (or in main context).	"purgeCondition"
[11]	flowName=[name-of-alarm-flow]	The alarm flow name on which to perform the action.	"REST-API driven Flow"

[12]	eligibilityScope=[scope-of-eligible-alarms]	Specifies a Java evaluated boolean expression defining the eligibility of an alarm to pass through at flow resynchronization (or at startup when summarize is "true").	"true"

Examples of use

Here below are few examples using the Linux curl command.

• Try on server with no DB enabled value pack first. List the AlarmDao beans (Here below, we suppose we have not defined an AlarmDao in main context)

```
$ curl http://your-uca-ebc-server-address:8888/uca-ebc-rest-api/invoke/db/list
{"id":[]}
```

• Deploy and start persistence-example. Redo same command

```
$ curl http://your-uca-ebc-server-address:8888/uca-ebc-rest-api/invoke/db/list
{"id":["alarmDao@persistence-example-3.1"]}
```

• List the alarms of the persistence-example

```
$ curl http://your-uca-ebc-server-address:8888/uca-ebc-rest-
api/invoke/db/getids?vp=persistence-example-3.1
{"id":[]}
```

• Generate few alarms. Redo same command.

```
$ uca-ebc-injector -f /var/opt/UCA-EBC/instances/default/deploy/persistence-example-3.1
/scenario/Alarms.xml -r --number 10
$ curl http://your-uca-ebc-server-address:8888/uca-ebc-rest-
api/invoke/db/getids?vp=persistence-example-3.1
{"id":
["CORRELATED-1","CORRELATED-2","CORRELATED-3","CORRELATED-
4","CORRELATED-5","CORRELATED...
```

• Retrieve an alarm

```
$ curl http://your-uca-ebc-server-address:8888/uca-ebc-rest-
api/invoke/db/get/CORRELATED-1?vp=persistence-example-3.1
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
><alarm xmlns:ns2="http://hp.com/uca/expert/
x733Alarm"><ns2:identifier>CORRELATED-1</
ns2:identifier><ns2:sourceIdentifier>TeMIP EMS</
ns2:sourceIdentifier><ns2:alarmRaisedTime>2014-03-
26T16:08:12.544+01:00</
ns2:alarmRaisedTime><ns2:originatingManagedEntity>BOX B1</
ns2:originatingManagedEntity><ns2:alarmType>COMMUNICATIONS_ALARM</
ns2:alarmType><ns2:probableCause>Fire</
ns2:probableCause><ns2:perceivedSeverity>CRITICAL</
ns2:perceivedSeverity><ns2:networkState>NOT_CLEARED</
ns2:networkState><ns2:operatorState>NOT_ACKNOWLEDGED</
ns2:operatorState><ns2:problemState>NOT_HANDLED</
ns2:problemState><ns2:customFields><ns2:customField value="1"
name="AlarmId"/></ns2:customFields></alarm>
```

• Acknowledge alarm

```
$ curl -X POST http://your-uca-ebc-server-address:8888/uca-ebc-rest-api/invoke/db/ack/
CORRELATED-1?vp=persistence-example-3.1
{"oldValue":"NOT_ACKNOWLEDGED","newValue":"ACKNOWLEDGED","name":"ope
ratorState"}
```

• Set a field, append new stuff to it and retrieve the final field value in XML
$ curl -X POST http://your-uca-ebc-server-address:8888/uca-ebc-rest-api/invoke/db/set/CORRELATED-1/additionalText/Test1?vp=persistence-example-3.1
```
{"oldValue":null,"newValue":"Test1","name":"additionalText"}
```
$ curl -X POST http://your-uca-ebc-server-address:8888/uca-ebc-rest-api/invoke/db/append/CORRELATED-1/additionalText/Test2?vp=persistence-example-3.1&separator=,
```
{"oldValue":"Test1","newValue":"Test1,Test2","name":"additionalText"
}
```
$ curl -H "Accept: application/xml" http://your-uca-ebc-server-address:8888/uca-ebc-rest-api/invoke/db/get/CORRELATED-1/additionalText?vp=persistence-example-3.1
```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
><alarmField><name>additionalText</name><value>Test1,Test2</value></
alarmField>
```

• Create new alarm encoded in JSON
$ curl http://your-uca-ebc-server-address:8888/uca-ebc-rest-api/invoke/db/new?vp=persistence-example-3.1 -H "Content-Type: application/json" -d '{"identifier":"CORRELATED-1000","sourceIdentifier":"TeMIP EMS","originatingManagedEntity":"BOX B2","alarmType":"COMMUNICATIONS_ALARM","probableCause":"Fire","perceivedSeverity":"MINOR","networkState":"…
```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?
><result><result>200</result><status>OK</status></result>
```

• Delete that alarm
$ curl -X DELETE http://your-uca-ebc-server-address:8888/uca-ebc-rest-api/invoke/db/del/CORRELATED-1000?vp=persistence-example-3.1
```
{"result":200,"status":"OK"}
```

A.2 UCA for EBC XML schemas

Note: All UCA-EBC configuration File schemas and Alarms (Alarms, AlarmStateChangeInterface, and AlarmAttributeChangeInterface) schemas have been removed from the documentation. They are now available in the *${UCA_EBC_HOME}/schemas* folder.

Glossary

DRL:	Drools Rule file
EVP:	UCA for EBC Value Pack
GUI:	Graphical User Interface
JAXB:	Java Architecture for XML Binding
JMS:	Java Messaging Service
JMX:	Java Management Extension, used to access or process action on the UCA for EBC product.
JNDI:	Java Naming and Directory Interface
Inference engine:	Process that uses a Rete algorithm
LHS:	Left Hand Side, is Drools naming convention for the condition part (when) of a Rule
NMS:	Network Management System
RHS:	Right Hand Side, is Drools naming convention for the action part (then) of a Rule.
SDK:	Software Development Kit
XML:	Extensible Markup Language
XSD:	Schema of an XML file, describing its structure. XSD stands for XML Schema Definition
X733:	Standard describing the structure of an Alarm used in telecommunication environment.