HP Unified Correlation Analyzer

Unified Correlation Analyzer
for
Event Based Correlation

Inference Machine

User Guide

Version 3.3

Edition: 1.0

September 2015

© Copyright 2015 Hewlett-Packard Development Company, L.P.

Legal notices

Warranty

The information contained herein is subject to change without notice. The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

License requirement and U.S. Government legend

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright notices

© Copyright 2015 Hewlett-Packard Development Company, L.P.

Trademark notices

Adobe®, Acrobat® and PostScript® are trademarks of Adobe Systems Incorporated.

HP-UX Release 10.20 and later and HP-UX Release 11.00 and later (in both 32 and 64-bit configurations) on all HP 9000 computers are Open Group UNIX 95 branded products.

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

Microsoft®, Internet Explorer, Windows®, Windows Server®, and Windows NT® are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Oracle® is a registered U.S. trademark of Oracle Corporation, Redwood City, California.

UNIX® is a registered trademark of The Open Group.

X/Open® is a registered trademark, and the X device is a trademark of X/Open Company Ltd. in the UK and other countries.

Red Hat® is a registered trademark of the Red Hat Company.

Linux® is a registered trademark of Linus Torvalds in the U.S. and other countries.

Contents

Preface ... 9

Chapter 1 ... 12

Inference Machine: a quick tour ... 12

1.1 Context ... 12
1.2 Naming disambiguation ... 12
1.3 Basic concepts ... 13
1.3.1 Inference Machine ... 13
1.3.2 Problem Detection ... 14
1.3.3 Topology State Propagator ... 15
1.4 Licensing ... 16

Chapter 2 ... 17

General features ... 17

2.1 Root Cause and Service Impact Analysis ... 17
2.2 Event grouping ... 18
2.3 Life cycle ... 21
2.4 Automatic actions ... 21
2.5 Automatic Trouble Ticketing ... 21
2.6 Cross domain correlation ... 21
2.7 Event enrichment ... 22
2.8 Performance ... 22
2.9 Robustness ... 22
2.10 Ease of use ... 23
2.11 Simulation ... 23

Chapter 3 ... 24

Architecture ... 24

3.1 Inference Machine ... 24
3.2 Problem Detection ... 25
3.3 Topology State Propagator ... 25
3.4 A common library ... 26
3.4.1 Actions Factory ... 27
3.4.2 Life cycle class for states and other events ... 27
3.4.3 Interfaces ... 27

Chapter 4 ... 29

The IM scenarios explained ... 29

4.1 Problem Detection (PD) ... 29
4.1.1 Its role in brief ... 29
4.1.2 Its main features ... 29
4.1.3 Alarm state propagation ... 30
4.2 Topology State Propagator (TSP) ... 30
4.2.1 Its role in brief ... 30
4.2.2 Its main features ... 31

4.2.3 Alarm state propagation 32

Chapter 5 .... **33**

Configuration **33**

5.1 Value Pack 33
5.2 Inference Machine 33
5.2.1 Actions to NMS 34
5.2.2 Trouble Ticket Actions 37
5.3 Problem Detection 39
5.3.1 Filters, tags and mappers 39
5.3.2 Specific configuration 40
5.4 Topology State Propagator 46
5.4.1 Filters, tags and mappers 46
5.4.2 Specific configuration 47
5.5 Orchestra 51

Chapter 6 .... **53**

Developing an IM Value Pack **53**

6.1 Eclipse Plugins 53
6.1.1 Creating a UCA EBC project in Eclipse 53
6.1.2 Creating a Problem Detection only Value Pack 54
6.1.3 Creating a Topology State Propagator only Value Pack 56
6.1.4 Creating an Inference Machine Value Pack 57
6.2 Creating a simple Problem Detection Value Pack 57
6.2.1 Analyzing the problems to be detected 57
6.2.2 Identifying the different types of alarms 58
6.2.3 Configuring the Time Window 58
6.2.4 Configuring Problem Alarm creation 59
6.2.5 Configuring Trouble Ticketing 59
6.2.6 Considering if the default behavior needs to be modified 60
6.2.7 Defining the Filters 60
6.2.8 Configuring Value Pack settings 62
6.2.9 Configuring specific settings 64
6.2.10 Customizing the default behavior for a specific problem 64
6.3 Creating a simple Topology State Propagator Value Pack 65
6.3.1 Analyzing the topology to be used and the propagations to be detected 65
6.3.2 Configuring state computation 66
6.3.3 Identifying the different types of alarms 66
6.3.4 Configuring Trouble Ticketing 67
6.3.5 Considering if the default behavior needs to be modified 67
6.3.6 Defining the Filters 67
6.3.7 Configuring Value Pack settings 71
6.3.8 Configuring specific settings 73
6.4 Creating a Standard IM VP 73

Chapter 7 .... **74**

Advanced features of Problem Detection **74**

7.1 Default behavior 74
7.1.1 Example 74

7.1.2 AlarmRoleCheck 76
7.1.3 EventRoleCheck 76
7.1.4 ProblemAlarmCreation 76
7.1.5 CommonEntityCheck 77
7.1.6 GroupUpdate 79
7.1.7 NetworkStateUpdate 79
7.1.8 OperatorStateUpdate 81
7.1.9 ProblemStateUpdate 83
7.1.10 AttributeUpdate 84
7.1.11 PeriodicCheck 85
7.1.12 AlarmEligibilityUpdate 86
7.1.13 EventEligibilityUpdate 87
7.1.14 TagsHandler 88
7.2 Supported generic events other than alarm types 90
7.3 Computing Problem Information 90
7.3.1 Problem information computing when Problem Detection is topology-aware 91
7.3.2 Problem information computing in default case (non-topology aware) 91
7.3.3 ProblemXmlConfig schema changes 91
7.3.4 ProblemPolicy 92
7.3.5 ProblemDefault.computeProblemEntity(Event event) 92
7.3.6 GeneralBehaviourDefault.computeSourceUniqueId(Event event) 94
7.3.7 ProblemDefault.computeDbRecords(String dbUniqueIdReference, Event event) 95
7.3.8 ProblemDefault.computeGroupPriority(Event event) 96
7.3.9 ProblemDefault.computeTimeWindow(Event event) 97
7.4 Customizing default behavior 97
7.4.1 XML customization 98
7.4.2 Java customization 99
7.4.3 My ProblemDefault 103
7.4.4 Problems initialization in version 3.2 and later 104
7.4.5 MyGeneralBehavior 109
7.4.6 Enrichment 110
7.4.7 MyGeneralBehavior 113

Chapter 8 116

Advanced features of the Topology State Propagator 116

8.1 The default behavior 116
8.1.1 Example 116
8.1.2 Propagation Interface 119
8.1.3 Event Role Check 119
8.1.4 State Creation 119
8.1.5 Service Alarm Creation and Clearance 119
8.1.6 Common Entity Check 119
8.1.7 PropagationGroup update 120
8.1.8 Network State Update 121
8.1.9 Operator State Update 122
8.1.10 Alarm Attribute Update 124
8.1.11 Periodic Check and General Behavior 125
8.1.12 Alarm Eligibility Update 126
8.1.13 State Eligibility Update 127

8.1.14 TroubleTicket update 127
8.2 Computing State 128
8.3 Customizing the default behavior 129
8.3.1 Java customization 129
8.3.2 My PropagationDefault 132
8.3.3 MyGeneralBehavior 137

Chapter 9 139

Troubleshooting 139

9.1 Logging 139

Chapter 10 142

Annexes 142

Annex A. 143

Migration steps 143

Annex B. 148

Problem Detection Value Pack example 148

Annex C. 157

Problem Detection Advanced customization 157

Annex D. 168

Problem Detection Value Pack example with Events only 168

Annex E. 169

Topology State Propagator Value Pack example 169

Annex F. 170

Topology State Propagator Advanced customization 170

Annex G. 171

Inference Machine Value Pack example 171

Tables

Table 1 - Software versions 10
Table 2 - Alarm state propagation from Problem Alarm to Sub-Alarms 30
Table 3 - Alarm state propagation from Sub-Alarms to Problem Alarm 30
Table 4 - IM actions configuration 34
Table 5 - IM action configuration 35
Table 6 – Specific optional IM action configuration for HP TeMIP 36
Table 7 – Specific optional IM action configuration for DB 37
Table 8 – IM troubleTicketActions configuration 37
Table 9 – IM troubleTicketAction configuration 38
Table 10 – Specific optional IM troubleTicketAction configuration for HP TeMIP 38
Table 11 – Tags for possible roles of an event within PD 39
Table 12 – Tags for possible roles of an alarm within PD 39
Table 13 – PD mainPolicy attributes 40
Table 14 – problemPolicy attributes 42
Table 15 – PD problemAlarm per-problem configuration 43
Table 16 – PD troubleTicket “per-problem” configuration 43
Table 17 – PD computeProblemEntityFromFields “per-problem” configuration 45
Table 18 – PD timeWindow “per-problem” configuration 45
Table 19 – PD customized “per-problem” configuration 46
Table 20 – Tags for possible roles of an alarm within TSP 46
Table 21 – TSP mainPolicy attributes 47
Table 22 – TSP serviceAlarm per-propagation configuration 48
Table 23 – TSP troubleTicket “per-propagation” configuration 49
Table 24 – TSP customized “per-propagation” configuration 51
Table 25 – PD: Possible roles for an alarm 62
Table 26 – TSP: Possible roles for an alarm 71
Table 27 - Trigger alarm group priority example 96
Table 28 - Trigger event group priority example 97
Table 29 - Deprecated APIs in IM 3.3 143
Table 30 - Deprecated APIs in PD 3.2 144
Table 31 - Java classes removed in PD 3.3 145
Table 32 - ProblemDefault method changes in PD 3.3 145
Table 33 - ActionsFactory method changes in PD 3.3 146
Table 34 - TroubleTicket method changes in PD 3.3 146
Table 35 - Overrides provided for pd-example problems 150
Table 36 - Example Problem Keys 159
Table 37 - Problem key grouping example 1 159
Table 38 - Problem key grouping example 2 160
Table 39 - Problem key grouping example 3 160

Figures

Figure 1 - Inference Machine Value Pack (RCA-SIA pattern)....13
Figure 2 – RCA-SIA Pattern....17
Figure 3 - Notation conventions....18
Figure 4 - Group (Propagation Group): position of Events....19
Figure 5 - Group already created: example....19
Figure 6 - Propagation Group already created: example....19
Figure 7 - Group to be created: example....20
Figure 8 - Propagation Group to be created is empty....20
Figure 9 – Inference Machine overview....24
Figure 10 – Problem Detection solution architecture....25
Figure 11 – Topology State Propagator solution architecture....26
Figure 12 - Explanation of the candidateVisibilityTimeMode=Max....41
Figure 13 - IM Orchestra configuration example....52
Figure 14 - How to create a UCA EBC project in Eclipse....53
Figure 15 – Create PD only Value Pack....54
Figure 16 - Files to edit to configure MyFirstProblemDetectionValuePack....55
Figure 17 – Create TSP only Value Pack....56
Figure 18 – Create IM Value Pack....57
Figure 19 - Time window illustration....59
Figure 20 - Alarm clearance sequence diagram example....75
Figure 21 – PD Alarm clearance example: PD group updates Step1....75
Figure 22 - PD Alarm clearance example: PD group updates Step2....76
Figure 23 - computeProblemEntity (Event event)....78
Figure 24 - computeProblemEntity (Alarm alarm)....79
Figure 25 - Alarm network state changes....80
Figure 26 - Alarm operator state changes....83
Figure 27 - Periodic checks....85
Figure 28 - Alarm eligibility update....86
Figure 29 - Event eligibility update....87
Figure 30 - Tags handling for computeProblemEntity()....88
Figure 31 - Tags handling for computeGroupPriority(Event)....89
Figure 32 - Tags handling for computeTimeWindow(Event)....90
Figure 33 - One problem specific customization....99
Figure 34 - Consolidation of alarm's qualifiers....103
Figure 35 - MyProblemDefault: a customization for a group of problems....104
Figure 36 – PD MyGeneralBehavior name matching....109
Figure 37 – TSP MyGeneralBehavior name matching....114
Figure 38 - Alarm termination sequence diagram example....117
Figure 39 - Topology of the example....117
Figure 40 - TSP: Alarm termination example: TSP group updates Step1....118
Figure 41 - TSP: Alarm termination example: TSP group updates Step2....118
Figure 42 - Alarm networks state change flow....121
Figure 43 - Alarm operator state change flow....123
Figure 44 - Periodic check and general behavior....125
Figure 45 - Alarm eligibility update....126
Figure 46 - State eligibility update....127
Figure 47 – One propagation specific customization....130
Figure 48 - MyPropagationDefault: a customization for a group of propagations....133
Figure 49 – TSP MyGeneralBehavior name matching....137
Figure 50 - Selecting the XSLT transformation file....147
Figure 51 – pd-example src/main/java directory contents....149
Figure 52 - pd-example src/test/java directory contents....151
Figure 53 - pd-example src/main/resources directory contents....153
Figure 54 - pd-example src/test/resources directory contents....154
Figure 55 - Implementation schema of the main Problem Detection interfaces....158

Preface

This guide describes how to use the HP Unified Correlation Analyzer (HP UCA for EBC Inference Machine solution.

Product name: UCA for EBC Inference Machine embeds two licensed products: UCA EBC Problem Detection and UCA EBC Topology State Propagator.

Product version: 3.3

Kit version: V3.3

Intended audience

This guide is primarily for developers (HP customers or HP consultants) who want to understand an HP UCA for EBC Inference Machine Value Pack containing Problem Detection and Topology State Propagator scenarios.
This document can be also interesting for anyone who want to know more about Inference Machine features.

Prerequisites

It is highly recommended to have some basic knowledge of HP UCA for EBC before reading this document.

The reader is advised to consult Chapter 1 and Chapter 2 of "HP UCA for Event Based Correlation – Reference Guide" and **"**HP UCA for Event Based Correlation – Value Pack Development Guide".

Typographical conventions

Courier font:

- Source code and examples of file contents
- Commands that you enter on the screen
- Path names
- Keyboard key names

Italic text:

- File names, 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 Reference Guide*

[R2] *Unified Correlation Analyzer for Event Based Correlation Value Pack Development Guide*

[R3] *Unified Correlation Analyzer for Event Based Correlation Installation Guide*

[R4] *Unified Correlation Analyzer for Event Based Correlation User Interface Guide*

[R5] *Unified Correlation Analyzer – Clustering and HA Guide*

[R6] *UCA for EBC Inference Machine – JavaDoc (%UCA_EBC_DEV_HOME%\apidoc\inference-machine\index.html)*

[R7] *UCA for EBC – JavaDoc (%UCA_EBC_DEV_HOME%\apidoc\uca-ebc\index.html)*

[R8] *Unified Correlation Analyzer for Event Based Correlation Inference Machine Installation Guide*

[R9] *Unified Correlation Analyzer for Event Based Correlation Topology Extension Guide*

[R10] *HP Unified OSS Console Version 1.2.0 – User Guide*

[R11] *UCA for EBC Administration, Configuration and Troubleshooting Guide*

[R12] *Unified Correlation Analyzer for EBC Inference Machine Release Notes*

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 3.3	• HP-UX 11.31 for Itanium • Red Hat Enterprise Linux Server, 64 bits, Release 5.9-5.11, 6.4-6.6, 7.0-7.1
UCA for Event Based Correlation Channel Adapter, version 3.3	• HP-UX 11.31 for Itanium • Red Hat Enterprise Linux Server, 64 bits, Release 5.9-5.11, 6.4-6.6, 7.0-7.1
UCA for Event Based Correlation Software Development Kit, version 3.3	• Windows 7 64 bits • Red Hat Enterprise Linux Server, 64 bits, Release 5.9-5.11, 6.4-6.6, 7.0-7.1
UCA for Event Based Correlation Inference Machine Kit, version 3.3	• Windows 7 64 bits • Red Hat Enterprise Linux Server, 64 bits, Release 5.9-5.11, 6.4-6.6, 7.0-7.1

Table 1 - Software versions

Support

Visit the HP Software Support Online website at https://softwaresupport.hp.com/ for contact information, and for details about HP software products, services, and support.

The software support area of the website includes the following:

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

Chapter 1

Inference Machine: a quick tour

1.1 Context

HP UCA for EBC is an expert system, which provides an embedded Inference Engine that end-users can complement with their own knowledge base of rules to execute.

HP UCA EBC Inference Machine is a framework based on top of HP UCA for EBC to deliver high-value Value Packs with an embedded knowledge base where end-users do not need to write rules.

An Inference Machine Value Pack is at a first glance very generic but is highly configurable to fit most of end-user needs.

The Inference Machine Development Kit is aimed at building an HP UCA for EBC Value Pack for the Root Cause Analysis or Service Impact Analysis pattern for all kinds of network elements.

1.2 Naming disambiguation

The term "*Inference Machine*" has different meanings in different contexts. It can refer to:

- Inference Machine Development Kit (IM SDK):
  The Eclipse environment (including plug-ins) to develop an Inference Machine Value Pack. The Inference Machine Development Kit is an addition to the HP UCA EBC Development Kit.
- Inference Machine Value Pack (IM VP):
  An HP UCA EBC Value Pack built using the Inference Machine Development Kit, including its libraries.

The term "*Problem Detection*" has different meanings in different contexts. It can refer to:

- Problem Detection framework (PD framework):
  The set of libraries, rules, and configuration files used to develop and run a Problem Detection Value Pack. This framework is delivered as part of the HP UCA EBC Inference Machine Development Kit and can be packaged into any Problem Detection Value Pack.
- Problem Detection Value Pack (PD VP):
  An Inference Machine Value Pack using only the Problem Detection framework.

The term "*Topology State Propagator*" has different meanings in different contexts. It can refer to:

- Topology State Propagator framework (TSP framework):
  The set of libraries, rules, and configuration files used to develop and run a

Topology State Propagator Value Pack. This framework is delivered as part of the HP UCA EBC Inference Machine Development Kit and can be packaged into any Topology State Propagator Value Pack.

- Topology State Propagator Value Pack (TSP VP):
  An Inference Machine Value Pack using only the Topology State Propagator framework.

1.3 Basic concepts

1.3.1 Inference Machine

Root Cause Analysis (RCA) is employed to determine the network element that caused the failure as opposed to the network element(s) merely reacting to the failure.

Service Impact Analysis (SIA) is used to determine the impact of such a failure, either on the physical components themselves or on logical services, generally in order to understand the impact on a service contract.

In most cases, a correlation engine is needed to provide root cause and/or service impact analysis.

Within the HP UCA EBC family:

- RCA is covered by the **Problem Detection (PD)** product.
- SIA is covered by the **Topology State Propagator (TSP)** product.

The conjunction of both RCA and SIA is called the **Inference Machine** (**IM**). An IM Value Pack follows the RCA-SIA pattern as shown in

- Figure 1.

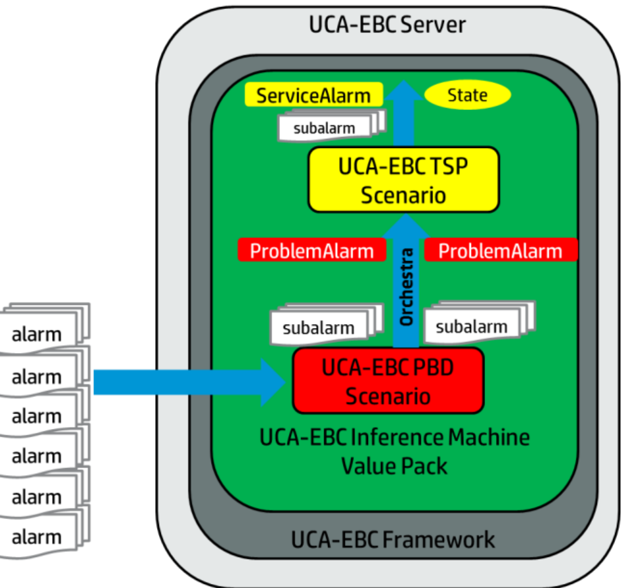


Figure 1 - Inference Machine Value Pack (RCA-SIA pattern)

1.3.2 Problem Detection

The goal of Problem Detection (PD) is to analyze a large number of alarms and, based on a set of conditions, to:

- Realize Root Cause Analysis
- Identify that a problem has occurred and create a Problem Alarm in order to summarize the problem
- Group alarms which are correlated into Sub-Alarms of the Problem Alarm

The main concepts to familiarize with when using PD are problem, alarm grouping, and Root Cause Analysis.

Both PD and TSP are capable of certain automated actions (for example, Trouble Ticket generation and alarms clearance), as well as cross-domain correlation and alarms enrichment.

Whereas in TSP, the topology is mandatory, it is optional in PD so it is not discussed in this section. For details on the topology extension, see the [R9] *Unified Correlation Analyzer for Event Based Correlation Topology Extension Guide*.

1.3.2.1 Problem

The primary role of a PD Value Pack is to identify that a failure (**problem**) has occurred based on the appearance of a certain alarm set and on the presence of certain conditions. Then, an operator readable Problem Alarm is generated to summarize the problem.

1.3.2.2 Problem Alarm

Another base feature of Problem Detection Value Packs is to hide all the Sub-Alarms under the Problem Alarm in the Network Management System (NMS) display. This improves the operator's experience: the most significant alarms stand out in the foreground, and less important alarms are hidden in the background. Note that it is assumed that the NMS has the capacity to group alarms.

When a type of failure (**problem**) occurs in the network on some specific resource at a specific point in time that is called *Tpb* in the current context, equipment in the neighborhood of that resource usually generate several alarms in a time window around *Tpb*.

Problem Detection aims at:

- Detecting such a set of symptom alarms, and identifying the problem that the alarms reveal
- Generating a Problem Alarm that identifies and summarizes the problem, and is readable by the operator
- Grouping symptom alarms (Sub-Alarms) under the Problem Alarm

Such a Problem Alarm generally aggregates:

- Alarms related to network resources in the neighborhood of the network resource(s) that is the source of the problem (same Managed Object, entity hierarchy, or network location)
- Alarms which occurred within a specific time window around *Tpb*

The Problem Alarm is the main alarm handled by operators. Additionally, the Problem Alarm manages the life cycle of the Sub-Alarms grouped under it, with regards to:

- State policy (acknowledgement, termination)

- Clearance policy
- Severity

A PD Group describes a problem and contains important information on:

- The Problem Alarm
- The Sub-Alarms of the Problem Alarms (Sub Service Alarms)
- Candidate Alarm, Trigger Alarm, and Orphan Alarms

Since V3.2 the same applies for event, therefore, in a Group, we can have Candidate Events and Trigger Events.

Trouble Ticket generation can be automated so that each Problem Alarm (including its Sub-Alarms) is handled by just one Trouble Ticket (TT) on the Trouble Ticketing system.

1.3.3 Topology State Propagator

The goal of Topology State Propagator (TSP) is to analyze Root Cause Alarms (usually Problem Alarms grouped by Problem Detection) in order to:

- Realize Service Impact Analysis with multi-layer network elements
- Identify propagations and mark each of them by creating a State that represents the propagation and, optionally, by creating a "Service Alarm" in a NMS, in order to identify the impacted propagation
- Group alarms which are correlated into Sub-Alarms of the Service Alarms

The main concepts to familiarize with when using Topology State Propagator are propagation, alarm grouping, and Service Impact Analysis.

As PD, TSP is also capable of certain automated actions (for example, Trouble Ticket generation and Alarms clearance), as well as cross-domain correlation and alarms enrichment.

Whereas in PD, the topology is optional, it is mandatory in TSP. So the right to use the HP UCA EBC topology extension has to be checked before implementing a TSP use-case.

In a standard way, one TSP scenario is associated with a specific domain (which can be physical or logical).

1.3.3.1 Propagation and state

Propagation in TSP is equivalent to the notion of Problem in PD. Propagation defines an impact on a specific service. The impact is characterized by a state of that service.

Propagation can be triggered by either:

- A Root Cause Event (usually a Problem Alarm coming from PD)
- Another state generated by TSP (for example, a state generated for a sub-service).

The propagation is responsible for creating the state and optionally storing it into a DB, which is possible due to the HP UCA EBC V3.1 DB persistence and DB forwarder features.

Multiple propagations can be defined through the filters file, each top Filter representing one specific propagation.

1.3.3.2 Topology Point of Interest (POI)

The Topology POI is an information utility feature introduced in HP UCA EBC V3.1. It is used in the UCA GUI graph-display tool to track events in the topology tree in real-time. TSP can create POI on a specific node or on a specific relation and is responsible for clearing it if necessary.

1.3.3.3 Service Alarm

Whereas in PD the presence of certain events and conditions is necessary for the creation of a Problem Alarm summarizing the problem, in TSP, the creation of a Service Alarm summarizing the propagation is optional and is based on the presence of certain Root Cause Alarms or states.

The Service Alarm is an alarm that can be created by TSP in a Network Management System (NMS), in order to identify the impacted propagation. It follows the same concerns as the Problem Alarm used in PD.

As PD manages the Problem Alarm, a similar mechanism is implemented in TSP when the Service Alarm feature is enabled. TSP can hide all the Sub-Alarms in the NMS display under the Service Alarm. This improves the operator's experience: the most significant alarms stand out in the foreground, and less important alarms are hidden in the background.

A TSP Propagation Group describes a propagation and contains important information on:

- The state
- The impacting State List
- The Root Cause Alarms
- The whole Sub Tree of Root Cause Alarms (optional)
- The Service Alarm (optional)The Sub-Alarms of the Service Alarm (Sub Service Alarms) (optional)

1.4 Licensing

Inference Machine is a term for two licensed products: UCA EBC Problem Detection and UCA EBC Topology State Propagator.

Chapter 2

General features

2.1 Root Cause and Service Impact Analysis

When a type of failure occurs in the network on some specific resource at a specific point in time that is called *Tpb* in the current context, equipment in the neighborhood of that resource usually generate several alarms in a time window around *Tpb*.

Hence, from those alarms emitted, there is a need to:

- Detect what the problem is behind the failure and summarize it to an operator. This is performed by the Problem Detection (PD) scenario.
- Eventually deduce from the topology of the network what services are impacted by such a failure and summarize them to an operator. This is performed by the Topology State Propagator (TSP) scenario.

Both scenarios run within HP UCA EBC Server as an Inference Machine Value Pack.

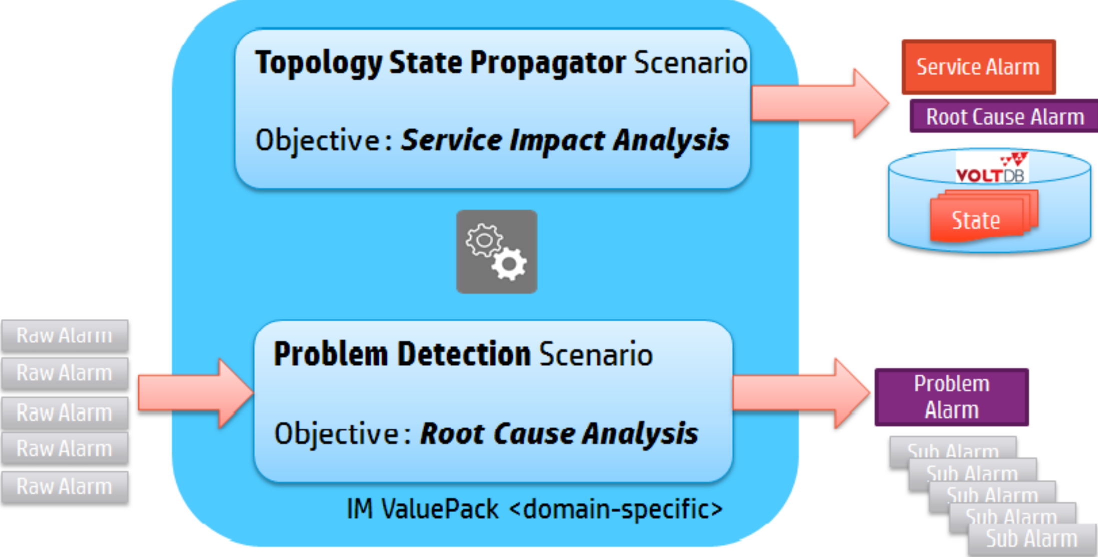


Figure 2 – RCA-SIA Pattern

When the same Network Management System (NMS) is used to handle Problem Alarms (generated by PD) and Service Alarms (generated by TSP), the alarms can be grouped together by the Inference Machine Value Pack, so that the operator is able to navigate from one to the other using, for instance, the HP Unified OSS Console. For more details, see the [R10] *HP Unified OSS Console Version 1.2.0 – User Guide*.

2.2 Event grouping

Both with Problem Detection (PD) and with Topology State Propagator (TSP), a base feature of the Inference Machine Value Packs is that event grouping is possible under a summarized alarm which represents the group Problem Alarm for PD and Service Alarm for TSP, detailed in 1.3.2.2 and 1.3.3.3. As for TSP, the Service Alarm is optional because the grouping is internally represented by the state in TSP.

For PD, problem grouping generates the creation of Groups. The same principle is valid for TSP, where propagation grouping generates the creation of Propagation Groups.

Several schemas and diagrams describe the event grouping concepts in this document. Figure 3 shows the notations that are used

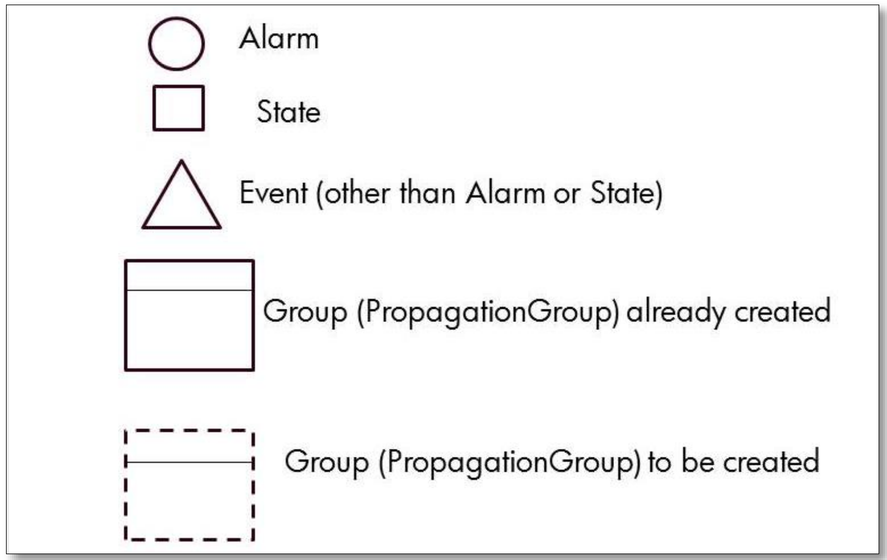


Figure 3 - Notation conventions

Depending on their position in the Group and Propagation Group:

- A state can be the state representing the Propagation Group or an impacting state of it.
- An alarm can be the Problem Alarm of a Group, the Service Alarm of a Propagation Group, or a Sub-Alarm in the case of Group and Propagation Group, and a Root Cause Alarm in the case of a Propagation Group.
- An event can be a Trigger Event or a Sub-Event of a Group.

These concepts are explained in the following figures.

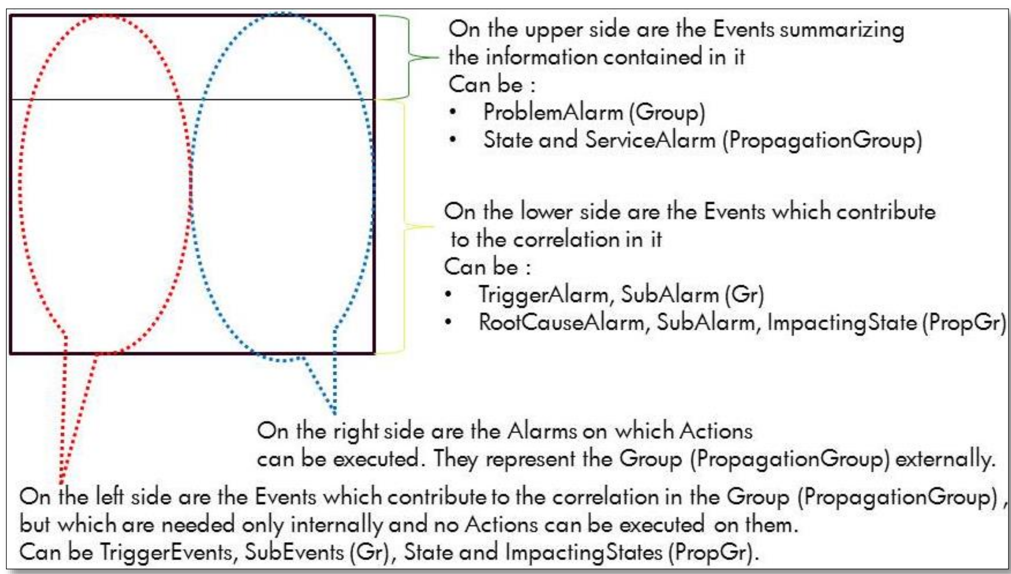


Figure 4 - Group (Propagation Group): position of Events

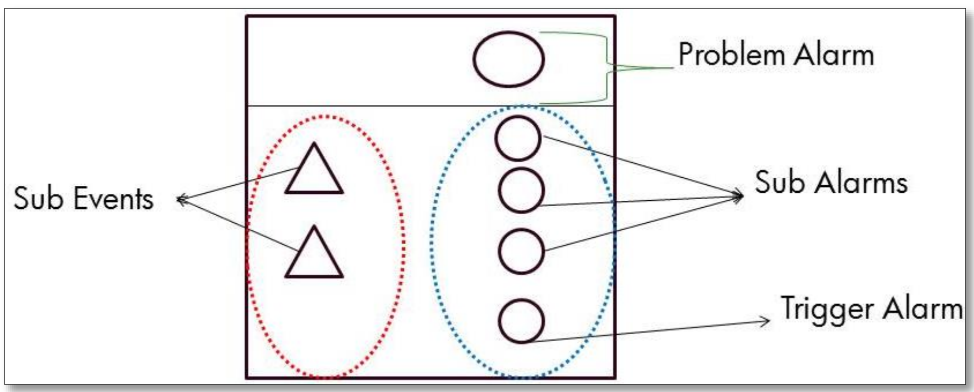


Figure 5 - Group already created: example

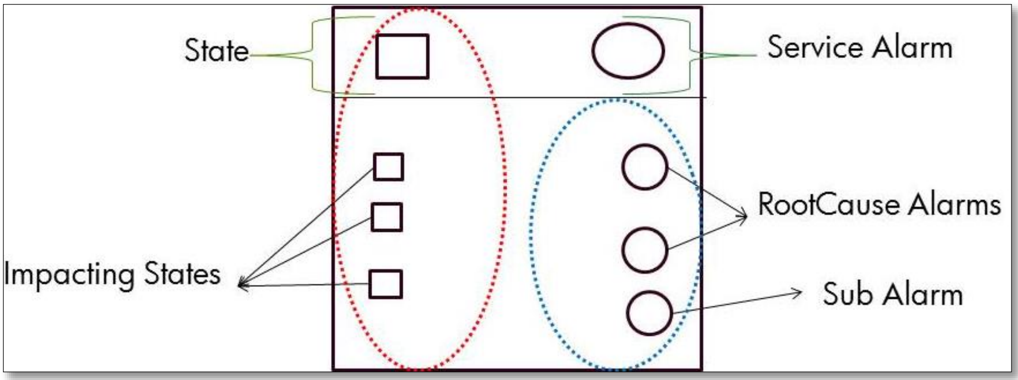


Figure 6 - Propagation Group already created: example

Depending on whether the Group is created or not:

- An alarm can be a Candidate Alarm of a Group
- An event can be a Candidate Event of a Group.

Figure 7 shows an example of a Group to be created and its events and alarms which will contribute to the correlation in the group, set for the moment as Candidate.

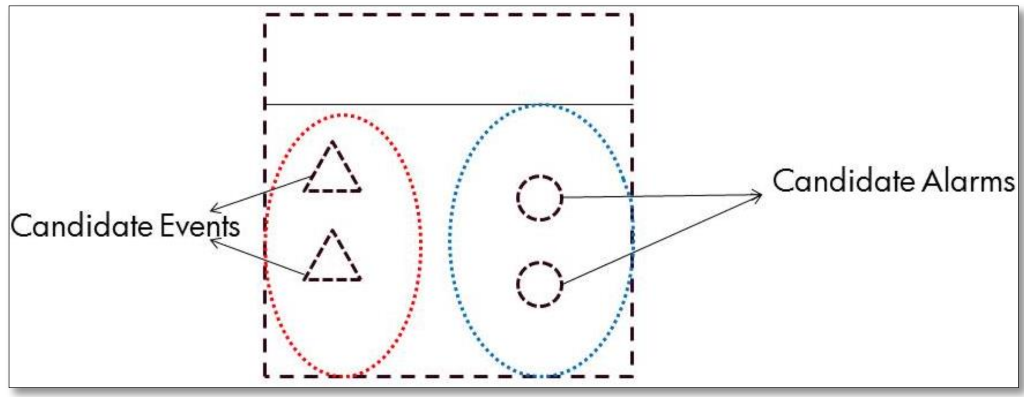


Figure 7 - Group to be created: example

In comparison with the Group, in the Propagation Group, there is no notion of Candidate Event or Candidate Alarm. Therefore, the Propagation Group to be created is empty. As soon as the creation of a propagation group is set questioned by the framework, its state is computed and the Propagation Group is created. So the notation of the Propagation Group to be created is empty.

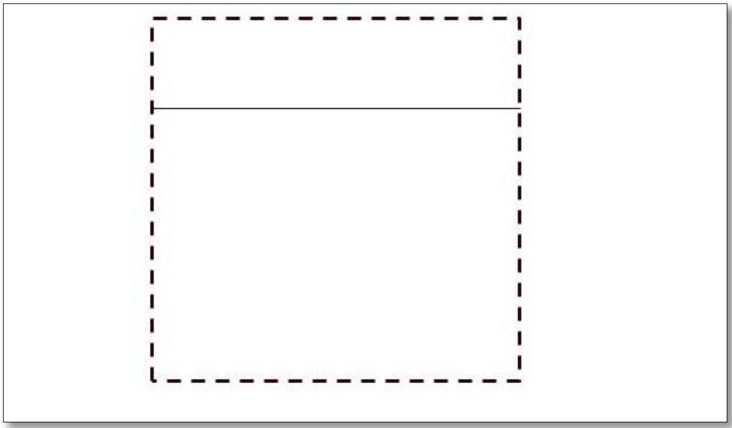


Figure 8 - Propagation Group to be created is empty

In brief:

- The PD scenario can hide all the Sub-Alarms in the Network Management System (NMS) display under the Problem Alarm.
  Since V3.2, PD is also able to group events (not necessarily alarms).For more details, see Annex D.
- The TSP scenario can aggregate state and/or Root Cause Alarms that impact the same service under the same group.
  TSP is able to group Root Cause Alarms in the Network Management System (NMS) display under a single Service Alarm, if the same NMS is used.

Hence, this improves the operator's experience: the most significant alarms stand out in the foreground, and less important alarms are hidden in the background.

Users can navigate from Root Cause view to Service view in their console of choice, for example, in the HP Universal OSS Console. For more details, see [R10] *HP Unified OSS Console Version 1.2.0 – User Guide*.

2.3 Life cycle

Both the PD and TSP frameworks packed in the IM come with default alarm and events life cycle, as well as with a default behavior.

In case the default behavior needs to be enhanced, the Value Pack developer can write the custom code in overridable methods or through configuration when available.

The appropriate overridable methods are called depending on the life cycle of the alarm, state, or other event and depending on the Problem or Propagation contexts.

Both PD and TSP frameworks automatically invoke the methods whatToDoWhenXXX(…), at specific times of the life cycle of every alarm, state, or other event.

2.4 Automatic actions

Besides noticing and reporting problems, and grouping events, Inference Machine scenarios can execute other automatic actions with respect to the life cycle of alarms (alarm state propagation from Problem or Service Alarm to Sub-Alarms and the other way round) and with respect to Trouble Tickets (creation and propagation).

The automated actions, common to PD and TSP, are done using the Actions Factory detailed in 2.5 Automatic Trouble Ticketing.

2.5 Automatic Trouble Ticketing

Trouble Ticket generation can be automated so that each Correlation Alarm (Problem or Service) can be handled by just one Trouble Ticket (TT) on the Trouble Ticketing system.

This can be done independently or simultaneously on the following two scenarios:

- On Problem Detection (PD) to associate a Problem Alarm and its Sub-Alarms to a single TT
- On Topology State Propagator (TSP) to associate a Service Alarm and its Root Cause Alarms (coming from Problem Detection) or Sub Service Alarms to a single TT
- .

2.6 Cross domain correlation

PD scenarios, as all HP UCA for EBC Value Packs, are able to process alarms coming from various Network Management Systems (NMS) through the mediation layer. The same applies to TSP scenarios, which by providing the Service Impact Analysis (SIA) function complete the RCA-SIA pattern in the IM. Therefore, the standard IM Value Pack contains one PD scenario which usually sends its grouped Problem Alarms to the TSP scenario.

Without developers having to write any Java code, both PD and TSP frameworks are able to send actions to HP TeMIP, and are able to interact with the HP Service Manager Trouble Ticketing system through HP TeMIP.

Because HP UCA EBC has been designed as an independent platform, it is capable of receiving alarms and sending actions to other third-party Network Management Systems and Trouble Ticketing or Incident Management Systems. This applies to the PD and TSP frameworks too because they are layered on top of the HP UCA EBC framework in the IM package.

PD and TSP in IM offer an open API available to support:

- Any Network Management System (in addition to HP TeMIP)
- Any Trouble Ticketing System (in addition to HP Service Manager)

The support of additional Network Management Systems and Trouble Ticketing Systems are done through the new Unified Mediation Bus (UMB) introduced in UCA for UBC version 3.3 or through the HP OSS Open Mediation.

Following is an example of a PD use case where cross correlation can be useful:

Consider a situation where all the alarms concerning a GSM network of a telecom company in country 1 are managed with Network Management System A and the alarms concerning a fixed network of the same telecom company in country 2 are managed with Network Management System B.

If the call services from country 1 to country 2 are not working anymore, a well configured Problem Detection Value Pack is able to correlate alarms from Network Management System A with alarms from Network Management System B.

2.7 Event enrichment

If some of the alarms received from the Network Management System (NMS) do not contain enough information to be correlated, both the PD and TSP frameworks offer two pre-formatted ways to get additional data:

- A synchronous way to extract data from an XML file
- An asynchronous way to get data, through the execution of an action (through standard actions that can be customized)

In addition, it is possible to write Java code doing any synchronous or asynchronous request (database access, file access, HTTP request, and so on).

2.8 Performance

Compared to a standard HP UCA for EBC Value Pack developed to perform correlation, an Inference Machine Value Pack is likely to perform significantly better. The reason is that the Inference Machine framework uses optimization based on several hash maps, which allows processing of subsets of relevant alarms rather than blindly feeding the rules engine with whole sets of alarms.

The performance of Problem Detection Value Packs in terms of processing time is close to being a linear function of the number of alarms, whereas in the case of standard HP UCA for EBC Value Packs (performing the same type of correlation) the processing times are likely to be a quadratic function of the number of alarms.

2.9 Robustness

One of the greatest advantages of the Inference Machine is its robustness.

All PD or TSP Value Packs use the fixed set of rules provided by the PD and TSP frameworks, respectively. This fixed set of rules has been extensively tested to ensure good performance and a sound behavior (that is, predictable results).

The developer of either an IM (PD + TSP) Value Pack, or of just a PD or TSP Value Pack neither has to worry about the rules nor the performance of the Value Pack.

However, an important size of memory for the JVM must be foreseen, depending on the numbers of resident alarms in the Working Memory.

2.10 Ease of use

The steps to create a PD or TSP Value Pack are simple and short.

If you are satisfied with the default behavior of PD or TSP scenarios, the creation of an IM Value Pack does not require any Java coding or rule writing. It only requires modifying some XML configuration files.

2.11 Simulation

Through a simple process, it is possible to check the correctness of an Inference Machine Value Pack before actually building and deploying it.

Developing an Inference Machine Value Pack does not involve writing correlation rules. Nevertheless, it is highly recommended to unit test your code prior to kit generation and deployment.

Another advantage of IM is that it is easy to write and run simple test files, simulating the injection of alarms to validate that the problems are detected correctly, and that the behavior of the Value Pack is as expected.

Chapter 3

Architecture

3.1 Inference Machine

HP UCA EBC provides a correlation engine based on incoming events but this capability might not be sufficient for end-users who need a more complete events analysis solution.

Inference Machine is the cornerstone for achieving this extended capability. It provides an RCA-SIA pattern that is designed to fit any customer needs. Users of Inference Machine do not need to write correlation rules, but they need to provide configuration files and/or some customization of Java classes.

Inference Machine is composed of two scenarios running in an HP UCA EBC server

- Problem Detection (PD) for doing Root Cause Analysis
- Topology State Propagator (TSP) for doing Service Impact Analysis

Raw alarms coming from any source (usually NMS) are handled by PD, which groups them and generates correlated Root Cause Alarms. The RCAs are forwarded to TSP which groups them to analyze the impacts of the network topology and to generate Service Alarms.

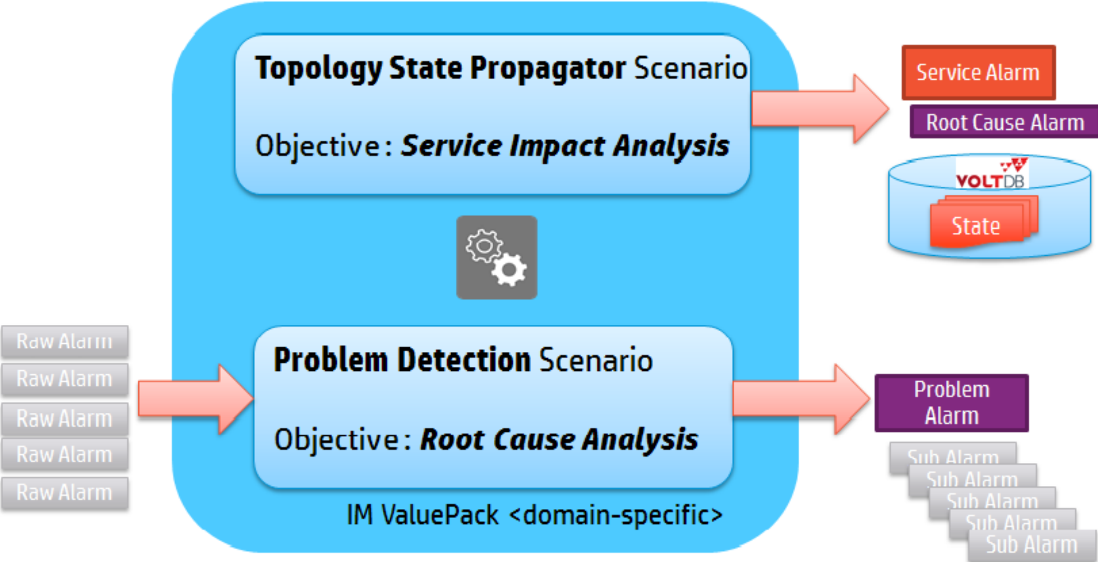


Figure 9 – Inference Machine overview

3.2 Problem Detection

The diagram below shows a Problem Detection Value Pack (PD VP) deployed on an HP UCA for EBC Server. Several Network Management Systems are connected to the HP UCA for EBC Server through a mediation layer.

The PD scenario receives its alarms through the Alarm Collection flow coming from one or several of the Network Management Systems. It can also receive alarms directly from other scenarios through the HP UCA EBC Orchestra component.

The Actions (to create Problem Alarms, to group Sub-Alarms under the Problem Alarm, and so on) use Action Service and are routed and processed by the proper Network Management System.

Contrary to other HP UCA for EBC Value Packs, a PD scenario does not allow its developer to modify the set of rules as they are embedded in the PD framework.

However, PD provides a set of Java methods that the developer can use to control the life cycle of events, the Problem Alarm creation, and so on, within the PD VP. This is called Customization in Figure 10 – Problem Detection solution architecture.

The filters can, as per any other HP UCA EBC VP, be tuned directly by end-user. For more details, see [R1] *Unified Correlation Analyzer for Event Based Correlation Reference Guide*.

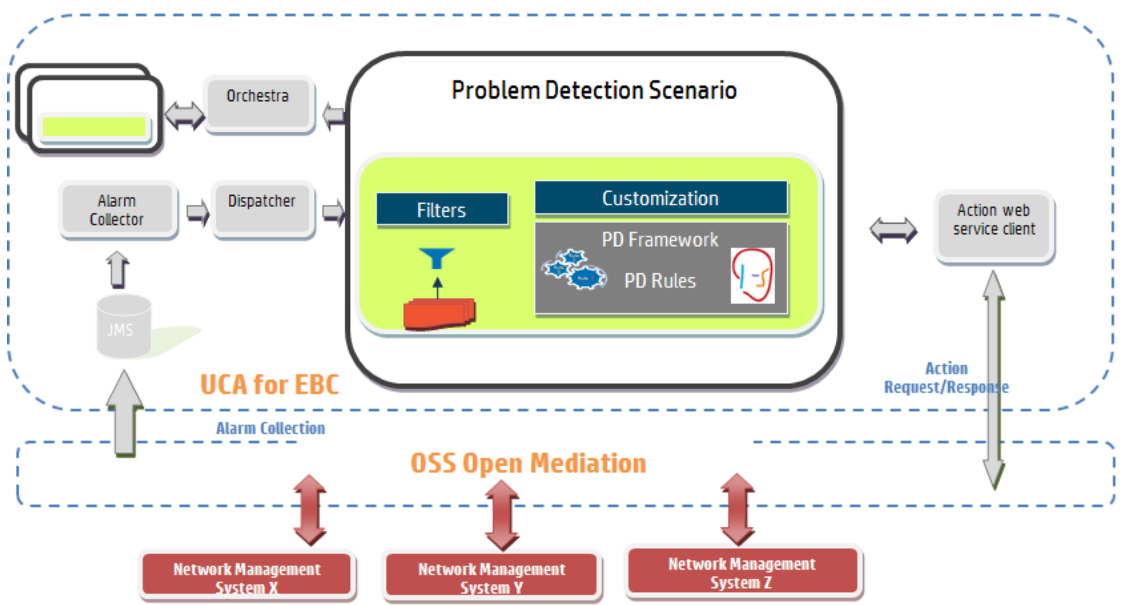


Figure 10 – Problem Detection solution architecture

3.3 Topology State Propagator

The diagram below shows a Topology State Propagator Value Pack (TSP VP) deployed on an HP UCA for EBC Server. Several Network Management Systems are connected to the HP UCA for EBC Server through a mediation layer.

The TSP scenario receives its alarms directly from other scenarios (for example, from PD) through HP UCA EBC Orchestra. However, it can also receive alarms

through HP UCA EBC Alarm Collection flow coming from one or several of the Network Management Systems through the mediation layer.

In order to find out what the impacted services are, a topology describing the network elements (that is, links and nodes) must be defined using HP UCA EBC Topology Extension.

The Actions (to create Service Alarms, to group Sub-Alarms under the Service Alarm, and so on) use HP UCA EBC Action Service and are routed to the proper Network Management System where they will be executed.

Similarly to the PD scenario, a TSP scenario does not allow its developer to modify the set of rules as they are embedded into TSP framework.

However, The TSP framework provides a set of Java methods that the developer can use to control the life cycle of specific events (that is, alarms or states), the Service Alarm creation, and so on. This is called Customization in Figure 11 – Topology State Propagator solution architecture.

The filters for defining the propagations can be tuned directly by end-users. For more details, see [R1] *Unified Correlation Analyzer for Event Based Correlation Reference Guide*.

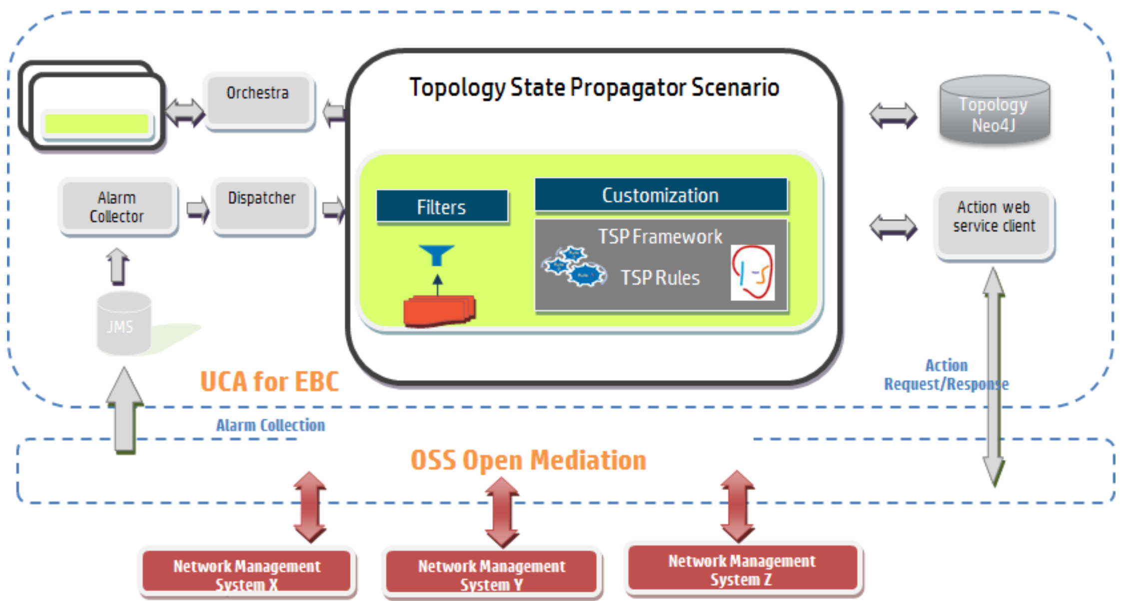


Figure 11 – Topology State Propagator solution architecture

3.4 A common library

As PD and TSP have several common needs, a common library is provided, which is delivering its own namespace.

The common library of the IM (***uca-evp-im-common.jar***) contains the Actions Factories, a common life cycle class for state events, as well as several interfaces, described in the following sections.

3.4.1 Actions Factory

Both TSP and PD need to execute actions on NMS (for example, to create alarms or group alarms). Therefore, the Actions Factory is provided as part of the ***uca-evp-im-common.jar*** common library. The same applies to the access to the database (DbActionsFactory.class is provided).

The Inference Machine developer can configure and use a single Actions Factory for both PD and TSP scenarios in the same Value Pack.

As the new Actions Factory has a different namespace, **the compatibility is broken in PD V3.2**. PD does not provide any automatic migration tool for the Java files. However, SDK provides an XLST (eXtensible Stylesheet Language Transformation) file that can be used to migrate PD configuration files. For more details, see Annex A

The advantages of the new Actions Factory include:

- The logic of Actions is separated from PD and TSP.
- It is reusable: the same ActionsFactory or DbActionsFactory can be used across PD and TSP.
- It is easier to understand.

3.4.2 Life cycle class for states and other events

The class *com.hp.uca.expert.vp.common.lifecycle.MixEventsAndStateLifeCycleExtended.class* is added in the ***uca-evp-im-common.jar*** common library. This class is an enriched alarm life cycle class, managing the life cycle of states, alarms, and others events. Alarms passing just the top filter "*ReservedForGeneralBehavior*" are not inserted in the Working Memory.

For the IM Value Pack, there are two new classes extending this common class:

- *com.hp.uca.expert.vp.pd.im.lifecycle.InferenceMachineLifeCycleExtended* is used as the life cycle class for the PS scenario in an Inference Machine Value Pack. This class handles alarms, events, and states life cycle and it bypasses Service Alarms received from the network.
- *com.hp.uca.expert.vp.tp.im.lifecycle.InferenceMachineLifeCycleExtended* is used as the life cycle for the TSP scenario in an Inference Machine Value Pack. This class handles alarms, events, and states life cycle.

For an IM VP example, see Annex F.

3.4.3 Interfaces

In the common library, several interfaces are included for:

- Actions and Trouble Ticketing
- Common configurations of a problem or a propagation (Booleans, Longs, Strings)
- Problem and Service Alarm creation and History Navigation
- Topology tags definition in filters of Neo4J Cypher Queries
- General Behavior of a problem or a propagation for common methods to all propagations or problems

Full documentation of methods is available in the IM Javadoc part of the SDK [R6]. Most of above interfaces have a default implementation which is used implicitly by ProblemDefault or PropagationDefault Java classes.

Chapter 4

The IM scenarios explained

Two scenarios are included in the Inference Machine:

- Problem Detection (PD)
- Topology State Propagator (TSP)

4.1 Problem Detection (PD)

4.1.1 Its role in brief

In short, Problem Detection (PD) is responsible for Root Cause Analysis.

Problem Detection aims at:

- Detecting a set of symptom alarms from a numerous number of raw alarms, and identifying the problem that the alarms reveal
- Generating a Problem Alarm that identifies and summarizes the problem, and is readable by the operator
- Grouping symptom alarms (Sub-Alarms) under the Problem Alarm

Such a Problem Alarm generally aggregates:

- Alarms related to network resources in the neighborhood of the network resource(s) that is the source of the problem (same Managed Object, entity hierarchy, or network location)
- Alarms which occurred within a specific time window around T_{pb}

The Problem Alarm is the main alarm handled by operators. Additionally, the Problem Alarm manages the life cycle of the Sub-Alarms grouped under it, with regards to:

- State policy (acknowledgement, termination)
- Clearance policy
- Severity

The Network Management System (NMS), which initially displays a constellation of alarms, is instructed by the PD Value Pack to display only a relevant Problem Alarm, and to group and hide all correlated Sub-Alarms beneath it. Note that it is assumed that the NMS has the capacity to group Alarms.

4.1.2 Its main features

The primary role of a Problem Detection (PD) scenario is problem Identification:

- Identifying that a failure (problem) has occurred based on the appearance of a certain set of alarm, and on the presence of certain conditions
- Generating an operator readable Problem Alarm that summarizes the problem

4.1.3 Alarm state propagation

Problem Detection offers the following default behaviors.

When a Problem Alarm's state is changed to	Change Sub-Alarms' state to
ACKNOWLEDGED	ACKNOWLEDGED
NOT_ACKNOWLEDGED	NOT_ACKNOWLEDGED
CLEARED	Sub-Alarms' state left unchanged
CLOSED	Sub-Alarms' state left unchanged
TERMINATED	TERMINATED (If Sub-Alarm was cleared) NOT_ACKNOWLEDGED (If Sub-Alarm was **not** cleared) + "sub-alarms" promoted back to "alarms"
No longer eligible	TERMINATED (If Sub-Alarm was cleared) NOT_ACKNOWLEDGED (If Sub-Alarm was **not** cleared)

Table 2 - Alarm state propagation from Problem Alarm to Sub-Alarms

The eligibility of an alarm to be inserted in the Working Memory or to remain in the Working Memory is determined by the alarm eligibility policy.

The alarm eligibility policy is an expression that evaluates to a Boolean. Below is an example of an alarm eligibility policy:

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

For more details, see the "alarmEligibilityPolicy" chapter in [R1] *Unified Correlation Analyzer for Event Based Correlation Reference Guide*.

When the state of all Sub-Alarms are changed to	Change the state of the Problem Alarm to
CLEARED	CLEARED
No longer eligible	CLEARED

Table 3 - Alarm state propagation from Sub-Alarms to Problem Alarm

4.2 Topology State Propagator (TSP)

4.2.1 Its role in brief

In short, Topology State Propagator (TSP) is responsible for Service Impact Analysis.

In the context of TSP:

- Propagation refers to an impact on an element defined in the network topology, which element is part of multiple assets that usually defines a service. Similarly, propagation is equivalent to problem in Problem Detection.
- State refers to the status of that impact in the topology. For example, a service is degraded but can have different levels of degradation (low, medium, high, and so on)

TSP aims at:

- Detecting from one or more Root Cause Alarms a set of propagations, and identifying the impacts that the propagations reveal
- Generating a state to identify the status of a particular propagation, given that a new propagation can also have impacts on new propagations.
- Generating optionally a Service Alarm that identifies and summarizes the concerned propagation, and is readable by the operator
- Grouping Root Cause Alarms and/or other Service Alarms (as Sub-Alarms) under the Service Alarm

Such a Service Alarm generally aggregates:

- Problem Alarms that have been previously correlated from alarms coming from network equipment (coming from Problem Detection)
- States that have an impact on a specific propagation

The Service Alarm can be the main alarm handled by operators. Additionally, the Service Alarm can also manage the life cycle of the Root Cause Alarms associated with it (and if handled within the same Network Management System), with regards to:

- State policy (acknowledgement, termination)
- Clearance policy

When a hierarchy of propagations is defined, The Network Management System (NMS) is instructed by the TSP Value Pack to display only the top Service Alarm, and to group and hide all sub Service Alarms beneath it. Note that it is assumed that the NMS has the capacity to group alarm.

4.2.2 Its main features

The primary role of a Topology State Propagator (TSP) scenario is to assess the propagation impact:

- Identifying what the impacted services (propagation) are, based on the appearance of certain Root Cause Alarms (previously correlated as Problem Alarms by PD) and based on the description of the impacted network (though Topology API)
- Generating a state defining the status of that impacted service at a given time

Optional features include:

- Creating a Point of Interest (POI) in the Topology in Memory Attribute Manager that is visible through the Graph display application available in HP UCA EBC UI or in HP Unified OSS Console (For more details, see [R10] *HP Unified OSS Console Version 1.2.0 – User Guide.)*
- Generating a copy of that state into a DB for monitoring the historical changes for a specific service
- Creating the Service Alarm that summarizes the propagation in another DB or in the Network Management System (NMS)

4.2.3 Alarm state propagation

Topology State Propagator (TSP) offers the exact same services in terms of Alarm State propagation for Service Alarms that Problem Detection (PD) provides for Problem Alarms.

Chapter 5

Configuration

This chapter covers the configuration of Inference Machine.

5.1 Value Pack

An HP UCA EBC VP is delivered with two configuration files:

- "*ValuePackConfiguration.xml*" defines the configuration used by any HP UCA EBC Value Pack.
- "*context.xml*" defines the spring beans to instantiate within the IM VP.

An IM VP provides these files already configured for running correctly.

The "*ValuePackConfiguration.xml*" file provided by an IM VP has several sections:

<scenarios>	This section **should not be modified** unless upon an HP Support request, or in some rare conditions, where, for example, some periods need to be modified for performance reasons.
<mediationFlows> *<dbFlows>*	These sections **may be modified** to support different NMSs or DBs that must be considered as sources for the IM VP.

The "*context.xml*" file is closely related to the IM VP code itself. Particularly, it can contain the following beans:

"*problemsFactory*"	Present if the PD scenario is defined in the "*ValuePackConfiguration.xml*" file. It **should not be modified.**
"propagationsFactory"	Present if the TSP scenario is defined in the "ValuePackConfiguration.xml" file. It **should not be modified.**

It may also contain the various beans to define the DB connections and the state forwarders to use for storing states and/or Service Alarms into a DB, which may **be modified** to satisfy your DB connection needs.

For more details on the value pack configuration files, see [R1] *Unified Correlation Analyzer for Event Based Correlation Reference* Guide.

5.2 Inference Machine

Actions to NMS and Trouble Ticket Actions are defined specifically in PD and TSP but the way to configure them is common to PD and TSP within the IM framework.

Therefore, this section describes the common configuration parts that can be used by any scenario within the IM framework.

It applies to both the *ProblemXmlConfig.xml* and *PropagationXmlConfig.xml* files.

5.2.1 Actions to NMS

By the default, the IM framework supports two Actions Factories, both of which come with default alarm directives for handling alarms:

- In HP **TeMIP:** in that case <*actionClass*> should be set to *com.hp.uca.expert.vp.common.actions.temip.TeMIPActionsFactory*
- In a **DB:** in that case <*actionClass*> should be set to *com.hp.uca.expert.vp.common.actions.db.DBActionsFactory*

The <*actions*> element contains the following properties:

Name	**Type**	**Value**
defaultActionScriptReference	property	The unique reference that is used in the rule to define the routing information of a script-based Action
Action	property	The container for attributes defining the actions for a set of alarms

Table 4 - IM actions configuration

The <*action*> element contains the following properties:

Name	**Type**	**Value**
Name	attribute	Usually the "sourceIdentifier" field of incoming alarms is matched to this name to know which actionsFactory to use for a given alarm
actionReference	property	The unique reference that is used to get the routing information of an action. This actionReference has to be defined in the Action Registry. The Action Registry is a configuration file used to define routing information for all actions processed by the rules.
actionClass	property	The class implementing the `SupportedAction` interface which describes the methods needed to support any Action on alarms, for example, the `createAlarm`, `terminateAlarm`, and `clearAlarm` methods.

attributeUsedForKeyDuringRecognition	property	The Custom Field Name of the alarm that contains the information to identify that an Alarm is generated by the IM framework. In other words, this attribute defines the name of the field of the Problem Alarm (or Service Alarm) that PD (or TSP) has to look at when the alarms come back from the NMS. PD or TSP uses this field to find the information needed to attach the alarm to the right group. The name of the field is defined in HP UCA EBC format.
attributeUsedForKeyPbAlarmCreation	property	The Custom Field Name of the alarm that contains information about the problem. This attribute defines the name of the field of the Problem Alarm (or Service Alarm), in which PD (or TSP) adds useful information about the problem at the time of the creation of the alarm., for example, the name of the Trigger Alarm, the name of the problem/propagation, or the name of the problem/propagation entity. This information is read by PD or TSP when the alarm comes back from the NMS. The name of the field is defined in NMS format.
Booleans	Property (optional)	Multiple booleans for a specific use-case.
Strings	Property (optional)	Multiple strings for a specific use-case.
Longs	Property (optional)	Multiple longs for a specific use-case.

Table 5 - IM action configuration

The optional *booleans/strings/longs* elements used by TeMIPActionsFactory contain the following properties:

Name	Type	Value
maxChildrenLength	long property	The maximum size in Bytes of the “children” alarm field. The default size is 15000 (15 Kb). When the maximum is reached, Problem Detection stops requesting the NMS to add potential new children to the parent alarm.

useOnlyGroupingKeys	Boolean property	If set to true (the default is false), the GROUPALARM directive is not used. This implies that the “parent” and “children” fields of alarms are not filled. Only the “grouping Keys” field is filled; and the navigation in the HP TeMIP client is only possible through the “Alarms grouping” submenu.
copyReferenceAlarmOnPbAlarmCreation	Boolean property	If set to true (default), the Reference_Alarm directive is always used at Problem Alarm creation. If set to false, the Reference_Alarm directive might not be used at Problem Alarm creation, depending on the value of copyReferenceAlarmWhenNotPbAlarm (see below).
copyReferenceAlarmWhenNotPbAlarm	Boolean property	This field is used only if copyReferenceAlarmOnPbAlarmCreation is set to true (see above). If set to true (default), the Reference_Alarm directive is used at Problem Alarm creation only when the trigger of the new Problem Alarm is not a Problem Alarm created before by PBD. If set to false, the Reference_Alarm directive is never used.
ocName	string property	The value of the OC used.
navigationKey	string property	The navigationKey used during setHistoryNavigation() call. By default, it is set to “Pb”.

Table 6 – Specific optional IM action configuration for HP TeMIP

The optional *booleans/strings/longs* elements used by DBActionsFactory contain the following properties:

Name	**Type**	**Value**
useOnlyGroupingKeys	Boolean property	If set to true (the default is false), the “parent” and “children” fields of an alarm are not updated. Only the “groupingKey” field is filled.
navigationKey	string property	The navigationKey used during setHistoryNavigation() call. By default, it is set to “Pb”.

groupingKey	string property	The name of the groupingKey attribute stored with the alarm. By default, it is set to “groupingKey”.
jdbcAlarmForwarder	string property	The name of the JDBC alarm forwarder bean to use for writing alarms.
sourceIdentifier	string property	The value with which to fill the sourceIdentifier field in createAlarm(). By default, it is set to "UCA-EBC".
dbFlow	string property	The value with which to fill the dbFlow identifier in the targetValuePack field in createAlarm(). By default, it is *null* so that first dbFlow declared in value pack configuration is used.
childPrefix	string property	The prefix to use for each element of the “children” field in the associateAlarmsForHistoryNavigation() call. By default, it is set to “C:DB:".
parentPrefix	string property	The prefix to use for each element of the “parents” field in the associateAlarmsForHistoryNavigation() call. By default, it is set to "MASTER:C:DB:".

Table 7 – Specific optional IM action configuration for DB

5.2.2 Trouble Ticket Actions

The IM framework supports the HP Service Manager through HP TeMIP.

To benefit from it, the <*actionClass*> must be set to *com.hp.uca.expert.vp.common.actions.temip.TeMIPTroubleTicketActionsFactory*.

The <*troubleTicketActions*> element contains the following property:

Name	type	Value
troubleTicketAction	property	The container for attributes defining the trouble ticket actions for a set of alarms

Table 8 – IM troubleTicketActions configuration

The <*troubleTicketAction*> element contains the following properties:

Name	type	Value
Name	attribute	In the filters file, Alarms corresponding to a tag matching this name use the trouble ticket system defined in the actionReference property below.

actionReference	property	The unique reference that is used to define the routing information of a trouble ticket action.
actionClass	property	The class implementing the *SupportedTroubleTicketActions* interface, which describes the methods needed to support any Action on alarms, for example, the createTroubleTicket and the closeTroubleTicket method.
booleans	property (optional)	Multiple booleans for a specific use-case.
Strings	property (optional)	Multiple strings for a specific use-case. The container for a set of key / value <string> specifying parameters for the interaction with the Trouble Ticketing System.
Longs	property (optional)	Multiple longs for a specific use-case.

Table 9 – IM troubleTicketAction configuration

To know which Trouble Ticket System to use for an alarm, the value of the tag is matched to the name attribute of the <*troubleTicketAction*> element.

Example:

tag="TeMIP TT"

<troubleTicketAction name= "TeMIP TT" >

The optional *strings* elements used by TeMIPTroubleActionsFactory:

Name	type	Value
TT_SERVER entity	string property	By default, it is set to "TT_SERVER .SM".
Type	string property	By default, it is set to “Synchronous".
User	string property	By default, it is set to “temip".
CloseTemplateFile	string property	By default, it is set to “closeTroubleTicketByValueRequest.xml".
CreateTemplateFile	string property	By default, it is set to “createTroubleTicketByValueRequest.xml".
AssociateTemplateFile	string property	By default, it is set to “associateTroubleTicketByValueRequest.xml".
DissociateTemplateFile	string property	By default, it is set to “dissociateTroubleTicketByValueRequest.xml".
Input	string property	By default, it is set to “input".

Table 10 – Specific optional IM troubleTicketAction configuration for HP TeMIP

5.3 Problem Detection

5.3.1 Filters, tags and mappers

A PD scenario contains three standard HP UCA EBC configuration files:
- "*ProblemDetection_filters.xml*" defines the problems and their tags.
- "*ProblemDetection_filtersTags.xml*" defines the tags associated to the filters.
- "*ProblemDetection_mappers.xml*" defines the different mappers and the neo4j Cypher queries to use in PD VP, mainly specified by tags.

The <*topFilter*> elements defined in the "*ProblemDetection_filters.xml*" file are closely related to the PD VP code itself, because it defines the Java classes corresponding to a specific problem. It must not be modified except in rare conditions, for example:

- A problem priority needs to be re-assessed.
- A new mapper is used for computing the unique source ID of an incoming event.
- The role of a specific filtered alarm is updated. The "*ProblemDetection_filtersTags.xml*" is only used by the GUI to associate tags and filters in the filter builder panel.

The PD framework recognizes the following predefined tags:

- For event objects:

Tag	Event role	Description
tag="TriggerEvent"	Trigger Event	An event that marks an important problem symptom and triggers the creation of a group.
tag="SubEvent"	Sub-Event	An event that marks a problem and is grouped under a Problem Alarm.

Table 11 – Tags for possible roles of an event within PD

- For alarm objects:

Tag	Alarm role	Description
tag="Trigger"	Trigger Alarm	An alarm that marks an important problem symptom and triggers the creation of a Problem Alarm.
tag="SubAlarm"	Sub-Alarm	An alarm that marks a problem and is grouped under a problem Alarm.
tag="ProblemAlarm"	Problem Alarm	An alarm that summarizes the details of the problem, and is readable by the operator.

Table 12 – Tags for possible roles of an alarm within PD

You can also combine these tags, for example:

- tag="SubAlarm,ProblemAlarm" → Defines an alarm which is Problem Alarm of a problem, and the Sub-alarm of another problem.
- tag="Trigger,ProblemAlarm" → The Trigger Alarm is considered as a Problem Alarm (no new alarm is created).

The <***cypherQuery***> elements defined in the "*ProblemDetection_mappers.xml*" file are closely related to the topology loaded in Neo4j. This file must **not be modified** except in some rare conditions.

However, the <***mapper***> elements **can be changed** to handle new conditions on incoming events, but in such a case, the "*ProblemDetection_filtersTags.xml*" must be updated accordingly.

5.3.2 Specific configuration

A PD scenario is delivered with a specific "*ProblemXmlConfig.xml*" file.

5.3.2.1 Main Policy

The <mainPolicy> element is a configuration setting which is common to all problems defined in a PD scenario, and not linked to any problem.

It has the following attributes:

Name	**Type**	**Value**
enablePrioritySort	Boolean attribute	Enables the group sorting feature. Default value is "false"
multipleParentSupport	Boolean attribute	Specifies the ability to set the parent relationship for each group of the Problem Alarm (true) or only with the one of highest priority (false). Default value is "true"
enableTopoAccess	Boolean attribute	Specifies whether to access topology information when computing information for Problem Alarm during the workflow. If set to true, computing information is calculated, computeSourceUniqueID() and computeDBRecords()) are called. Default value is "false"

Table 13 – PD mainPolicy attributes

It also contains the following elements:

<candidateVisibility>

Before a problem is detected, an alarm belonging to a set of potential alarms characterizing a problem can be considered as a "*candidate alarm*" for this problem. When the problem is detected (for example when the Problem Alarm is received), the "candidate alarm" becomes a Sub-Alarm of the problem. A Trigger Alarm can also be considered a "candidate alarm" for the problem, until the problem is detected.

The `candidateVisibilityTimeValue` parameter indicates how long an alarm should be shown as a "candidate alarm" in the Network Management System viewer. This parameter is read-only if `candidateVisibilityTimeMode` is set to "`Value`". The value is expressed in milliseconds.

The `candidateVisibilityTimeMode` parameter is subtle.

It can take three values: "`Max`" (default value), "`Min`", or "`Value`"

"`Max`" means that the alarm will remain a candidate alarm as long as there is a chance that this alarm may be associated with a problem instance.

In the diagram below, the alarm (upper left arrow) can belong to three problem types. So it will remain as a candidate alarm for as long as there is a possibility that this alarm becomes part of one of the problems (problem A, B, or C).

To be part of a problem instance, an alarm must be included in a time window (see Figure 12) around the time of appearance of a Trigger Alarm for that problem. In the following diagram if none of the Trigger Alarms for problem A, B, and C appear, the alarm remains a candidate until the max value of `timeWindowBeforeTrigger` of problems A, B, and C. After the time window has expired, incoming Trigger Alarms are not taken into account.

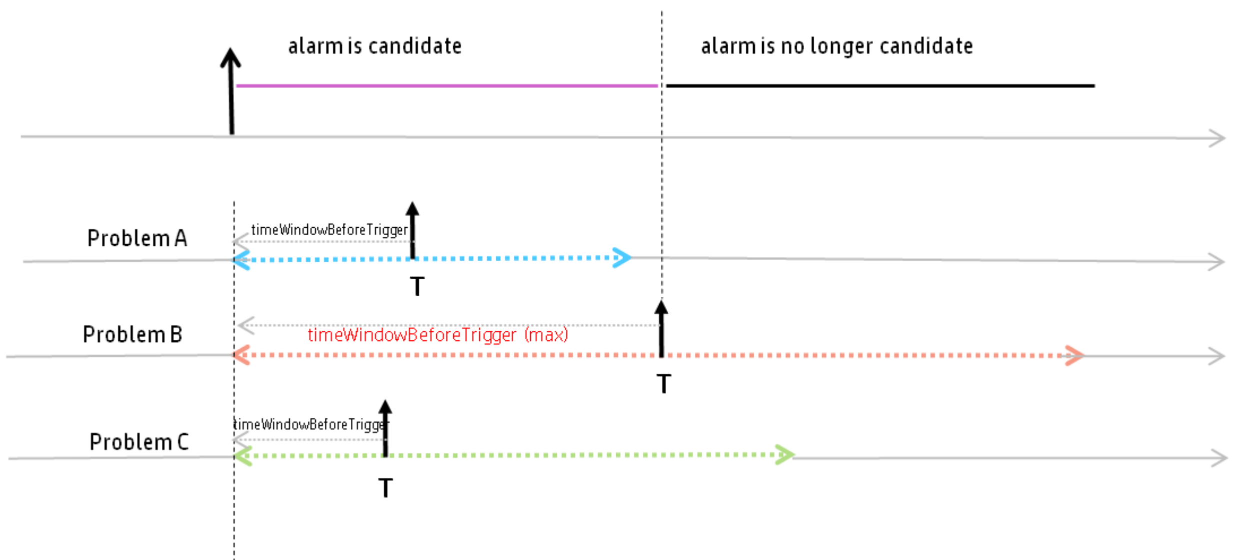


Figure 12 - Explanation of the candidateVisibilityTimeMode=Max

`candidateVisibilityTimeMode=Value` specifies how long the alarm remains a candidate alarm (expressed in milliseconds).

`candidateVisibilityTimeMode=Min` specifies that as soon as there is at least one potential problem instance an alarm cannot be part of, this alarm will not be marked as a candidate alarm any longer.

The `markCandidate` parameter indicates whether an alarm is marked as a "candidate alarm" in the Network Management System viewer (provided the NMS viewer has this capacity).

<transientFiltering>

The concept of transient filtering derives from the observation that alarms can disappear automatically after a certain time period; In this case it is useful for a PD Value Pack to query and verify which alarms are still active.

If `transientFilteringEnabled=true`, the Transient Filtering feature of the Problem Detection Value Pack delays processing received alarms for the duration specified in (`transientFilteringDelay`). It is possible the alarm disappears within this time range.

```
transientFilteringEnabled=true|false
transientFilteringDelay=<waiting period in milliseconds>
```

<actions>

The PD framework is able to configure multiple action factories in order to support multiple NMS. See 5.2.1 Actions to NMS for details.

< troubleTicketActions>

The PD framework is able to configure trouble ticket action factories. See 5.2.2 Actions for details.

This element is optional.

< counterTotalNumberAlarms>

This element specifies what to count for the Problem Alarm field measuring the total number of alarms: either the current number of alarms in the group, or the total number of alarms since the group creation.

5.3.2.2 Problem Specific Policies

Problem Specific Policies are configuration settings specific to each problem defined in a PD Value Pack.

These problem specific configuration settings are defined inside the `<problemPolicy name="…">` XML tag.

The <*problemPolicy*> element has the following attributes:

Name	Type	Value
enableComputeProblemEntityFromMappers	Boolean attribute	If set to false, the usage of calling mappers in computeProblemEntity() is disabled. Default value is “true”
enableComputeProblemEntityFromFields	Boolean attribute	If set to true, the calculation of fields key/value pairs in computeProblemEntity() is enabled. Default value is “false”

Table 14 – problemPolicy attributes

It also contains the following elements:

<problemAlarm>

The <*problemAlarm*> element specifies the behavior related to Problem Alarms.

Name	Type	Value
delayForProblemAlarmCreation	long (optional)	Delay, expressed in milliseconds, before the associated Problem Alarm is created. Example: A setting of 2000 applies a delay of 2000 ms (2 seconds) before creating Problem Alarms. Default value is 2000.
delayForProblemAlarmClearance	Long (optional)	Delay, expressed in milliseconds, before the Problem Alarm is cleared. Example: A setting of 0 means there is no delay in the clearance of Problem Alarms after all conditions are met for clearing Problem Alarms. Default value is 10000.

problemAlarmCanTriggerAnotherGroupForSameProblem	Boolean (optional)	Supports the concept of nested problems, for example. One alarm can have multiple roles for the same problem. It can be a Problem Alarm for one group, a Trigger for another group, or attached to another group of the same problem. If set to false, the Problem Alarm cannot create a new group for the same problem. If set to true, the Problem Alarm of a group can also create new groups for the same problem. Default setting is false.
terminateWhenLonesome	Boolean attribute	If set to true, the PD framework will automatically terminate Problem Alarms that have become lonesome (with no sub-alarms attached) after a resynchronization. Default value is "false"

Table 15 – PD problemAlarm per-problem configuration

<troubleTicket>

It is possible for PD Value Packs to automatically create a trouble ticket associated to a Problem Alarm.

The following configuration parameters control the creation of trouble tickets for Problem Alarms:

Name	Type	Value
automaticTroubleTicketCreation	Boolean	If set to false, a trouble ticket is not created automatically when a Problem Alarm is created. If set to true, a trouble ticket is automatically created when a Problem Alarm is created.
propagateTroubleTicketToSubAlarms	Boolean	If set to true all Sub-Alarms (of the Problem Alarm), are associated to the trouble ticket linked with the Problem Alarm. If set to false, Sub-Alarms are not associated to the trouble ticket linked with the Problem Alarm.
propagateTroubleTicketToProblemAlarm	Boolean	When false, if a Sub-Alarm has a trouble ticket, the Problem Alarm is not linked to this trouble ticket. When true, if a Sub-Alarm has a trouble ticket, the Problem Alarm is linked to this trouble ticket.
delayForTroubleTicketCreation	Long (optional)	Delay, expressed in milliseconds, () before the associated trouble ticket is created after the creation of a Problem Alarm. Default is 10000.

Table 16 – PD troubleTicket "per-problem" configuration

<groupTickFlagAware>

When set to true, the PD Value Pack executes user code at regular tick intervals, if customized for such behavior.

HP recommends not to change this setting unless required by the VP developer.

<sameGroupForAllProblemEntities>

This property is optional and applicable only if a Trigger Alarm has multiple problem entities.

If set to false, multiple groups are created for the same Trigger Alarm.

If set to true, only one group is created for the Trigger Alarm, and this group covers all problem entities of the Trigger Alarm.

<problemAlarmAbleToCreateGroup>

This property is optional.

By default in Problem Detection, a Problem Alarm is allowed to create a group, if the trigger that created this Problem Alarm is not present.

This generally does not cause any problem, because the lifecycle of the group is properly managed.

In some cases, the lifecycle of Problem Alarms is not handled directly (only life cycle of non-'Problem Alarms' is handled). As a consequence, the life cycle of the group is also not handled.

For such cases, this property helps to prevent Problem Alarms from creating groups.

If set to 'true', it does not change the recommended default behavior of Problem Detection. If set to 'false', Problem Alarms corresponding to triggers that are not present anymore in the working memory, or present as mere Sub-Alarms, are discarded.

<enableTriggerConsistencyAfterResync>

This property is optional.

By default in Problem Detection, a created group can change its Trigger Alarm after a resynchronization. This is useful because alarms that are getting resynchronized are received in the reverse order compared to the original order: If the Problem Alarm of a group is received before the original trigger that was used to create the group.

If the <enableTriggerConsistencyAfterResync> property is set to true, the following mechanism is used to keep consistency among groups. If Problem Detection detects a case in which an original Trigger Alarm is received after the group is created, because of the prior reception of the Problem Alarm of the same group, then the original trigger takes back its original role of Trigger Alarm, instead of the Problem Alarm that was assumed as the Trigger Alarm.

To disable this feature, this property must be set to 'false'.

It can be useful to disable this feature if, for example, your customization of PD framework already recomputed the Trigger Alarm.

<computeProblemEntityFromFields>

This element is optional.

The "*keyValueSeparator*" attribute defines the separator string, which is by default "=".

It holds a sequence of <*field*> elements that are defined as below:

Name	Property	Value

Key	Property	The field key of an alarm used as a key/value pair for computeProblemEntity(). The field can be a custom field.
valueIgnored	Property (optional)	An optional value to be ignored for a field during computeProblemEntity().

Note that the key property is a tuple:

Name	Type	Value
tagName	string	The tag defining the field name to be used as key/value pair for computeProblemEntity().
fieldName	string	The field name to be used as key/value pair for computeProblemEntity().

Table 17 – PD computeProblemEntityFromFields "per-problem" configuration

<timeWindow>

This element holds the following properties:

Name	Type	Value
timeWindowMode	string	A TimeWindow is used to decide if an Alarm is part of a Group of Alarms depending on its alarmRaisedTime field. Default value is 'None', that is, no time window. This is the equivalent of an infinite time window. All alarms regardless of their timestamp can be associated with a problem. If set to 'Trigger', a time window around the (first) Trigger Alarm of a problem is used. Only alarms with timestamps inside this time window can be associated with a problem.
timeWindowBeforeTrigger	long (optional)	A time window expressed in milliseconds before the Trigger's alarmRaisedTime to consider an alarm as part of the Trigger's problem. Default value is 30000.
timeWindowAfterTrigger	long (optional)	A time window, expressed in milliseconds, after the Trigger's alarmRaisedTime to consider an alarm as part of the Trigger's problem. Default value is 30000.

Table 18 – PD timeWindow "per-problem" configuration

Also, depending on customer Value Pack:

Name	Property	Value
booleans	Property (optional)	For defining multiple booleans for a specific use-case.

strings	Property (optional)	For defining multiple strings for a specific use-case.
Longs	Property (optional)	For defining multiple longs for a specific use-case.

Table 19 – PD customized "per-problem" configuration

5.4 Topology State Propagator

5.4.1 Filters, tags and mappers

A TSP scenario contains three standard HP UCA EBC configuration files:
- "*TopologyPropagation_filters.xml*" defines the propagation and their tags.
- "*TopologyPropagation_filtersTags.xml*" defines the tags associated to the filters.
- "*TopologyPropagation_mappers.xml*" defines the different mappers and the neo4j Cypher queries to use in PD VP, mainly specified by tags.

The <*topFilter*> elements defined in the "*TopologyPropagation_filters.xml*" file are closely related to the TSP VP code itself, because it defines the Java classes corresponding to a specific propagation. It must not be modified except in rare conditions, for example:

- A propagation priority needs to be re-assessed.
- A new mapper is used for computing the unique source id of an incoming event.
- The role of a specific filtered alarm is updated

The "*TopologyPropagation_filtersTags.xml*" is only used by the GUI to associate tags and filters in the filter builder panel.

The TSP framework recognizes the following three predefined tags:

Tag	Alarm role	Description
tag="RootCauseAlarm"	Root Cause Alarm	A Root Cause Alarm that represents a problem, and is attached to a specific propagation. In the IM Value Pack, such a Root Cause Alarm is a Problem Alarm coming from PD.
tag="SubAlarm"	Sub-Service Alarm	An alarm that represents a Propagation but is associated under a higher Service Alarm.
tag="ServiceAlarm"	Service Alarm	An alarm that summarizes the propagation, and is readable by the operator.

Table 20 – Tags for possible roles of an alarm within TSP

Unlike PD, you cannot combine these tags.

The <*cypherQuery*> elements defined in the "*TopologyPropagation_mappers.xml*" file are closely related to the topology loaded in Neo4j. This file must not be modified except in some rare conditions.

However, the <*mapper*> elements can be changed to handle new conditions on incoming events, but in such a case, the "*TopologyPropagation_filtersTags.xml*" file must be updated accordingly.

5.4.1.1 The special topFilter named ReservedForGeneralBehavior

When an event is received by the TSP framework, its unique source identifier must be computed. This is done by calling the *computeSourceUniqueId*() method of the GeneralBehavior class defined for all propagations.

The TSP framework provides a default class which by default does the following:

- Looks into the passing ReservedForGeneralBehavior filter for the tag named "*ComputeSourceUniqueIdMapper*" that will give a name of a mapper to execute
- Executes that mapper which should be present in "*TopologyPropagation_mappers.xml*" and returns the computed string

5.4.1.2 The special tag named CypherQuery

After a unique source identifier is computed for an event in the TSP framework, it is necessary to retrieve the topology records associated to the object represented by the event, that is, all the nodes impacted by the object.

This is done by calling the *computeDbRecords*() method of the concerned propagation class.

The default propagation class of the IM framework does the following:

- Looks into the passing filter for the propagation to check the "*CypherQuery*" tag that gives the name of the Neo4j query to execute.
- Executes the query that must be present in "*TopologyPropagation_mappers.xml*" and returns the executed query that contains the resulted records.

Note that the GUI filter builder requires that all the <*cypherQuery*> elements that are defined in "*TopologyPropagation_mappers.xml*" must also be referenced in "*TopologyPropagation_filterTags.xml*" under a <*paramTag*> named "CypherQuery" and proposing an enum of all those queries.

5.4.2 Specific configuration

A TSP scenario is delivered with a specific "*PropagationXmlConfig.xml*" file.

5.4.2.1 Main Policy

The <*mainPolicy*> element is a configuration setting which is common to all propagations defined in a TSP scenario, and not linked to any propagation.

It has one attribute:

Name	Type	Value
stateSourceIdentifier	String attribute	It is used to fill the "sourceIdentifier" field of a state event generated by the TSP framework.

Table 21 – TSP mainPolicy attributes

It has the following elements:

<actions>

The TSP framework is able to configure multiple actions factories in order to support multiple NMSs. For more details, see section 5.2.1.

This element is optional.

<troubleTicketActions>

The TSP framework is able to configure trouble ticket actions factories. For more details, see 5.2.2.

This element is optional.

<counterTotalNumberAlarms>

It specifies what to count for the Service Alarm field representing the Total Number of Alarms: either the current number of alarms in the group or the total number of alarms since the group creation.

This element is optional.

5.4.2.2 Propagation Specific Policies

Propagation Specific Policies are configuration settings specific to each of the propagations defined in a TSP Value Pack.

These propagation specific configuration settings are defined inside the `<propagationPolicy name="…">` XML tag.

It has the following elements:

<serviceAlarm>

The <serviceAlarm> element specifies behavior around ServiceAlarm.

Name	Type	Value
enableServiceAlarmCreation	Boolean (optional)	If set to true, the Service Alarm is automatically created for this propagation. If set to false (by default), no Service Alarm is created for the propagation.
delayForServiceAlarmCreation	long (optional)	Delay, expressed in milliseconds, before the associated Service Alarm is created. Example: A setting of 10000 applies a delay of 10 seconds before creating Service Alarms. Default value is 2000.
delayForServiceAlarmClearance	long (optional)	Delay, expressed in milliseconds, before the Service Alarm is cleared. Example: A setting of 0 means there is no delay in the clearance of Service Alarms after all conditions are met for clearing Service Alarms. Default value is 10000.
attachWholeSubTreeRootCauses	Boolean (optional)	If set to true, the whole sub-tree of Root Cause Alarms are attached to the Service Alarm, that is, the direct Root Cause Alarms plus the Root Cause Alarms part of impacting states. If set to false (by default), only the direct Root Cause Alarms are attached to the Service Alarm.

Table 22 – TSP serviceAlarm per-propagation configuration

<troubleTicket>

It is possible for TSP Value Packs to automatically create a trouble ticket associated to a Service Alarm.

The following configuration parameters control the creation of trouble tickets for Service Alarms:

Name	Type	Value
automaticTroubleTicketCreation	Boolean	If set to false, a trouble ticket is not created automatically when a Service Alarm is created. If set to true, a trouble ticket is automatically created when a Service Alarm is created.
propagateTroubleTicketToSubAlarms	Boolean	If set to true, all Sub-Alarms (of the Service Alarm), are associated to the trouble ticket linked with the Service Alarm. If set to false, Sub -Alarms are not associated to the trouble ticket linked with the Service Alarm.
propagateTroubleTicketToMasterAlarm	Boolean	When false, if a Sub-Alarm has a trouble ticket, the Service Alarm is not linked to this trouble ticket. When true, if a Sub-Alarm has a trouble ticket, the Service Alarm is linked to this trouble ticket.
delayForTroubleTicketCreation	long (optional)	Delay, expressed in milliseconds before the associated trouble ticket is created after the creation of a Service Alarm. Default value is 10000.

Table 23 – TSP troubleTicket "per-propagation" configuration

Note that the <*troubleTicket*> container element is optional.

<groupTickFlagAware>

This element is optional. When set to true, the TSP scenario executes user code that regular tick intervals, if customized for such behavior.

HP recommends not to change this setting unless required by VP the developer.

<propagationRule>

This element is optional. When used, it defines a rule element that will be used during state calculation.

The possible values of that single rule element are:

Value	Description
WorstChildPercentage	The worst impacting node is used.
FullPercentage	An average of all impacting nodes is used.
Custom	The calculation is Java code based.

<nodes>

This optional element is a sequence of <dbType> elements used to configure the topology nodes. A <dbType> element is defined by the following property:

Name	Type	Value
key	Property	The type of the node to include

<poiCategories>

This optional element is a sequence of <poiCategory> elements used to configure the Point Of Interest Categories. A <poiCategory> element is defined by the following property:

Name	Type	Value
key	Property	The POI category to assign

<thresholdValues>

This optional element is a sequence of six elements used to configure the Threshold values. The elements are in strict order as follows:

Name	Type	Value
OK	Property	Threshold for state OK
LOW	Property	Threshold for state LOW
MEDIUM	Property	Threshold for state MEDIUM
HIGH	Property	Threshold for state HIGH
CRITICAL	Property	Threshold for state CRITICAL
DOWN	Property	Threshold for state DOWN

Each threshold value property must be defined using the following three elements:

Name	Type	Value
perceivedSeverity	Property	The perceived severity for the threshold value. The value can be: - INDETERMINATE - WARNING - MINOR - MAJOR - CRITICAL - CLEAR
availabilityPercentage	Property	The percentage of availability of the node for the threshold value. This property is a double.

poiImportance	Property	The importance for the POI for the threshold value. The value can be: - None - Low - Medium - High - Critical

You can have an example in section 8.3.2.**Error! Reference source not found.**

<propagationObject >

This optional element is a string defining the propagation state name when creating Node POIs and the name is used for creating Service Alarms.

<statusName>

This optional element is a string defining the attribute name for the status attribute when creating Node POIs.

<percentageAvailabilityKey>

This optional element is a string defining the attribute name for the percentageAvailability attribute when creating Node POIs.

Also, depending on customer Value Pack needs:

Name	Property	Value
<booleans>	Property (optional)	For defining multiple booleans for a specific use-case.
<strings>	Property (optional)	For defining multiple strings for a specific use-case.
<longs>	Property (optional)	For defining multiple longs for a specific use-case.

Table 24 – TSP customized "per-propagation" configuration

5.5 Orchestra

An IM Value Pack is delivered with its Orchestra configuration that must be added in the global *"OrchestraConfiguration.xml"* file.

A typical IM Orchestra configuration is to forward alarms from PD to TSP.

An example is given in Figure 13 - IM Orchestra configuration example.

```xml
<OrchestraWorkflow xmlns="http://hp.com/uca/expert/orchestra/config" >
 <Routes>
  <Route name="PD-> TSP" >
   <COPY>
    <Source>
     <ValuePackNameVersion><![CDATA[my-im-0.1]]></ValuePackNameVersion>
     <ScenarioName><![CDATA[com.hp.uca.expert.vp.im.pd.ProblemDetection]]></ScenarioName>
    </Source>
    <Destinations>
     <Destination>
      <Target>
       <ValuePackNameVersion><![CDATA[my-im-0.1]]></ValuePackNameVersion>
       <ScenarioName><![CDATA[com.hp.uca.expert.vp.im.tsp.TopologyStatePropagator]]></ScenarioName>
      </Target>
     </Destination>
    </Destinations>
   </COPY>
  </Route>
 </Routes>
</OrchestraWorkflow>
```

Figure 13 - IM Orchestra configuration example

For details on how to use the HP UCA EBC V3.1 Orchestration feature, see [R1] *Unified Correlation Analyzer for Event Based Correlation Reference Guide* and [R2] *Unified Correlation Analyzer for Event Based Correlation Value Pack Development Guide*

Chapter 6

Developing an IM Value Pack

The UCA for EBC Inference Machine SDK provides several Eclipse plugins to ease the Value Pack development of IM Value Packs, PD Value Packs, and TSP Value Packs.

6.1 Eclipse Plugins

The pre-requisite of using the Eclipse plugins is the installation of the HP UCA for EBC Inference Machine Development Kit which is comprised of:

- HP UCA for EBC Development Kit (see [R2] Unified Correlation Analyzer for Event *Based Correlation Value Pack Development Guide*)
- HP UCA for EBC Development Kit Inference Machine Extension

Four pre-defined Value Packs can be created:

- Problem Detection only VP
- Problem Detection with topology-enabled VP (requires topology)
- Topology State Propagator only VP (requires topology)
- Inference Machine complete VP (requires topology)

6.1.1 Creating a UCA EBC project in Eclipse

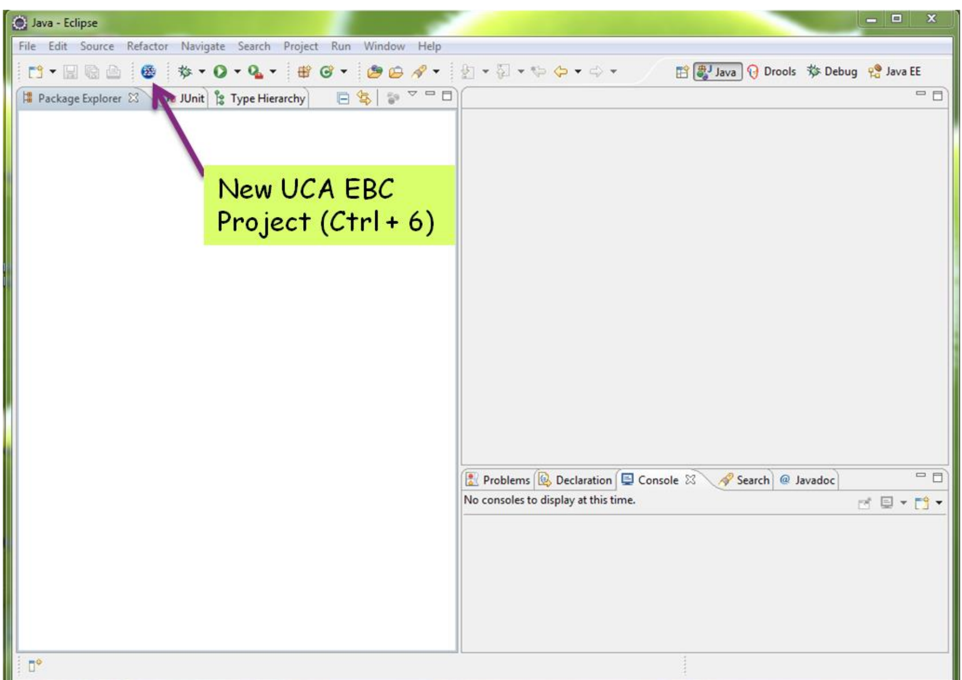


Figure 14 - How to create a UCA EBC project in Eclipse

6.1.2 Creating a Problem Detection only Value Pack

1. Create an UCA EBC project in Eclipse.
2. Select only “Problem Detection Scenario”.

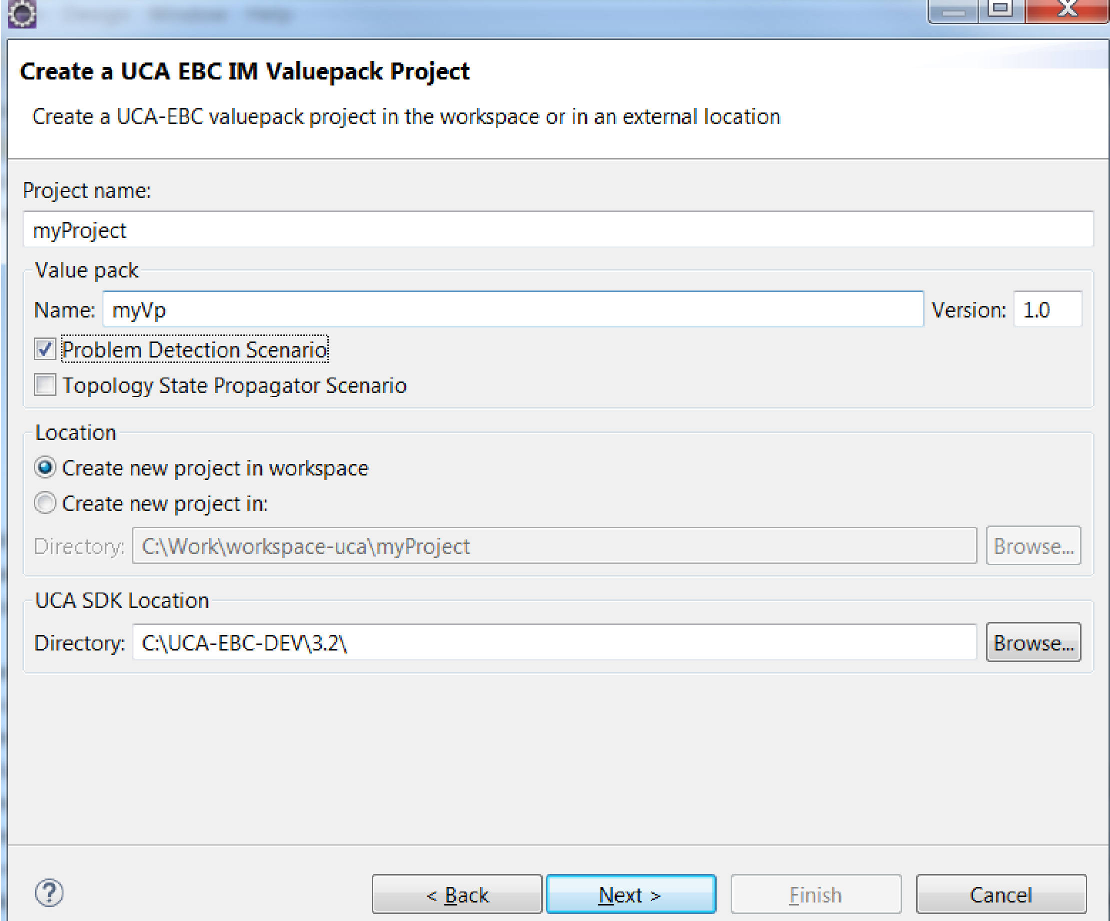


Figure 15 – Create PD only Value Pack

3. Configure the value pack.

   The following steps are **mandatory**:

   a. Rename and edit “Problem_Skeleton.java”.

   b. Edit the filters file.

   c. Configure the Main Policies and the Problem Specific Policies.

   In `src/test/resources com.hp.uca.expert.vp.pd.core ProblemDefault.java` is available as a reference (not for modification) for the default code of the overridable methods.

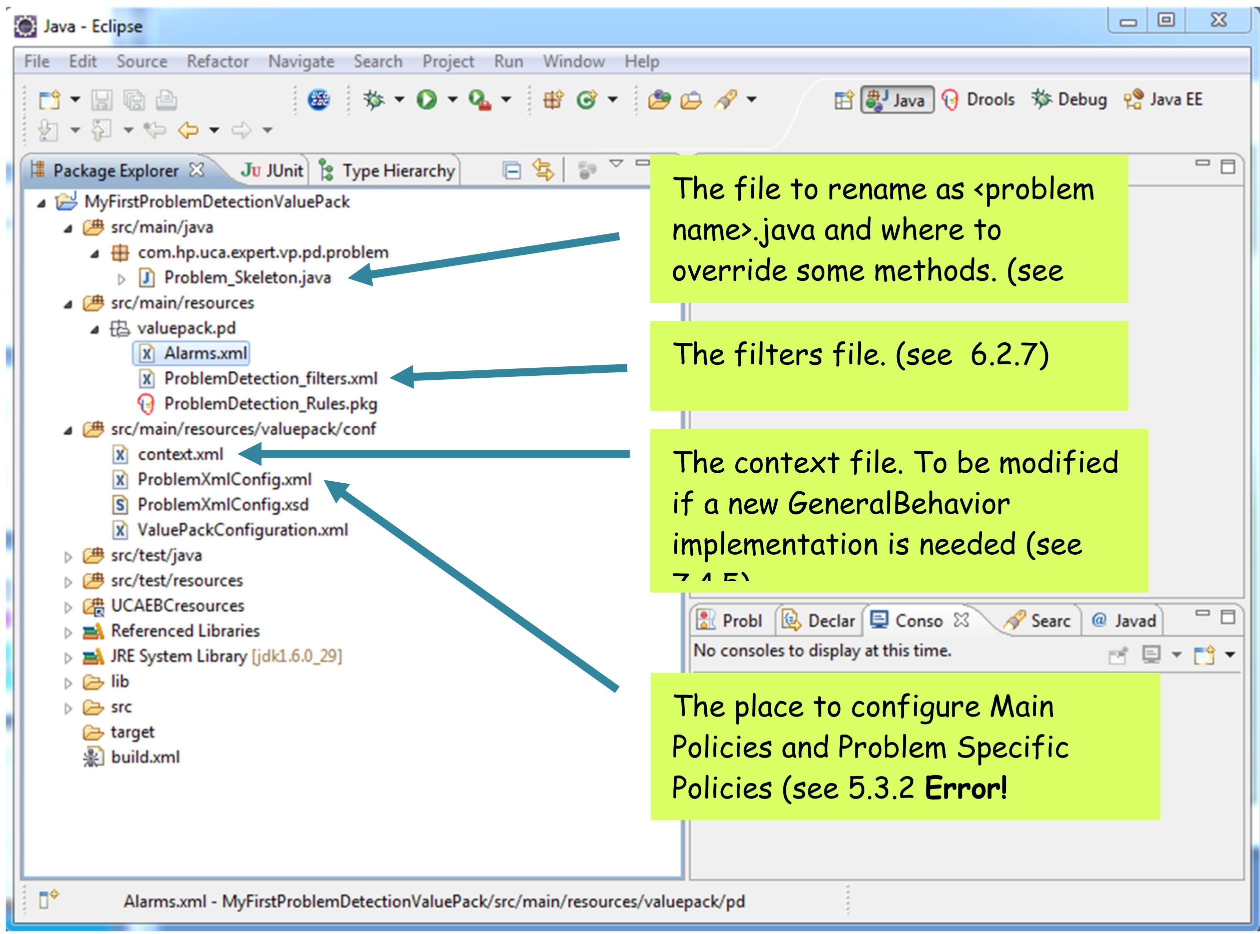


Figure 16 - Files to edit to configure MyFirstProblemDetectionValuePack

6.1.3 Creating a Topology State Propagator only Value Pack

1. Create an UCA EBC project in Eclipse.
2. Select only "Topology State Propagator Scenario".

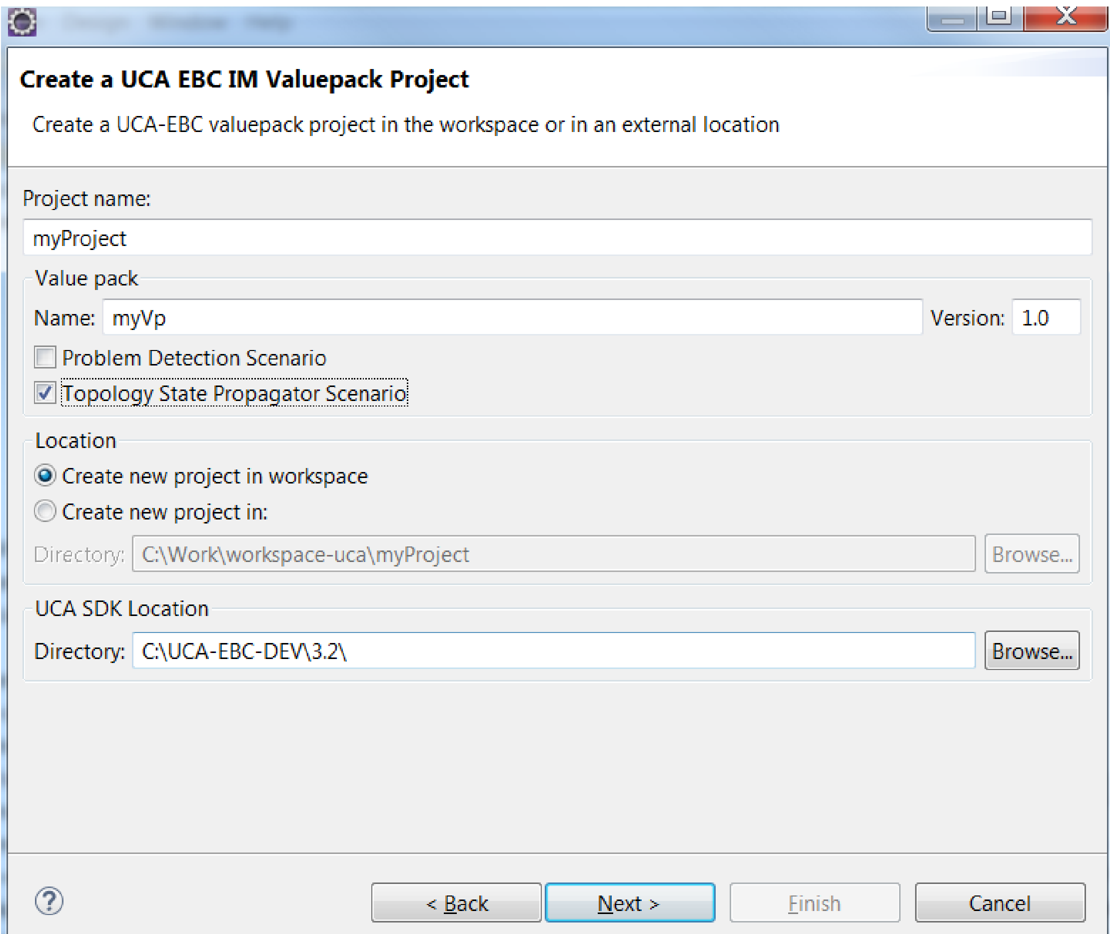


Figure 17 – Create TSP only Value Pack

6.1.4 Creating an Inference Machine Value Pack

1. Create an UCA EBC project in Eclipse.
2. Select both “Problem Detection Scenario” and “Topology State Propagator Scenario”.

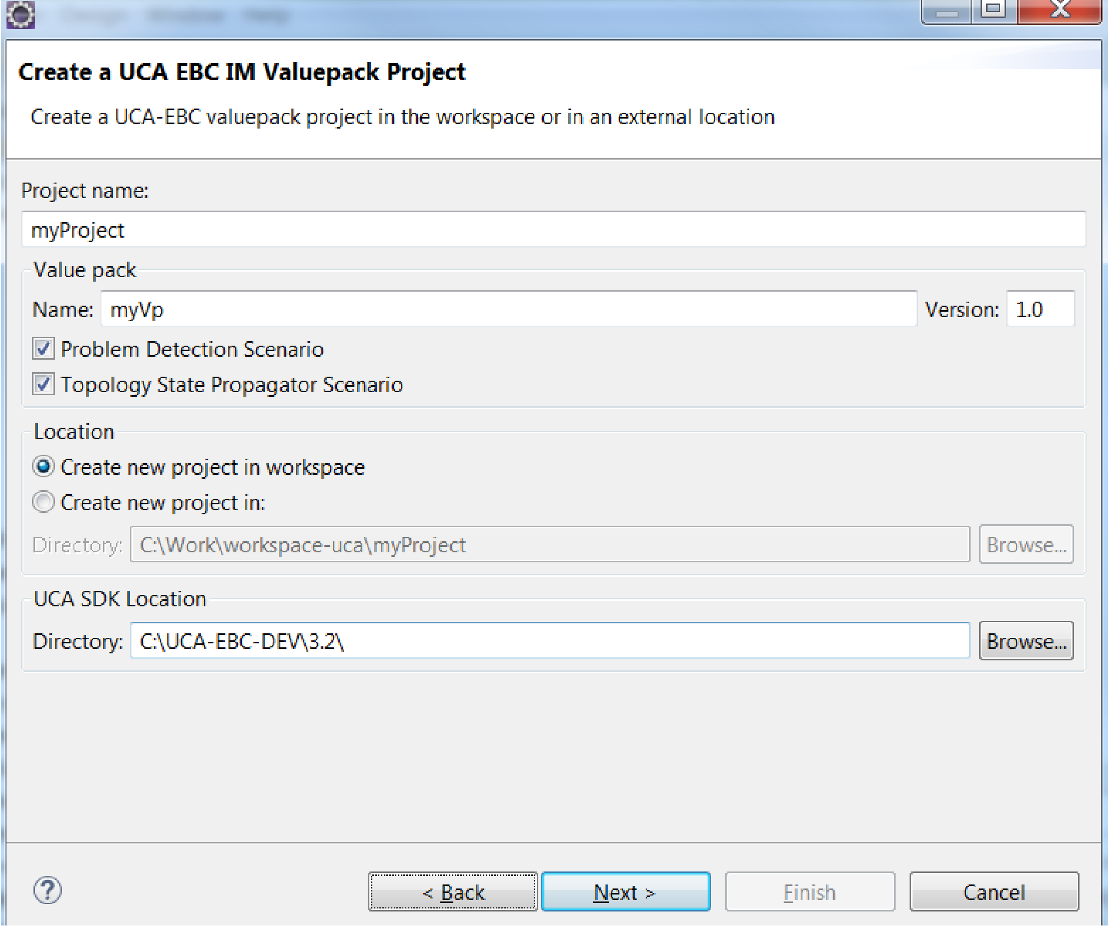


Figure 18 – Create IM Value Pack

6.2 Creating a simple Problem Detection Value Pack

The following sections describe how to create a simple Problem Detection Value Pack (PD VP). For advanced PD VP features, see Chapter 7.

6.2.1 Analyzing the problems to be detected

Before creating a Problem Detection Value Pack, it is essential to identify all the problems that could arise from an operations perspective, and the corresponding alarms that can be generated in the context of each problem.

To use a medical analogy:

- Alarms are the symptoms.
- Problem is the disease.

- Problem Detection Value Pack is the physician. Based on the symptoms observed (the alarms received), PD VP diagnoses the disease (identifies the problem).

The main steps of this initial PD analysis can be summarized as follows:

1. List all potential alarms that the Network Management System (NMS) might receive.
2. Do the RCA analysis: list the problems that might occur in the network and that the user of an NMS is likely be interested in.
3. For each problem, identify which alarms are associated with the problem (note that an alarm can be associated with several problems).

6.2.2 Identifying the different types of alarms

Among all the alarms associated with a problem, “trigger” alarms must be separated from “sub-alarms”. Trigger Alarms define the problem and trigger the creation of a Problem Alarm.

To continue with the medical analogy:

- Trigger Alarms are the primary symptoms.
- Sub-Alarms are the secondary symptoms.

At runtime, by default, a Problem Detection Value Pack considers that an instance of a problem has occurred if the following criteria are met:

- One trigger alarm of the problem is received.
- At least one Sub-Alarm of the problem is received.

This default behavior can be customized (see sections 7.4 and 8.1).

The main steps of alarm categorization can be summarized as follows:

1. Identify all the potential problems and the associated alarms (see section 6.2.1).
2. Separate trigger alarms from sub-alarms.
3. Configure the filters of for the Problem Detection Value Pack.

Filters give logical criteria to distinguish different alarms. They allow distinguishing which alarm belongs to which problem, and with which potential role (Problem Alarm, Trigger Alarm, or Sub-Alarm.

Filters are configured in an XML file.

For more details, see section 6.2.7 Defining the Filters and Annex B.

6.2.3 Configuring the Time Window

Consider T_{pb} to be the time at which the problem occurred. Note that for Problem Detection it is the time of the first Trigger Alarm.

We must configure a time window around T_{pb} where:

- All alarms outside this time window will not be associated with the problem.
- All alarms inside this time window are potential candidates to be associated with the problem.

Note that time windows can be infinite.

The following diagram illustrates the time window, defined by the `timeWindowBeforeTrigger` and `timeWindowAfterTrigger` properties in a configuration file. For more details, see section 5.3.2.2.

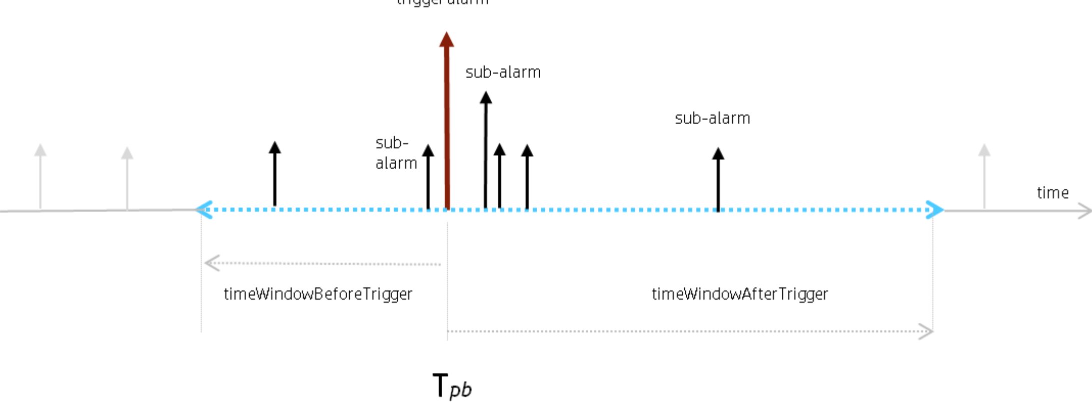


Figure 19 - Time window illustration

Alarms in grey are ignored because they are outside of the time window of the problem.

Alarms in black are not ignored because they are inside the time window of the problem. They will be evaluated by the Problem Detection Value Pack. Some of them will meet the conditions to become Sub-Alarm of the problem, while some others will not.

6.2.4 Configuring Problem Alarm creation

For each problem, you must decide whether, at runtime, upon occurrence of the problem, the Problem Detection Value Pack creates a Problem Alarm or re-use (promote) the Trigger Alarm (or one of the Trigger Alarms) as a Problem Alarm.

The main steps of configuring Problem Alarm creation can be summarized as follows:

1. Define the filters in the XML configuration file. For more details, see section 6.2.7.
2. If you want to create a new Problem Alarm for a problem, configure an action to effectively create this Problem Alarm in the Network Monitoring System (NMS).
3. Configure when the Problem Alarm is created. A Problem Alarm can be created as soon as the problem is detected or after a given amount of time. For more details, see Chapter 7.

6.2.5 Configuring Trouble Ticketing

For each problem, you must decide whether, at runtime, upon occurrence of the problem, the Problem Detection Value Pack raises a trouble ticket. For more details, see Chapter 7.

6.2.6 Considering if the default behavior needs to be modified

Problem Detection offers a default behavior that allows you to create a Value Pack without further configuration steps than the ones described in sections 6.2.1 to 6.2.5. Nevertheless, almost any aspect of the default behavior can be customized if necessary.

For example, by default, the Problem Detection framework sets the severity of the Problem Alarm based on the properties of the Sub-Alarm with the highest severity (among all Sub-Alarms of the problem), but the framework allows you to modify this rule.

Another aspect of the default behavior that frequently need to be modified is the way the problem entity is calculated.

The problem entity represents information related to the network resource that is common to all alarms of the problem. By default the problem entity is set to the originatingManagedEntity of the Trigger Alarm, but it can be location information (for example, “Paris_south _MKF2”) contained in the AdditionalText field.

For more details on the default behavior and how to customize it, see Chapter 7.

6.2.7 Defining the Filters

Defining the filters is the most important step when creating a Problem Detection Value Pack. Defining filters is not only about specifying which alarms are relevant to the Value Pack. It is also about specifying which alarm is associated to which problem, and what the role of each alarm is: Problem Alarm, Trigger Alarm, or Sub-Alarm.

Because a Problem Detection Value Pack is an HP UCA for EBC Value Pack, defining filters for Problem Detection Value Packs is done the same way as for any other HP UCA EBC Value Pack.

The definition of filters is done in the “`ProblemDetection_filters.xml`” file located in `src/main/resources/valuepack/pd/`

The filter file of a Problem Detection Value Pack can include several “top filter” sections, one for each problem to detect. The following example shows the “top filter” section of a “`ProblemDetection_filters.xml`” file for a problem named “*Problem_BitError*”.

To see an example of a filter file that contains several “top filter” sections in order to detect several problems, see the filter file of the Value Pack example in Annex B.

```
<topFilter name="Problem_BitError">
  <anyCondition>

    <allCondition tag="TeMIP TT">
            <allCondition>
                <stringFilterStatement>
                        <fieldName>originatingManagedEntity</fieldName>
                        <operator>matches</operator>
                        <fieldValue>motorola_omcr_system .* managedelement .*
                                        bssfunction .* btssitemgr .*</fieldValue>
                </stringFilterStatement>
                <stringFilterStatement tag="Trigger ">
                        <fieldName>additionalText</fieldName>
                        <operator>contains</operator>
                        <fieldValue>[14] Bit error OOS threshold exceeded</fieldValue>
                </stringFilterStatement>
                <stringFilterStatement tag="Trigger ">
                        <fieldName>additionalText</fieldName>
                        <operator>contains</operator>
                        <fieldValue>[6] Remote Alarm OOS Threshold Exceeded</fieldValue>
                </stringFilterStatement>
                <stringFilterStatement tag="SubAlarm">
                        <fieldName>additionalText</fieldName>
                        <operator>contains</operator>
                        <fieldValue>[10] Link Disconnected</fieldValue>
                </stringFilterStatement>
                <stringFilterStatement tag="SubAlarm">
                        <fieldName>additionalText</fieldName>
                        <operator>contains</operator>
                        <fieldValue>[0] Last RSL Link Failure</fieldValue>
                </stringFilterStatement>
            </anyCondition>
    </allCondition>

    <allCondition tag="TeMIP TT">
        <stringFilterStatement>
                <fieldName>userText</fieldName>
                <operator>matches</operator>
                <fieldValue>.*&lt;action&gt;UCA EBC .*</fieldValue>
        </stringFilterStatement>
        <stringFilterStatement tag="ProblemAlarm">
                <fieldName>additionalText</fieldName>
                <operator>contains</operator>
                <fieldValue>site down (BitError)</fieldValue>
        </stringFilterStatement>
    </allCondition>

  </anyCondition>
</topFilter>
```

The <topFilter name="Problem_BitError"> tag signifies the beginning of the filters definition for the “Problem_BitError” problem.

The following tags mean that conditions from block A **or** conditions from block B must be met, or both:

```
<anyCondition>
   <block A/>
   <block B/>
   …
</anyCondition>
```

The following tags mean that conditions from block A **and** conditions from block B must be met.

```
<allCondition>
   <block A/>
   <block B/>
   …
</allCondition>
```

The `<anyCondition>` and `<allCondition>` tags are recursive. A recursive tag is a tag that can be included in the same tag several times as shown below:

```
<allCondition>
  <allCondition>
    <allCondition>
```

The `<allCondition tag="TeMIP TT">` tag means that all alarms passing all the conditions included in this tag are associated to a specific Trouble Ticket System, HP TeMIP TT in this case.

The possible values for the tag name are given in the <troubleTicketActions> section of the `ProblemXmlConfig.xml` file. For more details on the `ProblemXmlConfig.xml` file, see section 5.3.2.

The following tags mean that alarms with the `[6] Remote Alarm OOS Threshold Exceeded` text in the `additionalText` field are considered Trigger Alarms for the "Problem_BitError" problem:

```
    <stringFilterStatement tag="Trigger">
      <fieldName>additionalText</fieldName>
      <operator>contains</operator>
      <fieldValue>[6] Remote Alarm OOS Threshold
Exceeded</fieldValue>
    </stringFilterStatement>
```

When	Alarm role	Description
tag="Trigger"	Trigger Alarm	An alarm that is an important symptom of a problem, and triggers the creation of a Problem Alarm
tag="SubAlarm"	Sub-Alarm	An alarm that is a symptom of a problem and is grouped under a Problem Alarm
tag="ProblemAlarm"	Problem Alarm	An alarm that summarizes the problem, and is readable by the operator
tag="SubAlarm,Problem Alarm"	Sub Problem Alarm	An alarm that is a Problem Alarm of a problem and a Sub-Alarm of another problem

Table 25 – PD: Possible roles for an alarm

If you want a Trigger Alarm to be used as a Problem Alarm (instead of creating a new one), the tag of the Trigger Alarm must be set as follows: `tag="Trigger, ProblemAlarm"`.

6.2.8 Configuring Value Pack settings

In the "`ValuePackConfiguration.xml`" file located in the `src/main/resources/valuepack/conf/` folder, only the sections related to mediation flow must be configured. Sections to be modified are highlighted in the following extract. For more details, see the "Value Pack definition file" chapter in [R1] *Unified Correlation Analyzer for Event Based Correlation Reference Guide.*

Extract of `ValuePackConfiguration.xml`

```
<mediationFlows 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>
<value>dynamic</value>
</actionParameter>
<actionParameter>
<key>operationContext</key>
<value>uca_network</value>
</actionParameter>
<actionParameter>
<key>operationContext</key>
<value>uca_pbalarm</value>
</actionParameter>
</flowCreation>
```

The "`context.xml`" file located in the `src/main/resources/valuepack/conf/` folder does not need to be modified, unless you want to customize:

- The enrichment example (the enrichment bean is highlighted in the following extract)
- The value for the generalBehaviorClassName property (for more details, see section 7.4.5)

For more details on the context.xml file, see the "Value Pack definition" chapter in [R1] *Unified Correlation Analyzer for Event Based Correlation Reference Guide*.

`context.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jms="http://www.springframework.org/schema/jms"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:amq="http://activemq.apache.org/schema/core"
xmlns:util="http://www.springframework.org/schema/util"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/jms
http://www.springframework.org/schema/jms/spring-jms.xsd
http://activemq.apache.org/schema/core
http://activemq.apache.org/schema/core/activemq-core.xsd">


<context:annotation-config />

<bean id="enrichment" class="com.acme.enrichment.EnrichmentProperties">
<property name="configurationFileName" value="Enrichment.xml" />
<property name="jmxManager" ref="jmxManager" />
</bean>

<bean id="problemsFactory" class="com.hp.uca.expert.vp.pd.core.ProblemsFactory">
<property name="problemPackageName" value="com.hp.uca.expert.vp.pd.problem." />
```

```
<property name="problemClassNamePrefix" value="Problem_" />
<property name="problemClassName" value="ProblemDefault" />
<property name="generalBehaviorClassName" value="MyGeneralBehaviorExample" />
<property name="xmlProblemClassName" value="XmlProblem" />
<property name="xmlGenericDefaultPrefix" value="XmlGeneric_" />
<property name="problemContextPackage" value="com.hp.uca.expert.vp.pd.core." />
</bean>


</beans>
```

6.2.9 Configuring specific settings

Main Policy is a configuration settings common to all problems defined in a Problem Detection Value Pack. These main configuration settings are defined in the `<mainPolicy>` XML tag.

Problem Policies are configuration settings which are specific to each problem defined in a Problem Detection Value Pack. These problem specific configuration settings are defined in the `<problemPolicy name="…">` XML tag.

Main Policy and Problem Policies are configured in the "`ProblemXmlConfig.xml`" file located in `src/main/resources/valuepack/conf/`.

Note that the XML schema of this file is named "`ProblemXmlConfig.xsd`" and it is located in the `src/main/resources/valuepack/conf/` folder.

You can also configure Transient Filtering, Actions, Trouble Tickets actions, Problem Alarm handling, and so on.

For more details, see section 5.3.2.

6.2.10 Customizing the default behavior for a specific problem

It is possible to assign basic customization directives for a specific problem, for example, for XmlGeneric_Synch as shown in the following extract. For more details, see section 7.4.1.

```
<problemPolicy name="XmlGeneric_Synch">
[...]
<strings><string key="ProblemAlarmAdditionalText">
<value><![CDATA[site down (XmlGeneric Synch)]]></value>
</string>
</strings>
```

6.3 Creating a simple Topology State Propagator Value Pack

The following sections describe how to create a simple Topology State Propagator Value Pack (TSP VP). For advanced TSP VP features, see Annex F.

6.3.1 Analyzing the topology to be used and the propagations to be detected

Before creating a Topology State Propagator Value Pack, it is essential to know on which topology the Value Pack will be based on. The topology can defer on the service model, on geographic criteria, or other criteria.

You must identify all the propagations that can be induced by a state update of the propagations that are underneath from a topology point of view. The state update can be triggered by several alarms and conditions, depending on the context of each of the propagations.

To continue with the medical analogy used in PD where alarms are the symptoms, problem is the disease, and the PD VP is the physician who diagnoses the disease (the problem) based on the symptoms (the received alarms):

- The correlated information containing the disease (the Problem Alarm) is received by TSP.
- TSP analyzes the information. If the disease is extremely contagious, it propagates it, resulting for example in epidemics (propagation on all upper level of the topology). If the disease is less contagious, it impacts only groups with low immunity systems (propagation on part of the upper level of the topology) or has no impact at all (no propagation in the topology).
- TSP analyzes secondary symptoms (Sub-Alarms), for example, the fact that children stopped going to the kindergarten because a lot of people were impacted before.
- TSP realizes Service Impact Analysis. For example, epidemics in a kindergarten can result in stopping the lessons activity (service) for a period.

The main steps of this initial TSP analysis can be summarized as follows:

1. Set the topology to establish the nodes and relationships. For details on the topology extension, see [R9] *Unified Correlation Analyzer for Event Based Correlation Topology Extension Guide*.
2. Do the SIA analysis: detect the services on which the impact must be computed.
3. If you use the RCA-SIA pattern of IM, list all potential alarms that can come from a standard Problem Detection scenario (in the same or in a different Value Pack). TSP can also be directly used for alarms coming directly from NMS, but HP recommends using it in conjunction with PD as an IM package.
4. List the propagations that might occur in the topology.
5. For each propagation, identify which alarms are associated with the propagation (note that an alarm can be associated with several propagations).

6.3.2 Configuring state computation

Default state computation is performed by the TSP framework. It is, however, possible to change this default computation, for example, to set your specific thresholds.

If you want to change it in Java, you must override the following method:

```
Boolean computeState(PropagationGroup group)
```

Note that the default behavior is to use the following service:

```
TP_Service_StateCalculation.computePercentageAvailability()
```

This service calculates the percentage of availability of the impacted node and deduces the state from the thresholdValues defined in configuration. For more details on this service computation, see section 8.2.

For more details on thresholdValues, see section 5.4.2.2.

6.3.3 Identifying the different types of alarms

Similarly to PD, the different types of alarms must be identified for TSP. As TSP is used in the IM on top of PD, a significantly reduced number of alarms is received by TSP because alarms are already grouped by PD into Problem Alarms.

Among all the alarms associated with a propagation, "Root Cause Alarms" must be separated from "Sub-Alarms". Root Cause Alarms are called as such because of their role in the propagation: they contribute to the trigger of re-computation of a propagation's state. Optionally, they can contribute to the creation, clearance, or update of a Service Alarm.

To continue with the medical analogy:

- Root Cause Alarms are the primary symptoms of the disease.
- Sub-Alarms are the secondary symptoms (for example, that a lot of children stopped going to the kindergarten).

At runtime, by default, a Topology State Propagator Value Pack considers that an instance of a propagation occurred if the state of the propagation was received. The computation of a state is an overridable method: the VP developer can modify the criteria for the creation of a state (for example, the number of Root Cause Alarms received with a critical status).

This default behavior can be customized (see section 8.1).

If the topology is set, and the possible impacting states and Root Cause Alarms are identified, as well as the propagations realizing the service impact analysis, then the filters of the Topology State Propagator Value Pack can be configured. Optionally, if the Service Alarm creation option is enabled, the Service Alarm and the "Sub-Alarms" must be identified and tagged in the filters.

Filters give logical criteria to distinguish different alarms and states. They allow distinguishing which alarm belongs to which propagation, and with which potential role (Root Cause Alarm, Sub-Alarm, or Service Alarm).

Filters are configured in an XML file.

For more details, see section 6.3.6 Defining the Filters and Annex E

Configuring Service Alarm creation

As opposed to Problem Alarm creation in PD, Service Alarm creation in TSP is optional. For each propagation, you must decide whether, at runtime, the TSP Value Pack creates a Service Alarm if the propagation occurs and several conditions are met. Service Alarms contain particular fields and can only be created by the framework.

The main steps of configuring Service Alarm creation can be summarized as follows:

1. Define the filters in the XML configuration file. For more details, see section 6.3.6.
2. Configure when the Service Alarm is created and cleared. For more details, see section 5.4.2.2.

6.3.4 Configuring Trouble Ticketing

For each propagation, you must decide whether, at runtime, upon occurrence of the propagation, the TSP Value Pack raises a trouble ticket. For more details, see section 5.4.2.2.

6.3.5 Considering if the default behavior needs to be modified

Topology State Propagator offers a default behavior that allows you to create a Value Pack without completing all the configuration phases described in the preceding sections. Nevertheless, almost any aspect of the default behavior can be customized if necessary.

For details on the default behavior and how to customize it, see Chapter 8.

6.3.6 Defining the Filters

Defining the filters is the most important step when creating a TSP Value Pack. Defining filters is not only about specifying which events (states, alarms, or other events) are relevant to the Value Pack. It is also about specifying which event is associated to which propagation, and what the role of each event is: state, Root Cause Alarm, Service Alarm, or Sub-Alarm.

Because a TSP Value Pack is an HP UCA for EBC Value Pack, defining filters for TSP Value Packs is done the same way as for any other HP UCA EBC Value Pack.

The definition of filters is done in the "`TopologyPropagation_filters.xml`" file located in `src/main/resources/valuepack/tp/`

As for PD, the filter file of a TSP Value Pack can include several "top filter" sections, one for each propagation to detect.

For TSP, a special top filter called *ReservedForGeneralBehavior* is defined in the "`TopologyPropagation_filters.xml`" file. This filter uses the extended mappers feature of HP UCA EBC V3.2. The following example shows the contents of *ReservedForGeneralBehavior* .

```xml
<topFilter name="ReservedForGeneralBehavior">
 <anyCondition>
  <anyCondition tag="PATTERN_Mappers">
   <allCondition tag="ComputeSourceUniqueIdMapper=NodeB_UniqueID_1">
    <instanceOfFilterStatement>
     <fullClassName>com.hp.uca.expert.alarm.AlarmCommon</fullClassName>
    </instanceOfFilterStatement>
    <stringFilterStatement>
     <fieldName>additionalText</fieldName>
     <operator>contains</operator>
     <fieldValue>PowerAntenna</fieldValue>
    </stringFilterStatement>
   </allCondition>
   <allCondition tag="ComputeSourceUniqueIdMapper=NodeB_UniqueID_2">
    <instanceOfFilterStatement>
     <fullClassName>com.hp.uca.expert.alarm.AlarmCommon</fullClassName>
    </instanceOfFilterStatement>
    <stringFilterStatement>
     <fieldName>additionalText</fieldName>
     <operator>contains</operator>
     <fieldValue>DIP_Failure</fieldValue>
    </stringFilterStatement>
   </allCondition>
  </anyCondition>
 </anyCondition>
</topFilter>
```

The tags used in the *ReservedForGeneralBehavior* top filter are defined in the "`TopologyPropagation_tags.xml`" file shown in the following example.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<tags xmlns="http://hp.com/uca/expert/filter/tags"
xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
 <groups>
  <group name="GeneralBehavior">
   <simpleTags>
   </simpleTags>
   <paramTags>
    <paramTag name="ComputeSourceUniqueIdMapper" default="computeSourceUniqueId"/>
   </paramTags>
  </group>

  <group name="TopologyPropagation">
   <simpleTags>
    <simpleTag name="ServiceAlarm"/>
    <simpleTag name="SubAlarm"/>
    <simpleTag name="RootCauseAlarm"/>
   </simpleTags>
   <paramTags>
   </paramTags>
  </group>

  <group name="GraphDB">
   <simpleTags>
   </simpleTags>
   <paramTags>
    <paramTag name="CypherQuery"
enum="GetCellFromNodeBOrBts,GetCustomerFromCell,GetNodeIdFromBtsOrNodeB,GetRelIdFromDigitalP
ath,GetNodeIdFromDigitalPath,GetNodeId,GetSite,GetPortLink"/>
   </paramTags>
  </group>
 </groups>
</tags>
```

In addition to the definition of the *ReservedForGeneralBehavior* top filter, propagations are also defined in the "`TopologyPropagation_filters.xml`" file. The following example shows the "top filter" section of a "`TopologyPropagation_filters.xml`" file for a propagation named "*Propagation_BtsOrNodeB*".

To see an example of a filter file that contains several "top filter" sections in order to detect several propagations, see the filter file of the Value Pack example in Annex E

```
<topFilter name="Propagation_BtsOrNodeB" tagsGroup="TopologyPropagation">
 <anyCondition>
  <anyCondition tag="PATTERN_SubAlarm">
   <anyCondition tag="SubAlarm">
    <allCondition>
     <instanceOfFilterStatement>
      <fullClassName>com.hp.uca.expert.alarm.AlarmCommon</fullClassName>
     </instanceOfFilterStatement>
     <stringFilterStatement>
      <fieldName>probableCause</fieldName>
      <operator>contains</operator>
      <fieldValue>houston we have a future sub service alarm!</fieldValue>
     </stringFilterStatement>
    </allCondition>
   </anyCondition>
  </anyCondition>
  <anyCondition tag="PATTERN_RootCause">
   <anyCondition tag="RootCauseAlarm">
    <allCondition>
     <instanceOfFilterStatement>
      <fullClassName>com.hp.uca.expert.alarm.AlarmCommon</fullClassName>
     </instanceOfFilterStatement>
     <stringFilterStatement>
      <fieldName>userText</fieldName>
      <operator>matches</operator>
      <fieldValue>
         <![CDATA[.*<action>UCA EBC.*</action><trigger>.*</trigger><group>.*</group>.*]]>
      </fieldValue>
     </stringFilterStatement>
     <stringFilterStatement>
      <fieldName>additionalText</fieldName>
      <operator>contains</operator>
      <fieldValue>PowerAntenna</fieldValue>
     </stringFilterStatement>
    </allCondition>
   </anyCondition>
  </anyCondition>
  <anyCondition tag="PATTERN_ServiceAlarm">
   <anyCondition tag="ServiceAlarm">
    <allCondition>
     <allCondition>
      <stringFilterStatement>
       <fieldName>userText</fieldName>
       <operator>matches</operator>
       <fieldValue>
         <![CDATA[.*<action>UCA
EBC.*</action><trigger>.*</trigger><propagationGroup>.*</propagationGroup>.*]]>
       </fieldValue>
      </stringFilterStatement>
     </allCondition>
     <anyCondition>
      <stringFilterStatement>
       <fieldName>additionalText</fieldName>
       <operator>contains</operator>
       <fieldValue>houston we have a propagation!</fieldValue>
      </stringFilterStatement>
     </anyCondition>
    </allCondition>
   </anyCondition>
  </anyCondition>
```

The `<topFilter name="Propagation_BtsOrNodeB">` tag signifies the beginning of the filters definition for the "*Propagation_BtsOrNodeB*" propagation.

The following tags mean that conditions from block A **or** conditions from block B must be met, or both:

```
<anyCondition>
    <block A/>
    <block B/>
    …
```

```
</anyCondition>
```

The following tags mean that conditions from block A **and** conditions from block B must be met:

```
<allCondition>
    <block A/>
    <block B/>
    …
</allCondition>
```

The <anyCondition> and <allCondition> tags are recursive. A recursive tag is a tag that can be included in the same tag several times as shown below:

```
<allCondition>
   <allCondition>
      <allCondition>
```

For more details on the `PropagationXmlConfig.xml` file, see section 6.3.8.

When	Alarm role	Description
tag="RootCauseAlarm"	Root Cause Alarm	An alarm that is an important root cause of a propagation, and that contributes to the creation of the service alarm
tag="SubAlarm"	Sub-Alarm	An alarm that contributes to the correlation of the propagation, and is grouped under the Service Alarm
tag="ServiceAlarm"	Service Alarm	An alarm that summarizes the propagation, and is readable by the operator

Table 26 – TSP: Possible roles for an alarm

6.3.7 Configuring Value Pack settings

In the "`ValuePackConfiguration.xml`" file located in the `src/main/resources/valuepack/conf/` folder, only the sections related to mediation flow must be configured. Sections to be modified are highlighted in the following extract. For more details, see the"Value Pack definition file" chapter in [R1] *Unified Correlation Analyzer for Event Based Correlation Reference Guide*.

Extract of `ValuePackConfiguration.xml`

```
<mediationFlows 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>
```

```xml
<key>flowType</key>
<value>dynamic</value>
</actionParameter>
<actionParameter>
<key>operationContext</key>
<value>uca_network</value>
</actionParameter>
<actionParameter>
<key>operationContext</key>
<value>uca_pbalarm</value>
</actionParameter>
</flowCreation>
```

The "`context.xml`" file located in the `src/main/resources/valuepack/conf/` folder does not need to be modified, unless you want to customize:

- The propagationsFactory (the propagationsFactory bean is highlighted in the following extract below)
- The value for the generalBehaviorClassName property (for more details, see section 7.4.5)

For more details on the context.xml file, see the "Value Pack definition" chapter in [R1] *Unified Correlation Analyzer for Event Based Correlation Reference Guide.*

`context.xml`

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jms="http://www.springframework.org/schema/jms"
xmlns:p="http://www.springframework.org/schema/p"
xmlns:context="http://www.springframework.org/schema/context"
xmlns:amq="http://activemq.apache.org/schema/core"
xmlns:util="http://www.springframework.org/schema/util"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd
http://www.springframework.org/schema/jms
http://www.springframework.org/schema/jms/spring-jms.xsd
http://activemq.apache.org/schema/core
http://activemq.apache.org/schema/core/activemq-core.xsd">


<context:annotation-config />


<bean id="propagationsFactory"
class="com.hp.uca.expert.vp.tp.core.PropagationsFactory">
<property name="propagationPackageName" value="com.hp.uca.expert.vp.pd.propagation."
/>
<property name="propagationClassNamePrefix" value="Propagation_" />
<property name="propagationClassName" value="PropagationDefault" />
<property name="generalBehaviorClassName" value="GeneralBehaviorDefault" />
<property name="xmlPropagationClassName" value="XmlPropagation" />
<property name="xmlGenericDefaultPrefix" value="Xml_" />
<property name="propagationContextPackage" value="com.hp.uca.expert.vp.tp.core." />
</bean>


</beans>
```

6.3.8 Configuring specific settings

Main Policy is a configuration settings which is common to all propagations defined in a TSP Value Pack. These main configuration settings are defined in the `<mainPolicy>` XML tag.

Propagation Policies are configuration settings which are specific to each propagation defined in a TSP Value Pack. These propagation specific configuration settings are defined in the `<propagationPolicy name="…">` XML tag.

Policies are configured in the "`PropagationXmlConfig.xml`" file located in `src/main/resources/valuepack/conf/`.

Note that the XML schema of this file is named "`PropagationXmlConfig.xsd`" and it is located in the `src/main/resources/valuepack/conf/` folder.

For more details, see section 5.4.2.

6.4 Creating a Standard IM VP

The objective of this chapter is to list and briefly explain the steps required to create a meaningful Inference Machine Value Pack.

Unfortunately, this chapter is not available at the moment.

For a good example of a standard IM VP, refer to the IM example delivered with the IM SDK.

Chapter 7

Advanced features of Problem Detection

With the basic configuration described in section 5.3, a Problem Detection Value Pack runs with a default behavior.

This default behavior is rich in the sense that, in many cases, it does not have to be altered or extended. However, for the use cases where modification or extension is required, Problem Detection offers the flexibility to change the default behavior.

The default behavior is presented in section 7.1. The ways to customize the default behavior are described in section 7.4.

7.1 Default behavior

The Problem Detection framework is a set of Java libraries, with some Java classes that can be extended and methods that can be overridden in order to change the default behavior of Problem Detection Value Packs.

Each of the following methods has a default behavior, which can be customized by overriding the method.

For the default behavior of all these methods, see the IM Javadoc part of the SDK [R6]. The implementation code of these methods is available in the example value pack delivered as part of the Problem Detection Dev Kit (see section vp-examples\pd-example\src\test\resources) .The code of each of these methods is executed for every problem for which the method is applicable and can be overridden by the value pack developer.

Section 7.1.1 presents an example. The subsequent sections present the available interfaces.

7.1.1 Example

An example workflow of the different methods triggered in the case of a Network State Update alarm is shown in the sequence diagram in Figure 20. In this example, an alarm clearance is managed for acontext where alarm 1:

- Is Problem Alarm in group1 of Problem1 (PB1).
- Is Sub Alarm in group2 of Problem2 (PB2).
- Has no role for any of the groups of Problem3 (PB3).

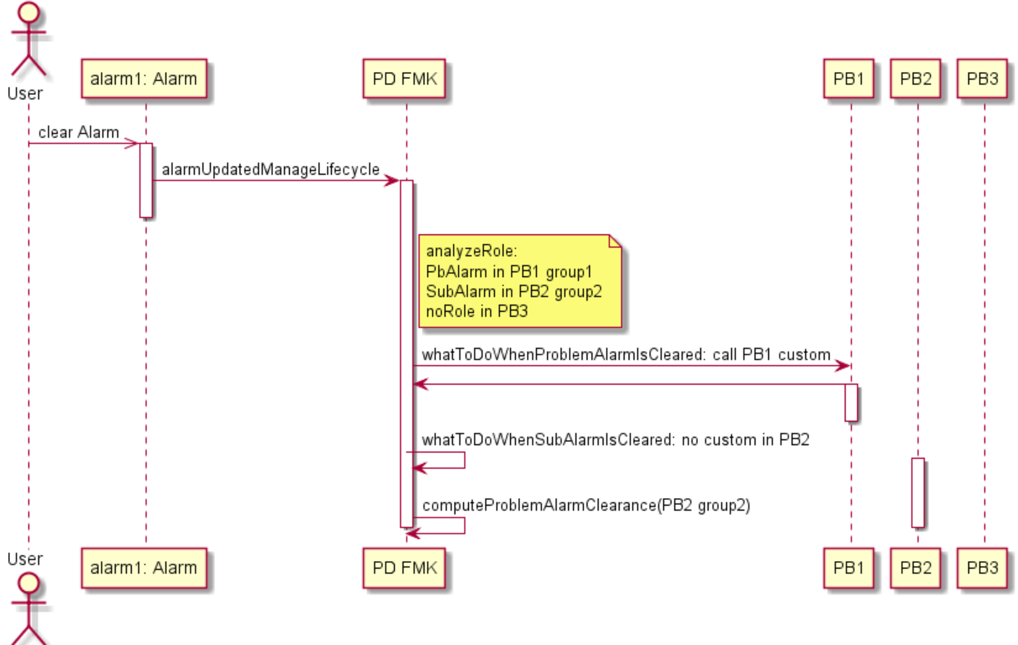


Figure 20 - Alarm clearance sequence diagram example

Figure 21 shows the event grouping before the alarm1 clearance is received.

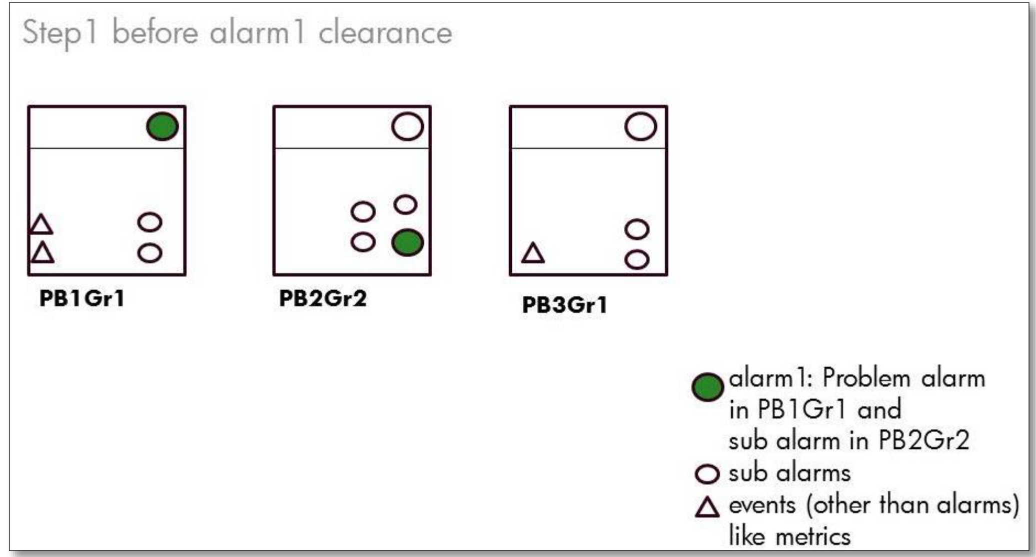


Figure 21 – PD Alarm clearance example: PD group updates Step1

The alarm1 clearance is received and, according to the sequence diagram in Figure 20, a number of methods are called by the PD framework. As a result:

- The Problem Alarm in group1 of PB1 is cleared and removed from the Working Memory.
- Concerning group2 in PB2, alarm1 has a role as Sub-Alarm but its clearance results in the computation of group2 from PB2 clearance. In the current example, assume that the clearance has an impact only on the severity change but the Problem Alarm in PB2 group2 is still present.
- Concerning group1 in PB3, there is no impact.

Figure 22 shows the event grouping updates after the alarm1 clearance is received.

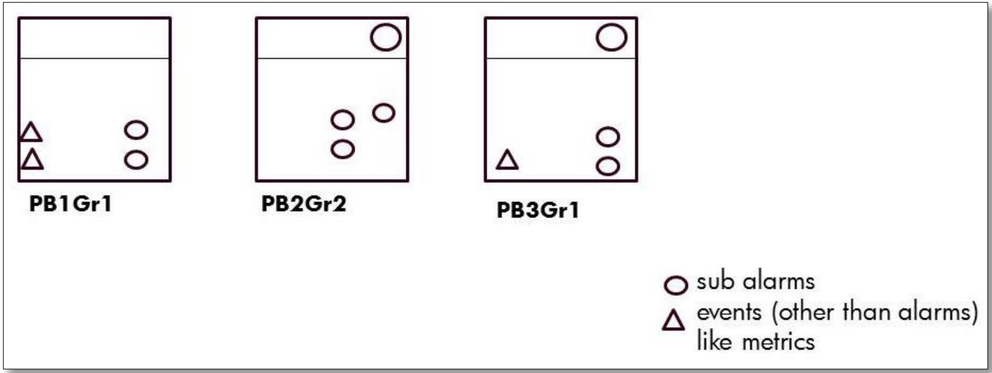
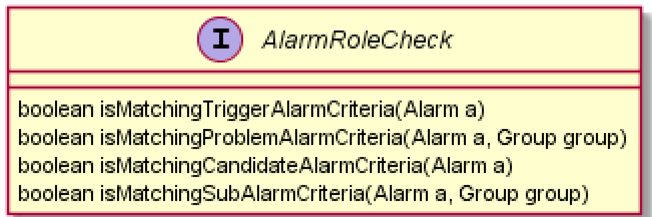


Figure 22 - PD Alarm clearance example: PD group updates Step2

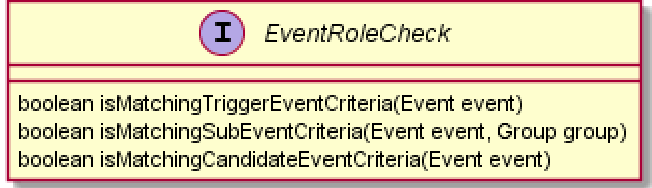
7.1.2 AlarmRoleCheck

The following methods are used to check the role of alarms.



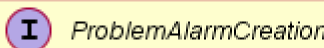
7.1.3 EventRoleCheck

The following methods are used to check the role of events.



7.1.4 ProblemAlarmCreation

The following methods are related to Problem Alarm creation.



The following method is used to check if a Problem Alarm must be created:

`isAllCriteriaForProblemAlarmCreation(Group)`

The following methods are used during Problem Alarm Creation:

`calculateReferenceAlarm(Group)`

`calculateProblemAlarmManagedEntity(Group)`

`calculateProblemAlarmAlarmType(Group)`

`calculateProblemAlarmProbableCause(Group)`

`calculateProblemAlarmAdditionalText(Group)`

`calculateProblemAlarmOperatorNote(Group)`

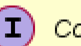
`calculateProblemAlarmUserText(Group, Action)`

`calculateProblemAlarmEventTime(Group)`

`calculateProblemAlarmOtherAttribute(Action)`

7.1.5 CommonEntityCheck

The following methods are used to calculate Information for optimizations.



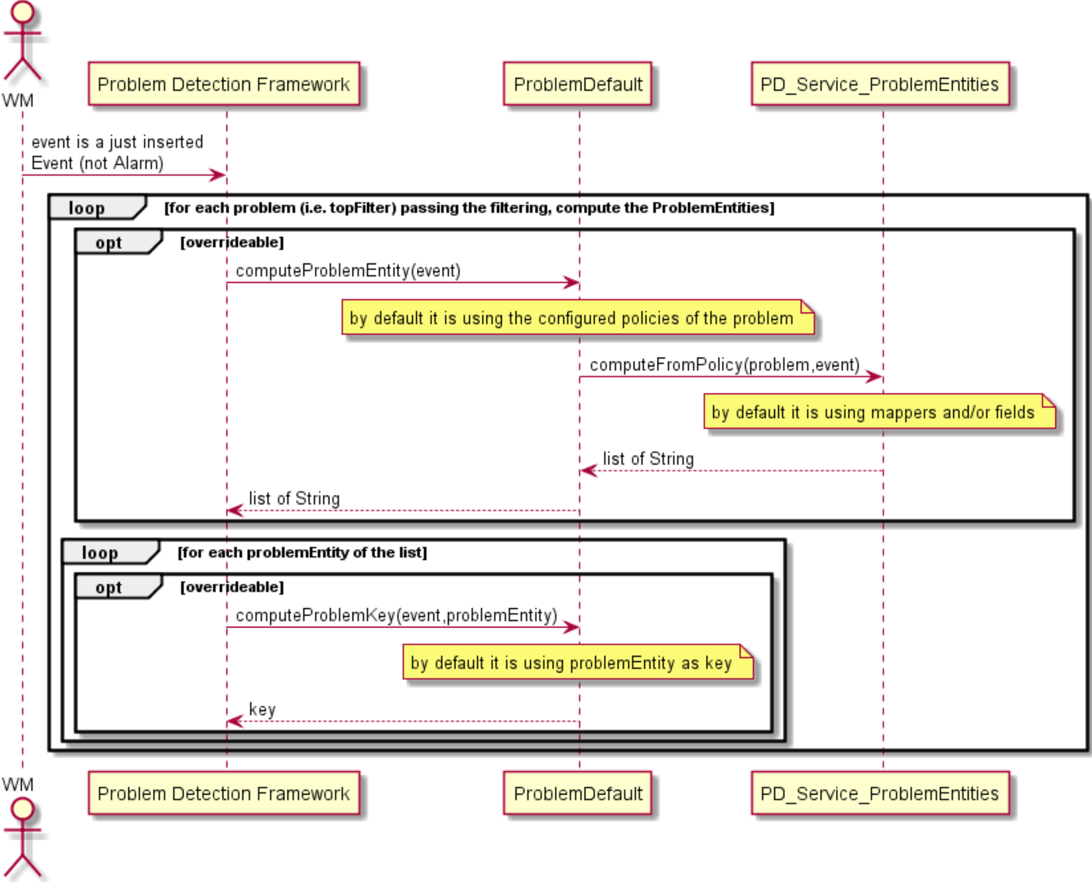


Figure 23 - computeProblemEntity (Event event)

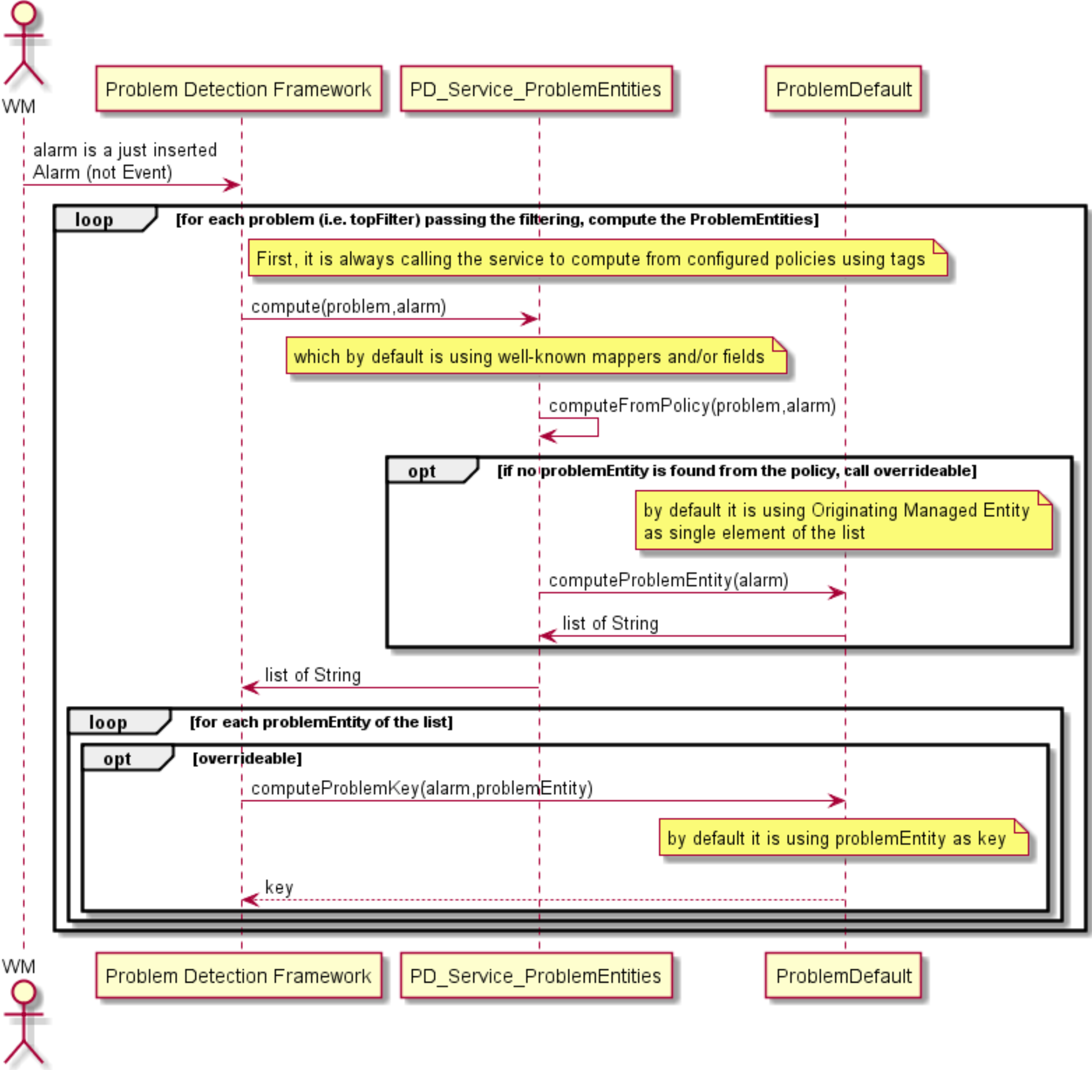
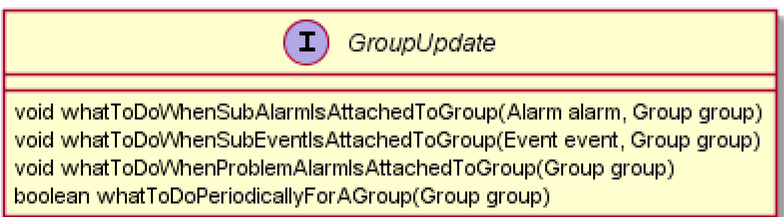


Figure 24 - computeProblemEntity (Alarm alarm)

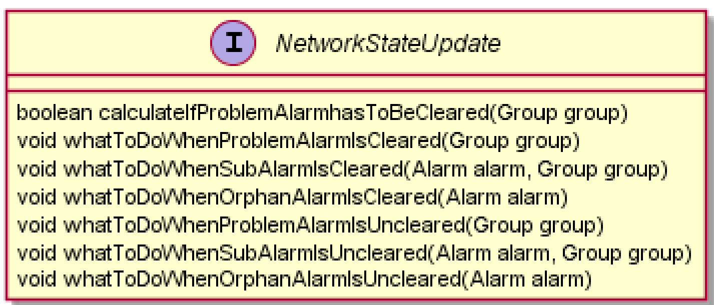
7.1.6 GroupUpdate

The following methods are used to manage the group life cycle and its associated alarms.



7.1.7 NetworkStateUpdate

The following methods are used to manage alarm network state updates.



Alarm Network State Changes

WM

Problem Detection Framework

ProblemDefault

alarm state has changed

alt [networkState = CLEARED]

loop [for all groups where alarm is ProblemAlarm]

whatToDoWhenProblemAlarmIsCleared(group)

loop [for all groups where alarm is SubAlarm or Trigger or Candidate]

possibly compute ProblemAlarm clearance for the group

whatToDoWhenSubAlarmIsCleared(alarm, group)

loop [for all problems where alarm has no specific role]

whatToDoWhenOrphanAlarmIsCleared(alarm)

[networkState = NOT_CLEARED]

loop [for all groups where alarm is ProblemAlarm]

whatToDoWhenProblemAlarmIsUncleared(group)

loop [for all groups where alarm is SubAlarm or Trigger or Candidate]

possibly compute ProblemAlarm clearance for the group

whatToDoWhenSubAlarmIsUncleared(alarm, group)

loop [for all problems where alarm has no specific role]

whatToDoWhenOrphanAlarmIsUncleared(alarm)

WM

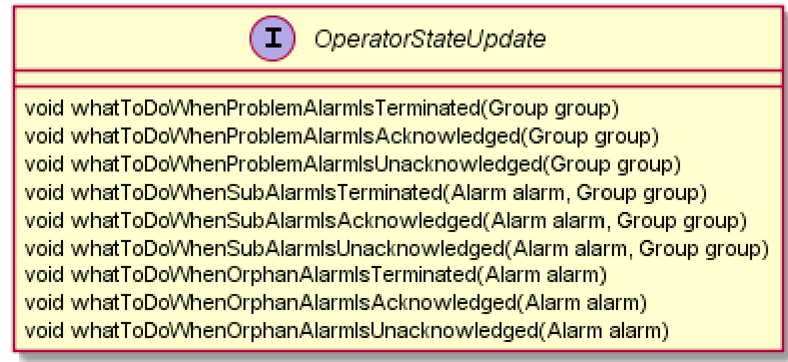
Problem Detection Framework

ProblemDefault

Figure 25 - Alarm network state changes

7.1.8 OperatorStateUpdate

The following methods are used to manage alarm operator state updates.



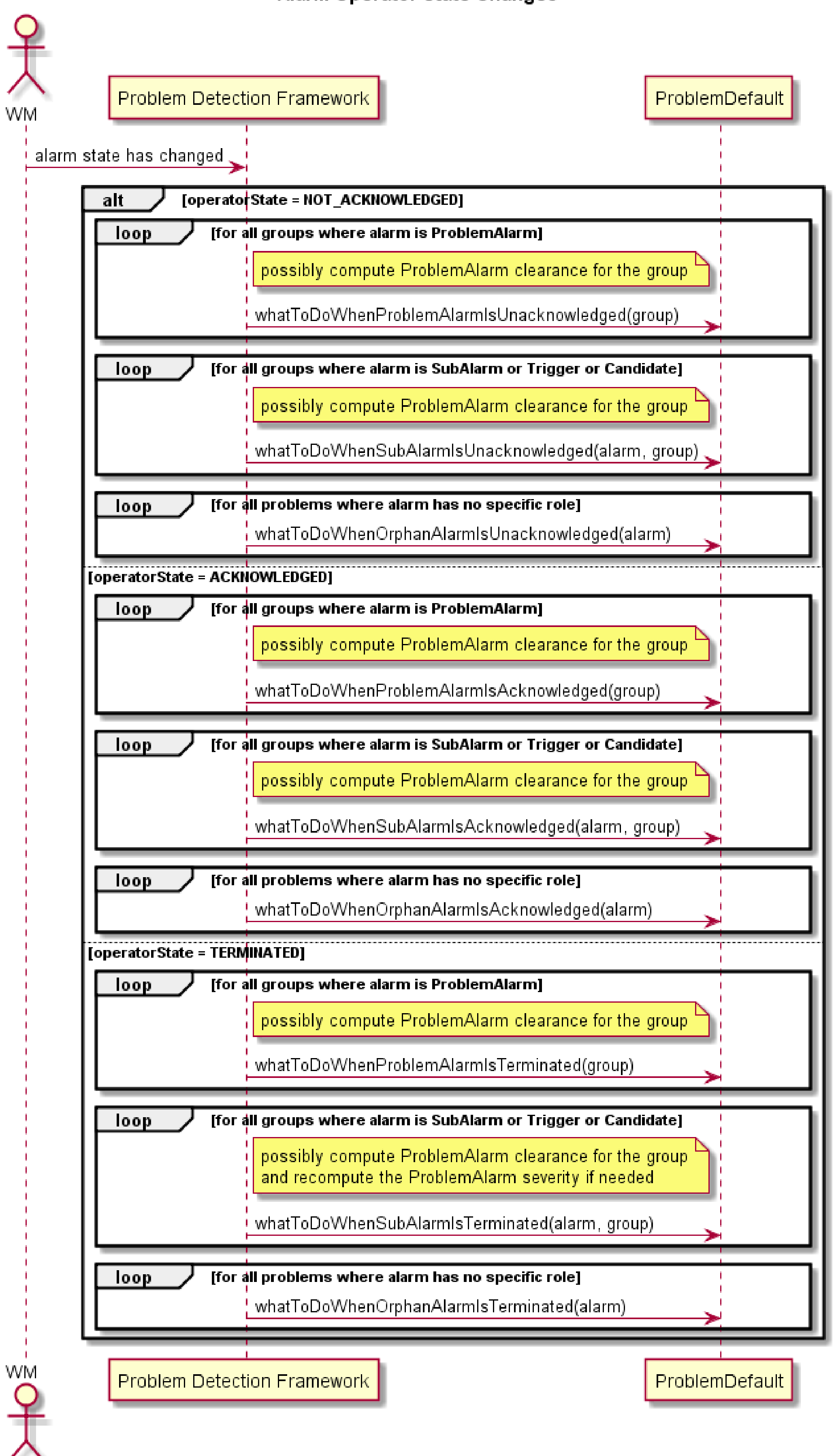
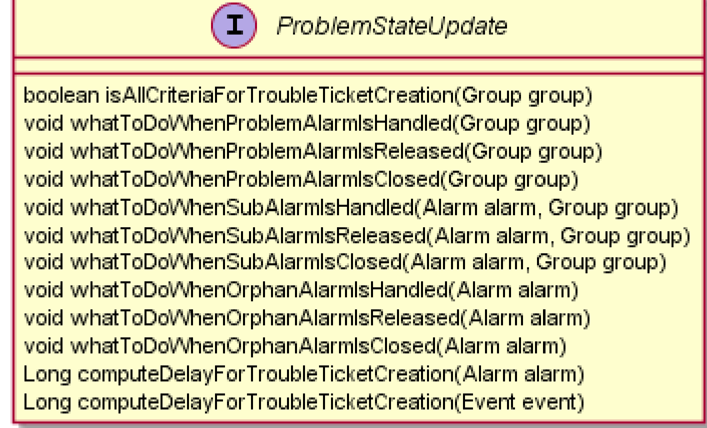

Alarm Operator State Changes
WM
Problem Detection Framework
ProblemDefault
alarm state has changed
alt
[operatorState = NOT_ACKNOWLEDGED]
loop
[for all groups where alarm is ProblemAlarm]
possibly compute ProblemAlarm clearance for the group
whatToDoWhenProblemAlarmIsUnacknowledged(group)
loop
[for all groups where alarm is SubAlarm or Trigger or Candidate]
possibly compute ProblemAlarm clearance for the group
whatToDoWhenSubAlarmIsUnacknowledged(alarm, group)
loop
[for all problems where alarm has no specific role]
whatToDoWhenOrphanAlarmIsUnacknowledged(alarm)
[operatorState = ACKNOWLEDGED]
loop
[for all groups where alarm is ProblemAlarm]
possibly compute ProblemAlarm clearance for the group
whatToDoWhenProblemAlarmIsAcknowledged(group)
loop
[for all groups where alarm is SubAlarm or Trigger or Candidate]
possibly compute ProblemAlarm clearance for the group
whatToDoWhenSubAlarmIsAcknowledged(alarm, group)
loop
[for all problems where alarm has no specific role]
whatToDoWhenOrphanAlarmIsAcknowledged(alarm)
[operatorState = TERMINATED]
loop
[for all groups where alarm is ProblemAlarm]
possibly compute ProblemAlarm clearance for the group
whatToDoWhenProblemAlarmIsTerminated(group)
loop
[for all groups where alarm is SubAlarm or Trigger or Candidate]
possibly compute ProblemAlarm clearance for the group
and recompute the ProblemAlarm severity if needed
whatToDoWhenSubAlarmIsTerminated(alarm, group)
loop
[for all problems where alarm has no specific role]
whatToDoWhenOrphanAlarmIsTerminated(alarm)
WM
Problem Detection Framework
ProblemDefault

Figure 26 - Alarm operator state changes

7.1.9 ProblemStateUpdate

The following methods are used to manage the Trouble Ticket life cycle when related to:

- A Problem Alarm
- A SubAlarm
- An Orphan Alarm



7.1.10 AttributeUpdate

The following methods are used to manage a Severity or an Attribute Update of:

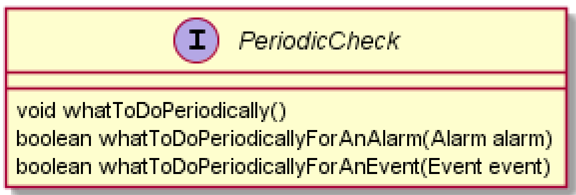
- A Problem Alarm
- A SubAlarm
- An Orphan Alarm

AttributeUpdate

```
void whatToDoWhenProblemAlarmSeverityHasChanged(Group group)
void whatToDoWhenSubAlarmSeverityHasChanged(Alarm alarm, Group group)
void whatToDoWhenOrphanAlarmSeverityHasChanged(Alarm alarm)
PerceivedSeverity calculateProblemAlarmSeverity(Group group)
void whatToDoWhenProblemAlarmAttributeHasChanged(Group group, AttributeChange attributeChange)
void whatToDoWhenSubAlarmAttributeHasChanged(Alarm alarm, Group group, AttributeChange attributeChange)
void whatToDoWhenOrphanAlarmAttributeHasChanged(Alarm alarm, AttributeChange attributeChange)
```

7.1.11 PeriodicCheck

The following methods are used to manage periodic checks for alarms and events.



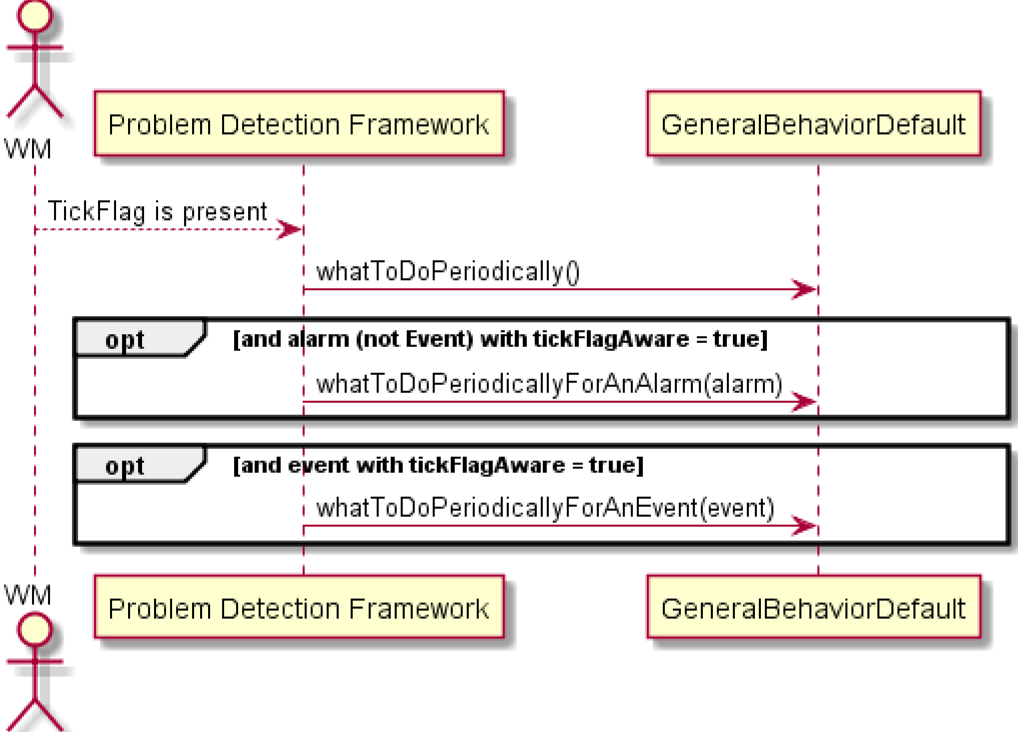
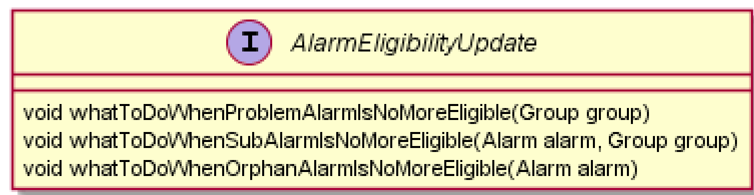


Figure 27 - Periodic checks

7.1.12 AlarmEligibilityUpdate

The following methods are used to manage alarm eligibility updates.



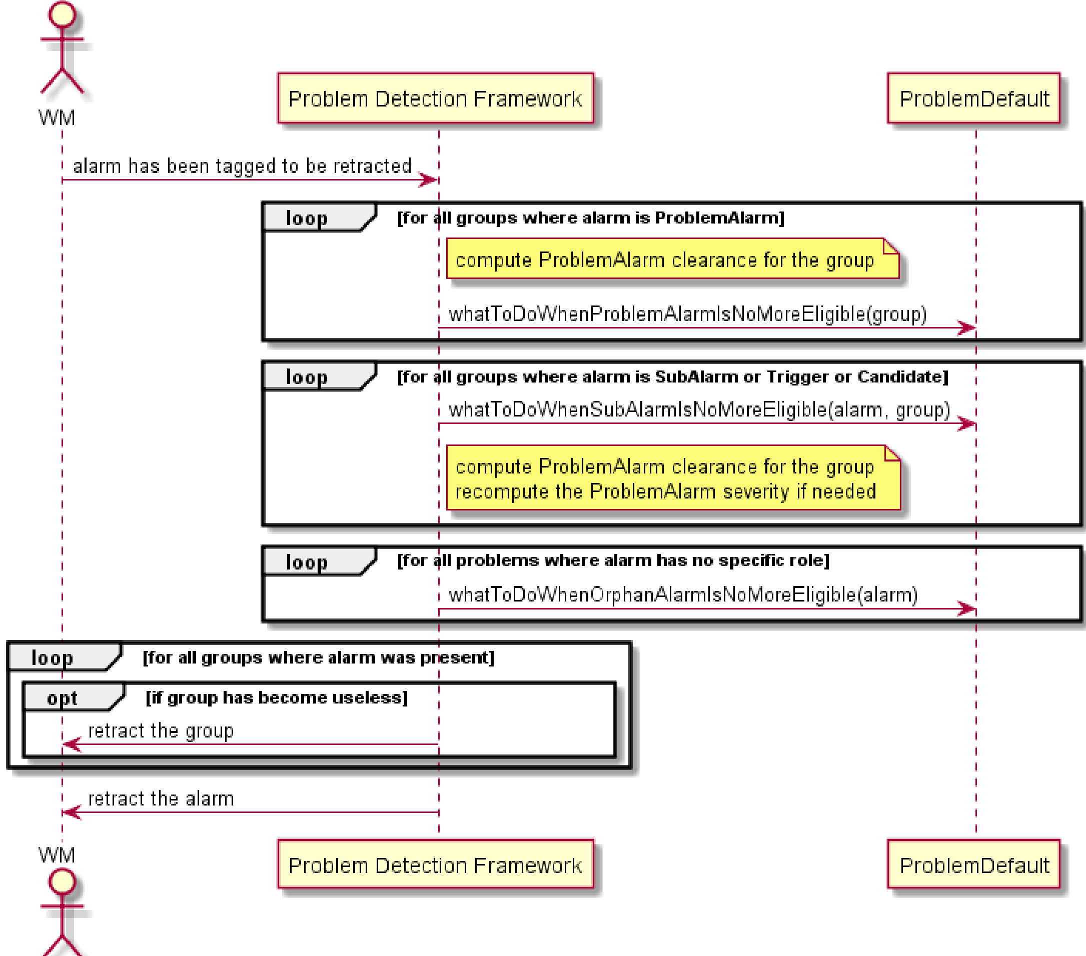
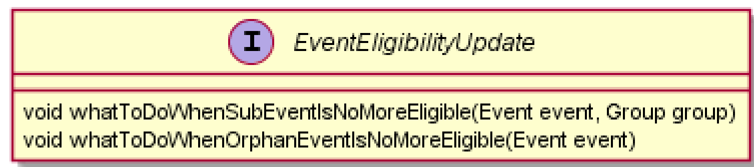


Figure 28 - Alarm eligibility update

7.1.13 EventEligibilityUpdate

The following methods are used to manage event eligibility updates.



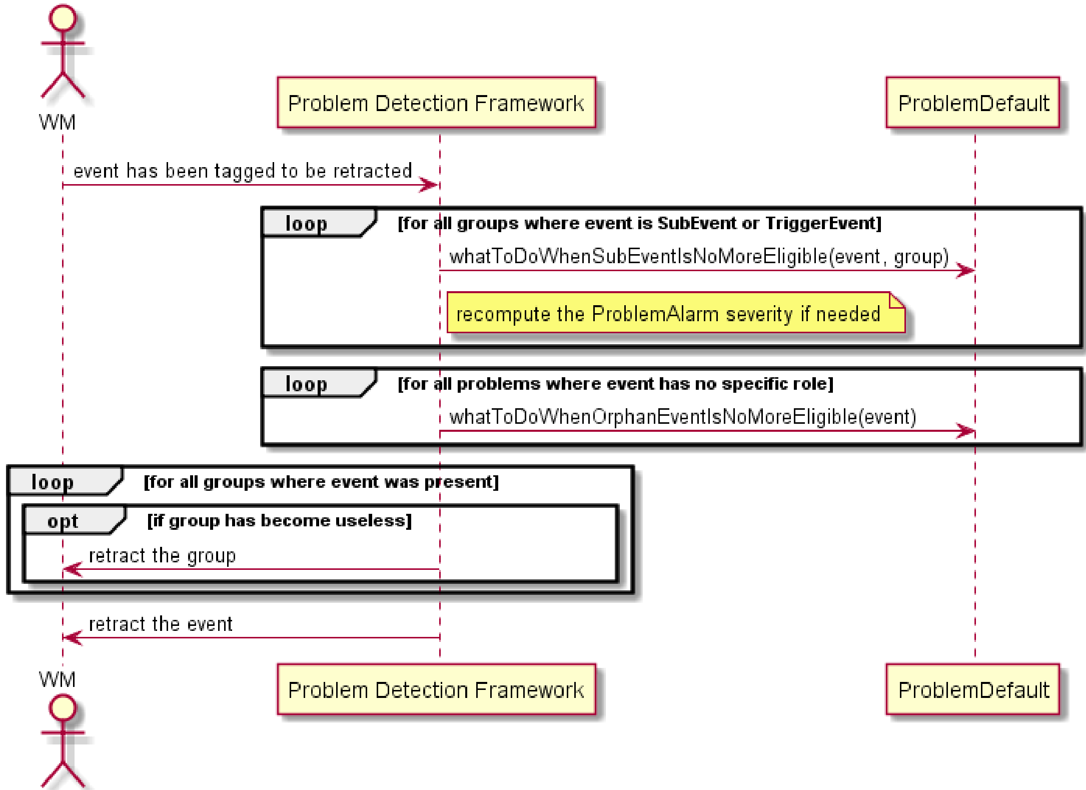
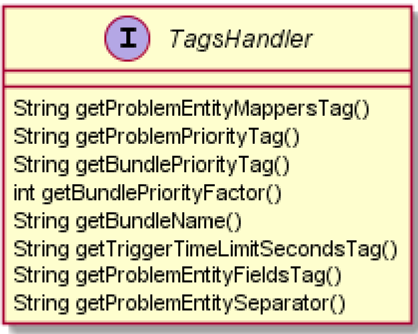


Figure 29 - Event eligibility update

7.1.14 TagsHandler

Tags handling features are introduced in HP UCA EBC V3.2.

The following methods are used to control the tag names used by the Problem Detection filter tags.



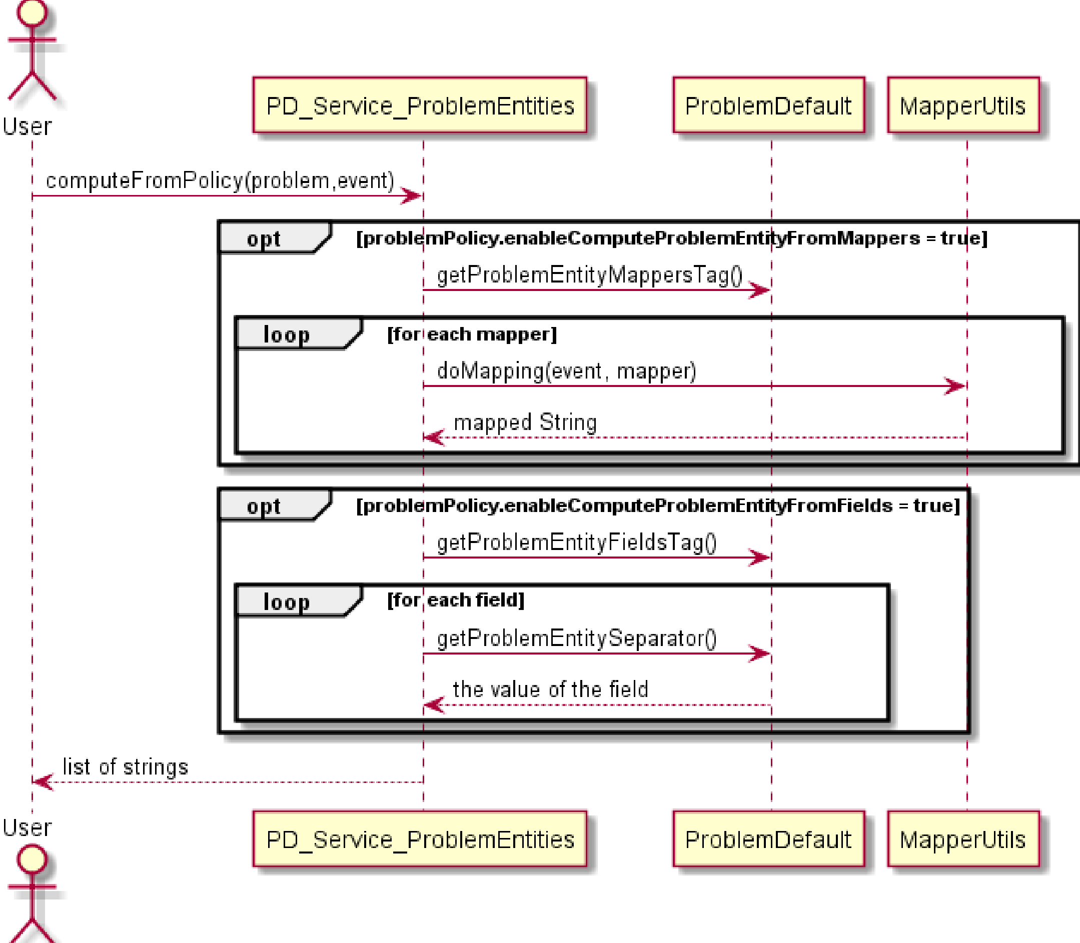


Figure 30 - Tags handling for computeProblemEntity()

Tags Handling for computeGroupPriority(Event)

User

ProblemDefault

computeGroupPriority(event)

getBundlePriorityTag()

getProblemPriorityTag()

opt [if at least one of above tag is present in passing filters]

getBundlePriorityFactor()

priority is computed from values of tags if they are present such as
(BundlePriority * PriorityFactor) + ProblemPriority

computed priority

User

ProblemDefault

Figure 31 - Tags handling for computeGroupPriority(Event)

Figure 32 - Tags handling for computeTimeWindow(Event)

7.2 Supported generic events other than alarm types

Problem Detection V3.2 is able to correlate and group generic events. The Trigger of a PD correlation group can be an event.

Most methods are applicable therefore for the event type as parameter and not only on the alarm type. As a result, some methods are now deprecated.

A Value Pack example using PD with events types other than alarms is provided with the IM SDK, described in Annex D

7.3 Computing Problem Information

When a new alarm is received by Problem Detection, Problem information is computed in two different ways, which are described in the following subsections.

7.3.1 Problem information computing when Problem Detection is topology-aware

The following conditions are checked by default:

- `MainPolicy.enableTopoAccess` attribute is set to true.
- the `CypherQuery` tag is present in the passing filter tags parameters and provides the name of the Cypher Query to execute.

If these conditions are met, both the `GeneralBehaviourDefault.computeSourceUniqueId (Event event)` and `ProblemDefault.computeDbRecords(String dbUniqueIdReference, Event event)` methods are used to compute the Problem Alarm information.

Note

The above default conditions can be changed by overriding the `ProblemDefault.isAllowingDbAccess(Event event)` method.

If computing is successful, the `ProblemDefault.computeProblemEntity(Event event)` method is not used.

7.3.2 Problem information computing in default case (non-topology aware)

If the scenario described in 7.3.1 does not apply or fails, the *ProblemDefault.computeProblemEntity(Event event)* method is used for computing.

7.3.3 ProblemXmlConfig schema changes

7.3.3.1 Namespace

Some elements defined in the `ProblemXmlConfig.xml` configuration file are now provided by the common schema defined in the IM common library, As a result, the namespace of some elements is different from others.

To resolve this issue, existing configuration files must be migrated. See the Appendix for more information on the migration procedure.

7.3.3.2 MainPolicy

The following new attributes are available:

`enablePrioritySort:` Defines whether the groups are sorted in priority order or not. Boolean flag, default setting is false.

`multipleParentSupport:` Defines whether an alarm grouping will send the parent relationship only for the highest priority parent (false), or for each of the Problem Alarms where this alarm is grouped (true). Default setting is true.

`enableTopoAccess:` Defines whether to use topologyAccess when computing information for Problem Alarms (true) or not (false).

Topology license is required to set this attribute to true. (Neo4j database is used). When set to true, the `computeSourceUniqueID(Event event)` and `computeDBRecords()` methods are called during the workflow) and the `computeProblemEntity(Event event)` method is not called.

7.3.4 ProblemPolicy

The following new attributes are available:

`enableComputeProblemEntityFromMappers`: Enables the use of calling mappers in `computeProblemEntity()` when set to true. Default setting is true.

`enableComputeProblemEntityFromFields`: When true, enables computation of fields key/value pairs in `computeProblemEntity()`. Default setting is false.

`computeProblemEntityFromFields`: Configures of the FieldsChooser element, which is a sequence of fields to be used as keys. It is called in `computeProblemEntity()` when computation of fields key/value pairs is enabled and whenthe ComputeProblemEntityFields tag is not used.

7.3.5 ProblemDefault.computeProblemEntity(Event event)

This method was introduced in version 3.2 and takes an Event as its parameter. It is called by the existing ***computeProblemEntity(Alarm alarm)*** method.

The default behavior of the new ***computeProblemEntity(Event)*** is enhanced to better satisfy end-user needs.
It executes the procedures described in (7.3.5.1, 0 and 7.3.5.3) in respective order.

7.3.5.1 Using extended mappers

The `ProblemDefault.computeProblemEntity(Event event)` method uses the extended mappers feature introduced in HP UCA EBC version 3.2.

When an event is received by the Problem Detection Value Pack, it is checked against the presence of the filter tag `ComputeProblemEntityMappers` which is a parameter tag that contains the name of the mappers to use for computing the problem entity.

If the tag is present in the incoming filtered alarm and the mappers referenced in this tag are well defined, the mappers are executed against the incoming alarm. The result of each mapper is used as an element of the problem entity list returned by this function.

The usage of extended mappers is automatically taken into account.

Note

- Mapper usage can be disabled by setting the corresponding `ProblemPolicy.enableComputeProblemEntityFromMappers` attribute to false in `ProblemXmlConfig.xml` file. By default, this setting considered as true.
- Mapper names contained in the `ComputeProblemEntityMappers` tag must be separated by a dot.
- To change the name of the used filter tag, override the `getProblemEntityMappersTag()` method of your problem.

Since V3.3, a single mapper can be used to return multiple problem entities. To have this feature enabled, you will need to declare the attribute separator in the definition of the mapper.

As per example below:

> Suppose you have a custom field "servers" that may contain several servers separated with commas. If you need each of the server be a separate problemEntity, then declare the mapper as per example below.

```
<mapper name="getServers" separator=",">
        <extract>
                <fieldName>servers</fieldName>
                <matcher>(.*)</matcher>
                <mappedTo>$1</mappedTo>
        </extract>
</mapper>
```

7.3.5.2 Directly mapping alarm fields as key/value pairs

The `ProblemDefault.computeProblemEntity(Event event)` can utilize the fields of the alarm computed as key/value pairs. The operation of this function is described as follows. Described options are evaluated in the following order:

1. Using a well-known tag

If the `ComputeProblemEntityFields` filter tag is present in the incoming alarm filtered tags, it must contain the name of the fields to use for computing the problem entity. Each field described in this tag is checked against its presence in the alarm and the resulting problemEntity is computed as `$field.name$separator$field.value`.

Note

- The computation of the key/value pairs is enabled by setting the corresponding `ProblemPolicy.enableComputeProblemEntityFromFields` attribute to true in `ProblemXmlConfig.xml` file. By default, this setting is considered false – this feature is not enabled by default.
- Each field name in the `ComputeProblemEntityFields` tag must be separated by a dot.
- To change the name of the used filter tag, override the `getProblemEntityMappersTag()` method of your problem.
- To change the value of `$separator`, override the `getProblemEntitySeparator()` method of your problem. By default, it is an equation sign (=).

2. Using version 3.2 policies

The corresponding `ProblemPolicy.computeProblemEntityFromFields` element is defined in the `ProblemXmlConfig.xml` file and is used to compute the problem entity. This policy defines a sequence of XML field elements and a keyValueSeparator XML element which is by default an equation sign (=).

Each field described in this XML element is used as an element of the problem entity list returned by the `computeProblemEntity()` method. Each field defines either a `tagName` or a `fieldName`.

- If `tagName` is defined, it must correspond to a tag that is present if the incoming alarm filtered tags define the field of the alarm to take into account. It is then checked against its presence in the alarm filtered tags and the resulted `problemEntity` is computed as `$alarmField$keyValueSeparator$alarmField.value`, where `$alarmField` must be present in the alarm and is equivalent to `$field.key.tagName.value`.

- If `fieldName` is defined, it corresponds directly to the field of the alarm taken into account. The field name is then checked against its presence in the alarm and the resulting `problemEntity` is computed as `$fieldName$keyValueSeparator$fieldName.value`.

Note

- The computation of the key/value pairs is enabled by setting the corresponding `ProblemPolicy.enableComputeProblemEntityFromFields` attribute to true in `ProblemXmlConfig.xml` file. By default, this setting is considered false – this feature is not enabled by default.
- If the filter tag `ComputeProblemEntityFields` is present in the incoming alarm filtered tags, it supersedes the policy and the policy is not used.
- You can ignore a specific value for each field using the `valueIgnored` XML element associated with it.

7.3.5.3 Default mode

If none of methods described in 7.3.5.1 and 0 are used, the function returns the originating managed entity of the incoming alarm.

7.3.5.4 Modifying examples

The pd-example value pack contains the updated classes `Problem_Synch` and `Problem_BitError`. These classes demonstrate the usage of the extended mappers feature to compute their problem entity based on bsc and bts identifiers. The `computeProblemEntity()` function was removed from these classes, and the `getBscBtsFromUserText` mapper is used instead.

7.3.6 GeneralBehaviourDefault.computeSourceUniqueId(Event event)

This method is used to calculate the unique identifier from information source stored in the event. It is called when Problem Detection is topology-aware, that is if the `MainPolicy.enableTopoAccess` attribute is set to true. In this case, a special filter must be defined with the `ReservedForGeneralBehavior` as the filter name.

This filter uses the `ComputeSourceUniqueIdMapper` tags to compute the source unique Id. When mappers are defined in the topFilter called `ReservedForGeneralBehavior`, Problem Detection calls the `computeSourceUniqueId(Event)` method.

An example filter and mapper is as follows:

```
<topFilter name="ReservedForGeneralBehavior">
 <anyCondition>
  <anyCondition tag="PATTERN_Mappers">
   <allCondition tag="ComputeSourceUniqueIdMapper=NodeB_UniqueID_1">
    <instanceOfFilterStatement>
     <fullClassName>com.hp.uca.expert.alarm.AlarmCommon</fullClassName>
    </instanceOfFilterStatement>
    <stringFilterStatement>
     <fieldName>additionalText</fieldName>
     <operator>contains</operator>
     <fieldValue>PowerAntenna</fieldValue>
    </stringFilterStatement>
   </allCondition>
   <allCondition tag="ComputeSourceUniqueIdMapper=NodeB_UniqueID_2">
    <instanceOfFilterStatement>
     <fullClassName>com.hp.uca.expert.alarm.AlarmCommon</fullClassName>
    </instanceOfFilterStatement>
    <stringFilterStatement>
     <fieldName>additionalText</fieldName>
     <operator>contains</operator>
     <fieldValue>DIP_Failure</fieldValue>
    </stringFilterStatement>
   </allCondition>
  </anyCondition>
 </anyCondition>
</topFilter>

<mapper name='NodeB_UniqueID_1'>
 <pattern>
  <expression>[btsID]~[location]</expression>
  <matcher>(.*)</matcher>
  <mappedTo>$1</mappedTo>
 </pattern>
</mapper>
```

7.3.7 ProblemDefault.computeDbRecords(String dbUniqueIdReference, Event event)

This method calculates the Neo4j query, which is executed to retrieve the database records for the database id reference of the Event. It is called by the Problem Detection framework when the `MainPolicy.enableTopoAccess` attribute is set to true and the `CypherQuery` tag is present.

An example filter and mapper is as follows:

```
 <anyCondition tag="ProblemAlarm,CypherQuery=GetCellFromNodeBOrBts">
  <allCondition>
   <instanceOfFilterStatement>
    <fullClassName>com.hp.uca.expert.alarm.AlarmCommon</fullClassName>
   </instanceOfFilterStatement>
   <stringFilterStatement>
    <fieldName>userText</fieldName>
    <operator>matches</operator>
    <fieldValue>
     <![CDATA[.*<action>UCA
EBC.*</action><trigger>.*</trigger><group>.*</group>.*]]>
    </fieldValue>
   </stringFilterStatement>
   <stringFilterStatement>
    <fieldName>additionalText</fieldName>
    <operator>contains</operator>
    <fieldValue>PowerAntenna</fieldValue>
   </stringFilterStatement>
  </allCondition>
 </anyCondition>

<cypherQuery name='GetCellFromNodeBOrBts'>
  <query>
   <![CDATA[START startNode=node:NodeBsByUniqueId(uniqueId = {nodeUniqueId})
MATCH (startNode)-[relation:ServingCell]->(endNode)<-[?:ServingCell]-
(endNodeRelatives)
```

```
RETURN startNode, relation,endNode, endNode.domain, endNode.type,
endNode.uniqueId, count(endNodeRelatives)]]>
  </query>
</cypherQuery>
```

7.3.8 ProblemDefault.computeGroupPriority(Event event)

A default implementation was introduced to utilize specific tags that can be set at the filter level:

- `Bundle.Priority` defines the priority for a family of Problems.
- `Problem.Priority` defines the priority of a Problem. The values for these tags should be numeric.

If one of those tags is present after filtering an alarm, the group priority is computed using the formula:

```
Bundle.Priority * $priority.factor + Problem.Priority
```

If none of the tags is present, the group priority is set to *null*.

The group priority is taken into account if the attribute `enablePrioritySort` is set to `true` in the `MainPolicy` of the `ProblemXmlConfig.xml` file. It means that all calls to `scenario.getGroups().getAllGroups() or to scenario.getGroups().getGroupsWhereXXX()` will return the groups sorted on priority.

By default, the attribute `enablePrioritySort` is considered as `false` if not defined and groups are not sorted.

Note

- Lower priority numbers come first. A *null* priority comes last.
- To change the value of the `$priority.factor` override the `getBundlePriorityFactor()` method of your problem.
- To change the name of the `Bundle.Priority` tag override the `getBundlePriorityTag()` method of your problem.
- To change the name of the `Problem.Priority` tag override the `getProblemPriorityTag()` method of your problem.

7.3.8.1 Example with alarms

The following alarms are received:

Table 27 - Trigger alarm group priority example

Trigger alarm	Bundle.priority	Problem.priority	Output group and priority
A1	10	1	G1, 10001
A2	-	2	G2, 2
A3	-	-	G3, NULL

If alarm S is subalarm of each of these trigger alarms, and if `MainPolicy.enablePrioritySort` is set to true, `getGroups().getGroupsWhereAlarmSetAs(S, Qualifier.SubAlarm)` returns [`G2, G1, G3`] in strict order .

7.3.8.2 Example with events

The following events are received:

Table 28 - Trigger event group priority example

Trigger event	Bundle.priority	Problem.priority	Output group and priority
A1	10	1	G1, 10001
A2	-	2	G2, 2
A3	-	-	G3, NULL

If alarm S is sub event of each of these trigger events, and if `MainPolicy.enablePrioritySort` is set to true, `getGroups().getGroupsWhereEventSetAs(S, Qualifier.SubEvent)` returns [`G2, G1, G3`] in strict order.

7.3.9 ProblemDefault.computeTimeWindow(Event event)

The default behavior of the default `computeTimeWindow(Alarm alarm)` method is to use the `Trigger.TimeLimit.Seconds` tag set at filters level and applied on the generic Event type.

If this tag is present after filtering an alarm, and its value is `T`, the returned timeWindow overrides the time window defined at the ProblemPolicy level and is computed as:

If T is 0: `TimeWindowMode.NONE`
If T is not 0: `TimeWindowMode.TRIGGER` and time window is [`abs(T) * 1000 , abs(T) * 1000`]

Note

It is possible to change the name of the `Trigger.TimeLimit.Seconds` tag by overriding the `getTriggerTimeLimitSecondsTag()` method of a problem.

7.4 Customizing default behavior

The default behavior of a Problem Detection Value Pack can be customized either by:

- Overriding java methods specially defined for this purpose
- Writing customization XML code

The list of java methods that can be overridden is presented in section 7.1 Default Behavior. Instructions on how to override these java methods is presented in section 7.4.2.

The way to modify the Problem Detection Value Pack default behavior by writing XML code is described in section 7.4.1 below.

7.4.1 XML customization

One aspect of the default behavior of Problem Detection Value Packs is to use the `originatingManagedEntity` property of the trigger alarm as the Problem Entity. An important purpose of Problem Alarm creation is to provide clear and concise information to the operator. For this reason it is useful to redefine the way Problem Detection computes the Problem Entity of a problem. This can be done two ways:

- Without customizing Java code - see the following example.
- Through Java code customization - see the next section.

The following example is an excerpt from the `ProblemXmlConfig.xml` file located in the `src/main/resources/valuepack/conf/` folder. It shows an example of overriding two methods: the `computeProblemEntity()` and `calculateProblemAlarmAdditionalText()`:

```
<problemPolicy name="XmlGeneric_Synch">
  <strings>
    <string key="computeProblemEntity">
      <value><![CDATA[
        if (alarm.getOriginatingManagedEntity().matches(
            "motorola_omcr_system .* managedelement .* bssfunction .*
btssitemgr .*")) {

            varStr1=alarm.getCustomFieldValue("userText");

            if (varStr1 != null) {
                varStr1 = varStr1.replaceAll(" ", "");
                varStr1 = varStr1.replaceAll(":", " bts ");
                varResult = "bsc " +varStr1;
            }
        }
        if (varResult==null) {
            varResult = alarm.getOriginatingManagedEntity();
        }
        ]]>
      </value>
    </string>

    <string key="calculateProblemAlarmAdditionalText">
       <value><![CDATA[site down (Synch_XML) - Generic XML]]></value></string>
  </strings>
</problemPolicy>
```

The following three methods are also available. Note that all other methods listed in section 7.1 are only overridable by writing Java code.

```
<string key="isMatchingTriggerAlarmCriteria">
   <value><![CDATA[true]]></value>
</string>

<string key="isMatchingProblemAlarmCriteria">
   <value><![CDATA[true]]></value>
</string>

<string key="isMatchingSubAlarmCriteria">
   <value><![CDATA[true]]></value>
</string>
```

Section 6.2.7 Defining the Filters, Table 12 – Tags for possible roles of an alarm describes how the role of an alarm is determined by the tag associated to it in the Filters XML file.

Neither setting the tag=SubAlarm or the method override takes precedence, both are taken in account.

For example, an alarm to be considered a sub-alarm by the Problem Detection Value Pack, it needs to be tagged as sub-alarm in the Filters XML file and the method `isMatchingSubAlarmCriteria()` must return true.

7.4.2 Java customization

To customize the default behavior of Problem Detection Value Packs an override must be provided on Java methods listed in section 7.1. Three customization levels exist:

- Per problem (described in this section)
- For a set of problems or all problems (see section 7.4.3 "My ProblemDefault")
- For non-problem specific matters (see section 7.4.5 "MyGeneralBehavior")

The methods that can be overridden to customize the problem-specific behavior of a Problem Detection Value Pack are all listed in the `ProblemInterface` Java interface.

The methods that can be overridden to customize the "non-problem specific" behavior of a Problem Detection Value Pack are all listed in the `GeneralBehaviorInterface` java interface.

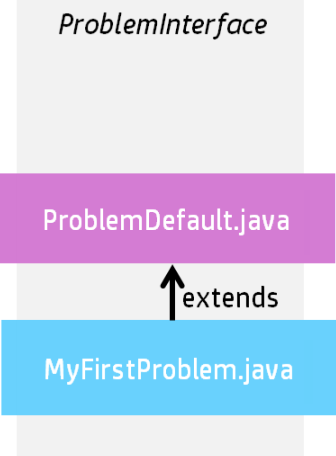


Figure 33 - One problem specific customization

`ProblemDefault.java` is the class implementing the methods of the ProblemInterface as seen in Figure 33. It defines the default behavior of Problem Detection Value Packs.

To override a method of the `ProblemInterface` one customization class must be created per problem, which extends `ProblemDefault`.

The following example is the `Problem_Skeleton.java` class created by the Eclipse plug-in. It is located in `src/main/java/[com.hp.uca.expert.vp.pd.problem]`

```
/**
 * This Problem is empty and ready to define methods to
```

```
 * customize this problem
 */
package com.hp.uca.expert.vp.pd.problem;

import org.apache.log4j.Logger;
import com.hp.uca.expert.vp.pd.core.ProblemDefault;
import com.hp.uca.expert.vp.pd.interfaces.ProblemInterface;

public final class Problem_Skeleton extends ProblemDefault implements
                                                    ProblemInterface {

                                                  public
Problem_Skeleton() {
                                                    super();

                                                  setLog(Logger.getLog
ger(Problem_Skeleton.class));
                                                  }

        }
```

The name of the class, in this example Problem_Skeleton, must be changed to the name of the problem for which we want to customize the behavior.

In other words, the name of the customization class for problem X must equal the name of problem X as defined in the filters file. For example, if the content of thevProblemDetection_filters.xml file is:

```
<topFilter name="Problem_LOS">
```

Then the extract of Problem_LOS.java must be:

```
public final class Problem_LOS extends ProblemDefault
implements ProblemInterface {
```

The following example is the same file renamed as MyFirstProblem.java, which overrides both the computeProblemEntity() and the calculateProblemAlarmAdditionalText() methods:

```
/**
 * This is my first Problem.
 * It customizes two methods:
 * - computeProblemEntity()
 * - calculateProblemAlarmAdditionalText()
 */
package com.hp.uca.expert.vp.pd.problem;

import org.slf4j.LoggerFactory;
import com.hp.uca.expert.vp.pd.core.ProblemDefault;
import com.hp.uca.expert.vp.pd.interfaces.ProblemInterface;

/**
 * @author Me
 *
 */
public final class MyFirstProblem extends ProblemDefault implements
                                                    ProblemInterface {

public MyFirstProblem () {
  super();
  setLog(LoggerFactory.getLogger( (MyFirstProblem.class));
                                                  }

        }
@Override
public List<String>  computeProblemEntity(Alarm a) {

if (getLog().isTraceEnabled()) {
LogHelper.enter(getLog(), "computeProblemEntity()",a.getIdentifier());
```

```
}
String problemEntity = null;
List<String> problemEntities = new ArrayList<String>();

if (a.getOriginatingManagedEntity().matches(
"motorola_omcr_system .* managedelement .* bssfunction .*
btssitemgr .*")) {

SupportedActions supportedActions = chooseSupportedActions(a, this);

String userText =
a.getCustomFieldValue(supportedActions.getAttributeUsedForKeyDuringReco
gnition());

if (userText != null) {
userText = userText.replaceAll(" ", "");
String[] table = userText.split(":");

if (table.length >= 2) {

problemEntity = String.format("bsc %s bts %s", table[0],
table[1]);


problemEntities.add(problemEntity);
                                                   }
  }
}

if (getLog().isTraceEnabled()) {
LogHelper.exit(getLog(), "computeProblemEntity()",
problemEntities.toString());
}
return problemEntities;
}


@Override
public String calculateProblemAlarmAdditionalText(Group group) {
return "site down (BitError)";

        }
```

The called overridable methods is decided depending on the life cycle of the alarm, the problem and its context.

The Problem Detection framework automatically invokes the methods listed in section 7.1, at specific times of the life cycle of every alarm.

For instance, when an alarm `alm1` is cleared, the Problem Detection framework invokes the method `whatToDoWhenXXXAlarmIsCleared(alm1…)`.

If `alm1` belongs to only one problem, `Problem A`, then the Problem Detection framework invokes the method `whatToDoWhenXXXAlarmIsCleared(alm1 …)` present in the customization class of `Problem A`. If the method `whatToDoWhenXXXAlarmIsCleared( )` is not overridden for `Problem A`, the default method is invoked.

If `alm1` also belongs to `Problem B`, the Problem Detection framework invokes in addition the method `whatToDoWhenXXXAlarmIsCleared(alm1 …)`, if present in the customization class of `Problem B`, or the default method otherwise.

Depending of the position of the alarm in its life cycle at a given time, the Problem Detection framework evaluates exactly which methods to invoke.

In the above example, assuming `alm1` belongs to both `Problem A` and `Problem B`, and that `alm1` at the moment it gets cleared, is:

- sub-alarm for `Problem A`.
- orphan alarm for `Problem B`.

Then the following methods are called:

- `whatToDoWhenSubAlarmIsCleared(alm1)` is called for `Problem A`.
- `whatToDoWhenOrphanAlarmIsCleared(alm1)` is called for `Problem B`.

Note

An Orphan Alarm is an alarm that does not belong to any group of the given problem.

A Candidate Alarm is an alarm that belongs to a group of the given problem, but the Problem Alarm of this group was not received yet.

A Sub-Alarm is an alarm that belongs to a group of the given problem, and the Problem Alarm of this group was received.

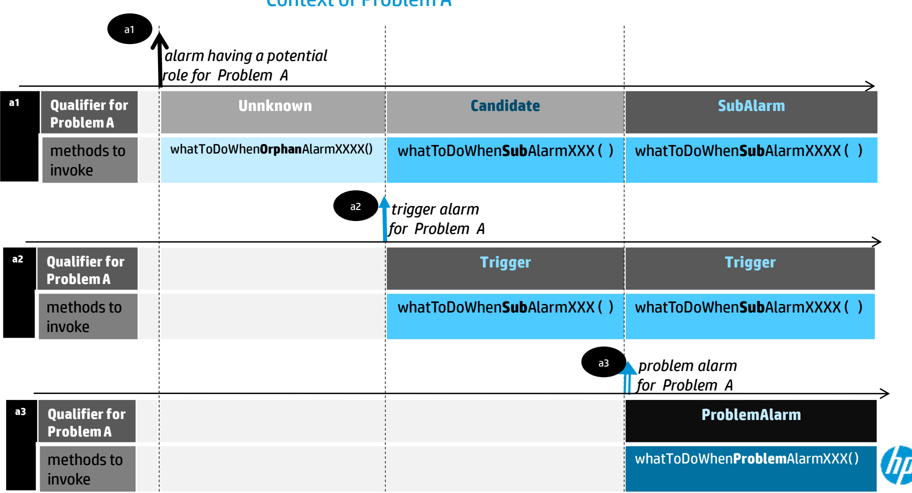
Figure 34 below shows a graphical representation of the methods invoked based on the life cycle of the alarm.

It contains three alarms:

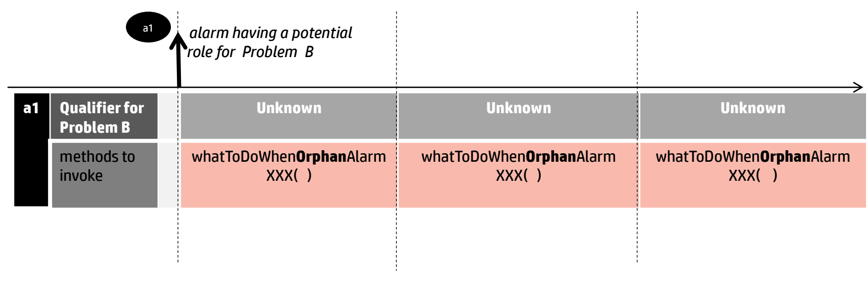
- `a1` belongs to `Problem A` and `Problem B`.
- `a2` is a Trigger Alarm and belongs to `Problem A` only.
- `a3` is a Problem Alarm and belongs to `Problem A` only.

Each alarm at a given time of its life has a qualifier for each of the problems it belongs to. It also has a consolidated view of its role across problems.

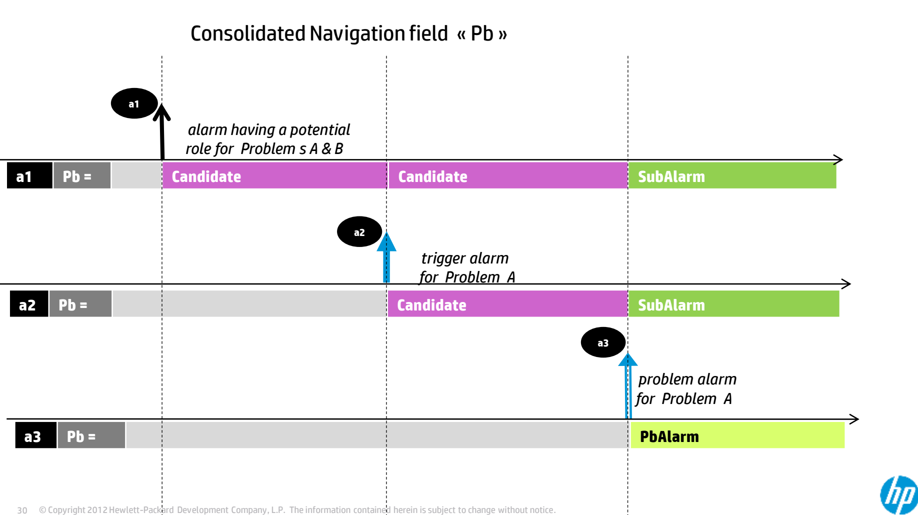
For example there is a time where `a1` is 'SubAlarm for `Problem A` and is an Orphan alarm for `Problem B`. At this time the consolidated role of `a1` across all problems is Sub-Alarm. This consolidated role is stored in the `Pb` field of the alarm.



Context of Problem B



29 © Copyright 2012 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice.



30 © Copyright 2012 Hewlett-Packard Development Company, L.P. The information contained herein is subject to change without notice.

Figure 34 - Consolidation of alarm's qualifiers

7.4.3 My ProblemDefault

The purpose of extending the `ProblemDefault` class is to modify the default behavior for all problems or for a set of problems.

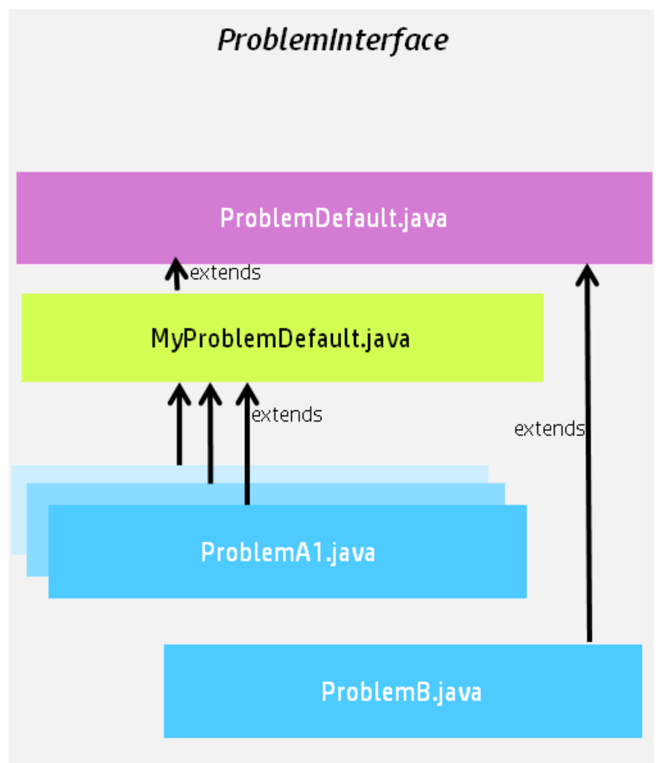


Figure 35 - MyProblemDefault: a customization for a group of problems

In the previous figure `MyProblemDefault.java` implements some or all methods of `ProblemInterface`. Each problem customization class that extends MyProblemDefault.java benefits from the implementation of those methods.

In the figure by default, `ProblemA1`, `ProblemA2` (hidden behind `ProblemA1`), and `ProblemA3` (hidden behind `ProblemA1`) use the methods implemented in `MyProblemDefault.java`. This happens only because the different propagation Java classes in `ProblemA1`, `ProblemA2`, and `ProblemA3` extend `MyProblemDefault` in their Java code.

Problem B uses methods implemented in `ProblemDefault.java`, unless these methods are overridden in `ProblemB.java`

For a comprehensive diagram showing the advanced possibilities of `Problemdefault.java` extensions see Annex C .

7.4.4 Problems initialization in version 3.2 and later

Initialization of problems defined inside the `<problemPolicy>` tag in the *ProblemXmlConfig.xml* file has changed in version 3.2.

In earlier versions a problem defined in the `ProblemXmlConfig.xml` file containing a subset of the problem policies as described in section 5.3.2.2 Problem Specific Policies. Other policies receive default values.

ProblemDefault (can be MyProblemDefault) configuration is used only to initialize a problem with its policy defined in a top filter in a `<topFilter>` tag of `ProblemDetection_filters.xml` file instead of the `ProblemXmlConfig.xml`.

Also, if no ProblemDefault policy tag is defined in the ProblemXmlConfig.xml file, then the default values are applied as specified in the ProblemDefault.java class.

Starting with version 3.2, all policies defined in the ProblemDefault problem policy (can be MyProblemDefault) are applied to all the other Problems, unless overwritten by their respective custom problem policy.

For policies listed in Table 19 – PD customized "per-problem" configuration: Strings, Longs and Booleans (which contain a sequence of String, Long and Boolean types) defined in the ProblemDefault are now valid for all other Problems and added to the ones defined in the sequence instead of overwriting them, even if they are defined in a custom problemPolicy. If specific behavior is preferred for a Problem, the recommended approach is to empty the ProblemDefault configuration and add definition in the custom problem policies. It is also recommended to identify what is common to all problems and define it in the common ProblemDefault configuration.

What has not changed compared to earlier releases is that ProblemDefault (can be MyProblemDefault) configuration is also used to completely initialize a problem the policy of which is not defined in *the* ProblemXmlConfig.xml, but as a top filter in a <topFilter> tag of ProblemDetection_filters.xml file. Also, if no ProblemDefault policy tag is defined in the ProblemXmlConfig.xml file, then the default values are applied as specified in the ProblemDefault.java class.

In the following configuration example in version 3.1 configuration the Strings for ProblemDefault, Problem_Synch, Problem_BitError and Problem_Power are identical but have to be defined for each. Also, the Booleans defined in ProblemDefault siteDown are valid also for Problem_Synch, Problem_BitError and Problem_Power, and each of these two problems have and extra Boolean to be defined (synchPb, bitErrorPb and powerPb).

It can be also identified that the delayForProblemAlarmCleareance setting is the same for all problems but has to be redefined each time, as well as the timeWindowBeforeTrigger and the timeWindowAfterTrigger. The delayForTroubleTicketCreation defined in ProblemDefault is the same as the one for Problem_Synch and Problem_BitError and the delayForProblemAlarmCleareance defined in ProblemDefault is the same as for Problem_ BitError.

```
…
<problemPolicy name="ProblemDefault">
 <problemAlarm>
  <delayForProblemAlarmCreation>1212</delayForProblemAlarmCreation>
  <delayForProblemAlarmClearance>0</delayForProblemAlarmClearance>
 </problemAlarm>
 <troubleTicket>
   <automaticTroubleTicketCreation>false
   </automaticTroubleTicketCreation>
   <propagateTroubleTicketToSubAlarms>false
   </propagateTroubleTicketToSubAlarms>
   <propagateTroubleTicketToProblemAlarm>false
   </propagateTroubleTicketToProblemAlarm>
   <delayForTroubleTicketCreation>60000</delayForTroubleTicketCreation>
  </troubleTicket>
 <groupTickFlagAware>false</groupTickFlagAware>
 <timeWindow>
  <timeWindowMode>None</timeWindowMode>
  <timeWindowBeforeTrigger>30000</timeWindowBeforeTrigger>
  <timeWindowAfterTrigger>30000</timeWindowAfterTrigger>
 </timeWindow>
 <booleans>
  <Boolean key="siteDown">
   <value>true</value>
```

```
  </Boolean>
 </booleans>
 <strings>
  <string key="ocName">
   <value>.uca_pbalarm</value>
   </string>
  </strings>
</problemPolicy>
…
<problemPolicy name="Problem_Synch">
 <problemAlarm>
  <delayForProblemAlarmCreation>5000</delayForProblemAlarmCreation>
  <delayForProblemAlarmClearance>10</delayForProblemAlarmClearance>
 </problemAlarm>
 <troubleTicket>
  <automaticTroubleTicketCreation>false
  </automaticTroubleTicketCreation>
  <propagateTroubleTicketToSubAlarms>false
  </propagateTroubleTicketToSubAlarms>
  <propagateTroubleTicketToProblemAlarm>false
  </propagateTroubleTicketToProblemAlarm>
  <delayForTroubleTicketCreation>60000</delayForTroubleTicketCreation>
 </troubleTicket>
 <groupTickFlagAware>false</groupTickFlagAware>
 <timeWindow>
  <timeWindowMode>Trigger</timeWindowMode>
  <timeWindowBeforeTrigger>30000</timeWindowBeforeTrigger>
  <timeWindowAfterTrigger>30000</timeWindowAfterTrigger>
 </timeWindow>
 <booleans>
   <Boolean key="siteDown">
    <value>true</value>
  </Boolean>
 <Boolean key="synchPb">
   <value>true</value>
   </Boolean>
 </booleans>
 <strings>
  <string key="ocName">
   <value>.uca_pbalarm</value>
  </string>
 </strings>
</problemPolicy>

<problemPolicy name="Problem_BitError">
 <problemAlarm>
   <delayForProblemAlarmCreation>1212</delayForProblemAlarmCreation>
   <delayForProblemAlarmClearance>0</delayForProblemAlarmClearance>
 </problemAlarm>
 <troubleTicket>
  <automaticTroubleTicketCreation>false
  </automaticTroubleTicketCreation>
  <propagateTroubleTicketToSubAlarms>false
  </propagateTroubleTicketToSubAlarms>
  <propagateTroubleTicketToProblemAlarm>false
  </propagateTroubleTicketToProblemAlarm>
  <delayForTroubleTicketCreation>60000</delayForTroubleTicketCreation>
</troubleTicket>
 <groupTickFlagAware>false</groupTickFlagAware>
 <timeWindow>
 <timeWindowMode>Trigger</timeWindowMode>
 <timeWindowBeforeTrigger>2500</timeWindowBeforeTrigger>
 <timeWindowAfterTrigger>5000</timeWindowAfterTrigger>
 </timeWindow>
 <booleans>
   <Boolean key="siteDown">
    <value>true</value>
  </Boolean>
  <Boolean key="bitErrorPb">
   <value>true</value>
  </Boolean>
 </booleans>
 <strings>
   <string key="ocName">
    <value>.uca_pbalarm</value>
   </string>
```

```
  </strings>
</problemPolicy>
<problemPolicy name="Problem_Power">
  <problemAlarm>
   <delayForProblemAlarmCreation>2700</delayForProblemAlarmCreation>
   <delayForProblemAlarmClearance>0</delayForProblemAlarmClearance>
  </problemAlarm>
  <troubleTicket>
   <automaticTroubleTicketCreation>false
   </automaticTroubleTicketCreation>
  <propagateTroubleTicketToSubAlarms>false
  </propagateTroubleTicketToSubAlarms>
  <propagateTroubleTicketToProblemAlarm>false
  </propagateTroubleTicketToProblemAlarm>
  <delayForTroubleTicketCreation>90000</delayForTroubleTicketCreation>
  </troubleTicket>
  <groupTickFlagAware>true</groupTickFlagAware>
  <timeWindow>
    <timeWindowMode>None</timeWindowMode>
  </timeWindow>
  <booleans>
   <Boolean key="powerPb">
    <value>true</value>
   </Boolean>
  </booleans>
 <strings>
   <string key="ocName">
    <value>.uca_pbalarm</value>
   </string>
  </strings>
</problemPolicy>
…
```

If the same configuration file is transformed to version 3.2 considering that all problems have their top filter defined in the `ProblemDetection_filters.xml` file (or if there are other problems, they are handled entirely by the `ProblemDefault` policy), the following configuration file is required:

```
…
<problemPolicy name="ProblemDefault">
 <problemAlarm>
  <delayForProblemAlarmCreation>1212</delayForProblemAlarmCreation>
  <delayForProblemAlarmClearance>0</delayForProblemAlarmClearance>
 </problemAlarm>
 <troubleTicket>
  <automaticTroubleTicketCreation>false
  </automaticTroubleTicketCreation>
  <propagateTroubleTicketToSubAlarms>false
  </propagateTroubleTicketToSubAlarms>
  <propagateTroubleTicketToProblemAlarm>false
  </propagateTroubleTicketToProblemAlarm>
  <delayForTroubleTicketCreation>60000</delayForTroubleTicketCreation>
 </troubleTicket>
 <groupTickFlagAware>false</groupTickFlagAware>
 <timeWindow>
  <timeWindowMode>None</timeWindowMode>
  <timeWindowBeforeTrigger>30000</timeWindowBeforeTrigger>
  <timeWindowAfterTrigger>30000</timeWindowAfterTrigger>
 </timeWindow>
 <booleans xmlns:p1="http://config.im.vp.expert.uca.hp.com/">
   <p1:boolean key="siteDown">
     <p1:value>true</p1:value>
   </p1:boolean>
 </booleans>
 <strings xmlns:p1="http://config.im.vp.expert.uca.hp.com/">
   <p1:string key="ocName">
   <p1:value>.uca_pbalarm</p1:value>
  </p1:string>
 </strings>
</problemPolicy>
…
```

```xml
<problemPolicy name="Problem_Synch">
 <problemAlarm>
 <delayForProblemAlarmCreation>5000</delayForProblemAlarmCreation>
 <delayForProblemAlarmClearance>10</delayForProblemAlarmClearance>
 </problemAlarm>
 <troubleTicket>
 <automaticTroubleTicketCreation>false
 </automaticTroubleTicketCreation>
 <propagateTroubleTicketToSubAlarms>false
 </propagateTroubleTicketToSubAlarms>
 <propagateTroubleTicketToProblemAlarm>false
 </propagateTroubleTicketToProblemAlarm>
 </troubleTicket>
 <groupTickFlagAware>false</groupTickFlagAware>
 <timeWindow>
  <timeWindowMode>Trigger</timeWindowMode>
  <timeWindowBeforeTrigger>30000</timeWindowBeforeTrigger>
  <timeWindowAfterTrigger>30000</timeWindowAfterTrigger>
 </timeWindow>
 <booleans xmlns:p1="http://config.im.vp.expert.uca.hp.com/">
  <p1:boolean key="synchPb">
   <p1:value>true</p1:value>
  </p1:boolean>
 </booleans>
</problemPolicy>
<problemPolicy name="Problem_BitError">
 <problemAlarm></problemAlarm>
 <troubleTicket>
 <automaticTroubleTicketCreation>false
 </automaticTroubleTicketCreation>
 <propagateTroubleTicketToSubAlarms>false
 </propagateTroubleTicketToSubAlarms>
  <propagateTroubleTicketToProblemAlarm>false
  </propagateTroubleTicketToProblemAlarm>
 </troubleTicket>
 <groupTickFlagAware>false</groupTickFlagAware>
 <timeWindow>
  <timeWindowMode>Trigger</timeWindowMode>
  <timeWindowBeforeTrigger>2500</timeWindowBeforeTrigger>
  <timeWindowAfterTrigger>5000</timeWindowAfterTrigger>
 </timeWindow>
 <booleans xmlns:p1="http://config.im.vp.expert.uca.hp.com/">
  <p1:boolean key="bitErrorPb">
    <p1:value>true</p1:value>
  </p1:boolean>
</problemPolicy>
<problemPolicy name="Problem_Power">
  <problemAlarm>
   <delayForProblemAlarmCreation>2700</delayForProblemAlarmCreation>
 </problemAlarm>
 <troubleTicket>
 <automaticTroubleTicketCreation>false
 </automaticTroubleTicketCreation>
  <propagateTroubleTicketToSubAlarms>false
  </propagateTroubleTicketToSubAlarms>
  <propagateTroubleTicketToProblemAlarm>false
  </propagateTroubleTicketToProblemAlarm>
  <delayForTroubleTicketCreation>90000</delayForTroubleTicketCreation>
 </troubleTicket>
 <groupTickFlagAware>true</groupTickFlagAware>
 <timeWindow>
   <timeWindowMode>None</timeWindowMode>
 </timeWindow>
 <booleans xmlns:p1="http://config.im.vp.expert.uca.hp.com/">
   <p1:boolean key="powerPb">
     <p1:value>true</p1:value>
   </p1:boolean>
  </booleans>
</problemPolicy>
…
```

7.4.5 MyGeneralBehavior

The `GeneralBehaviorInterface` Java interface contains methods that can be overridden to customize the “non-problem specific” behavior of a Problem Detection Value Pack.

Non-problem-specific behavior is a behavior that is not related to any problem in particular. For example, the actions done when a Problem Detection Value Pack is initialized is a “non-problem-specific” behavior.

The process to customize such behavior is as follows:

- Create a `MyGeneralBehavior.java` (name can be different) Java class in the following directory: `src/main/java/[com.hp.uca.expert.vp.pd.core]`.
- Ensure that the value of the property `generalBehaviorClassName` in the `src/main/resources/valuepack/conf/context.xml` file matches MyGeneralBehavior , as shown in *Figure 36 – PD MyGeneralBehavior name* matching
- Override the methods of the `GeneralBehaviorInterface` for which the behavior has to be customized.

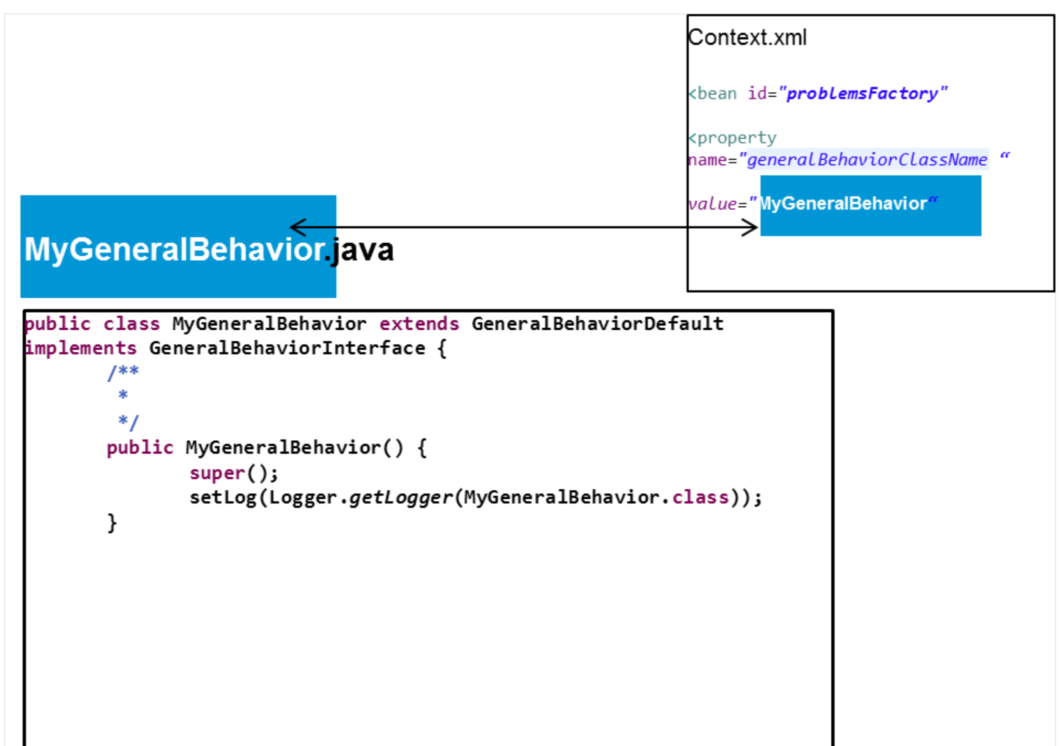


Figure 36 – PD MyGeneralBehavior name matching

The following example `MyGeneralBehavior.java` class overrides the `whatToDoWhenNewAlarmIsJustInserted()` method of the **GeneralBehaviorInterface** interface:

```
public class MyGeneralBehavior extends GeneralBehaviorDefault implements
                                             GeneralBehaviorInterface {

/**
 *
 */
```

```
public MyGeneralBehavior() {
 super();
 setLog(LoggerFactory.getLogger(MyGeneralBehavior.class));
}
/*
 * (non-Javadoc)
 *
 * @see
 * com.hp.uca.expert.vp.pd.core.CustomDefault#whatToDoWhenNewAlarmIsJustInserted
 * (com.hp.uca.expert.alarm.Alarm)
 */
@Override
public void whatToDoWhenNewAlarmIsJustInserted(Alarm alarm) {
 if (getLog().isTraceEnabled()) {
  LogHelper.enter(getLog(), "whatToDoWhenNewAlarmIsJustInserted()",
  alarm.getIdentifier());
}
 if (getLog().isDebugEnabled()) {
  getLog().debug(
   "I am the method whatToDoWhenNewAlarmIsJustInserted() of ProblemDefault : "
 + this.getClass().getSimpleName());
  getLog().debug(
  "whatToDoWhenNewAlarmIsJustInserted(): new alarm inserted : "
  + alarm.getIdentifier());
 }
 Flag flag = new Flag("JustInserted: " + alarm.getIdentifier(),
  "Flag checking whatToDoWhenNewAlarmIsJustInserted()", true);
  getScenario().getSession().insert(flag);

  if (getLog().isTraceEnabled()) {
  LogHelper.exit(getLog(), "whatToDoWhenNewAlarmIsJustInserted()");
  }
  }

        }
```

7.4.6 Enrichment

Three methods exist to enrich alarms in Problem Detection.

Through the HP UCA EBC life cycle, synchronous enrichment is possible. For more details, see [R1] *Unified Correlation Analyzer for Event Based Correlation Reference Guide*.

A "*One time*" and "*independent of all problems*" synchronous enrichment is possible by overriding the method **whatToDoWhenNewAlarmIsJustInserted()** Independent of all problems means that the enrichment applies to all alarms managed by the value pack regardless of the problem(s) they correspond to.

A "*per problem*" enrichment is possible by overriding the method **isInformationNeededAvailable( )** in the problem's customization class. This enrichment can be performed in synchronous or asynchronous mode.

The enrichment is synchronous when the Problem Detection value pack waits for the enrichment of the alarm to be completed before to proceed with the alarm processing. This enrichment can be synchronous, if the method **isInformationNeededAvailable()** is overridden with synchronous code.

The enrichment is asynchronous when the Problem Detection value pack does not wait for the enrichment of the alarm to be completed. The execution continues and the value pack is notified later through a callback that the enrichment has been completed. This enrichment can be asynchronous, if the method **isInformationNeededAvailable()** is overridden with asynchronous code.

Example: Problem-independent enrichment

The following example illustrates problem independent one-time enrichment

It shows an override to the whatToDoWhenNewAlarmIsJustInserted () method and new custom fields added to all incoming alarms.

```
public class MyGeneralBehavior extends
                                                      GeneralBehaviorDefault
implements GeneralBehaviorInterface {


@Override
public void whatToDoWhenNewAlarmIsJustInserted(Alarm alarm)
                                                      throws Exception {

SupportedActions supportedActions = PD_Service_Action
.retrieveSupportedActions(getScenario(), alarm);

if (alarm.getCustomFieldValue("userText") == null) {
CustomField cf = new CustomField();
cf.setName("userText");
cf.setValue("myotherproblemidentifier site#sophia");
alarm.getCustomFields().getCustomField().add(cf);
 }
}
}
```

Example Synchronous enrichment per problem

The following example shows the method isInformationNeededAvailable() being overridden. The method checks if enough information is present in the alarm. In particular it checks if the content of the field originatingManagedEntity is having the right structure. If not, the method decides to enrich the alarm by reading an XML file.

```
@Override
public Boolean isInformationNeededAvailable(Alarm alarm) throws Exception {

  Boolean informationAvailable = false;
  String site = null;
  if (!(alarm.getOriginatingManagedEntity().matches(
      "motorola_omcr_system .* managedelement .* bssfunction .* btssitemgr .*")) {

    EnrichmentProperties enrichmentProperties = (EnrichmentProperties)
      PD_Service_Util.retrieveBeanFromContextXml(getScenario(),
ENRICHMENT_BEAN_NAME);
    if (enrichmentProperties != null) {
      synchronized (enrichmentProperties .getHashManagedObjectToSite()) {
        site = enrichmentProperties.getHashManagedObjectToSite().get(

                                                  alarm.getOriginatingManagedEntity
());
      }
    }
  }

  if (site != null) {
    informationAvailable = true;
    alarm.getVar().put(SITE_KEYWORD, site);
  } else {
    getLog().warn(String.format("Unable to retrieve enrichment for alarm
[%s]",alarm.getIdentifier()));
  }

return informationAvailable;
}
```

The example above is extracted from Problem_Power.java. This file is available in the HP UCA EBC Development Kit Problem Detection Extension in the com.hp.uca.expert.vp.pd.problem package.

Example Asynchronous enrichment per problem

The example below shows the method isInformationNeededAvailable() being overridden. The method controls if enough information is available, by checking whether field "grid" is present in the alarm. If not, the method decides to enrich the alarm by launching an asynchronous action.

```
public Boolean isInformationNeededAvailable(Alarm alarm) throws Exception {

 Boolean retValue = true;
 String gridField = alarm.getCustomFieldValue("grid");
 if (gridField == null) {
  retValue = false;
  try {
     SupportedActions supportedActions = PD_Service_Action
                                                    .retrieveSupportedActions(alarm,
this);

     Action action = new Action(supportedActions.getActionReference());

     /*
      * Really fill the command for a real Action
      */
     action.addCommand("<To be customized with the real command to execute to find the
information>", "<To be customized with the entity on which to run the command>");

     getScenario().addAction(action);

     action.setCallback(buildenrichmentCallback(getScenario(),
     alarm, action, getLog()));
     action.executeAsync(null);
     getScenario().getSession().update(action);
  }
 }
}
```

Code example for an enrichment callback:

```
public static Callback buildEnrichmentCallback(Scenario scenario,
                                                  Alarm alarm, Action action, Logger
log)
                                                    throws NoSuchMethodException {

Class<?> partypes[] = new Class[NB_CALLBACK_ARGUMENTS];
partypes[ARGUMENT_1] = Scenario.class;
partypes[ARGUMENT_2] = Alarm.class;
partypes[ARGUMENT_3] = Action.class;
partypes[ARGUMENT_4] = Logger.class;

Object arglist[] = new Object[NB_CALLBACK_ARGUMENTS];
arglist[ARGUMENT_1] = scenario;
arglist[ARGUMENT_2] = alarm;
arglist[ARGUMENT_3] = action;
arglist[ARGUMENT_4] = log;
Method method = Problem_Synch_MissingInfoAlarm.class.getMethod(
                                                  "enrichmentCallback", partypes);

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

return callback;
}
```

```
public static void enrichmentCallback(Scenario scenario, Alarm alarm,
                                      Action action, Logger log)
{

// To be customized : BEGIN

  if (action.isTestOnly()) {
    if (log.isInfoEnabled()) {
        log.info("Enrichment Action Response received, updating Alarm with result of
        the Action");
    }

    alarm.setCustomFieldValue("grid", "disabled");
  }

// To be customized : END

  PD_Service_Enrichment.setAlarmIsNoMoreMissingInformation(alarm,
  Problem_Synch_MissingInfoAlarm.class.getSimpleName());

  PD_Service_Enrichment.requestAlarmComputation(scenario, alarm);

}
```

7.4.7 MyGeneralBehavior

As explained for problems general behavior in 7.4.5, the same reasoning applies for propagations. The methods that can be overridden to customize the "non-propagation specific" behavior of a Topology State Propagator Value Pack are all listed in the **GeneralBehaviorInterface** Java interface.

A "non-propagation-specific" behavior is a behavior that is not related to any propagation in particular.

For example, the behavior of the initialization of a Topology State Propagator Value Pack is a "non-propagation-specific" behavior.

The way to customize a "non-propagation-specific" behavior is presented in the following steps:

- Create a **MyGeneralBehavior.java** (name can be different) Java class in the following directory: `src/main/java/[com.hp.uca.expert.vp.tp.core]`.
- Ensure that the value of the property `generalBehaviorClassName` in the file `context.xml` in `src/main/resources/valuepack/conf/` folder matches

MyGeneralBehavior , as shown in Figure 37 – TSP MyGeneralBehavior name matching

- Override the methods of the **GeneralBehaviorInterface** for which the behavior has to be customized.

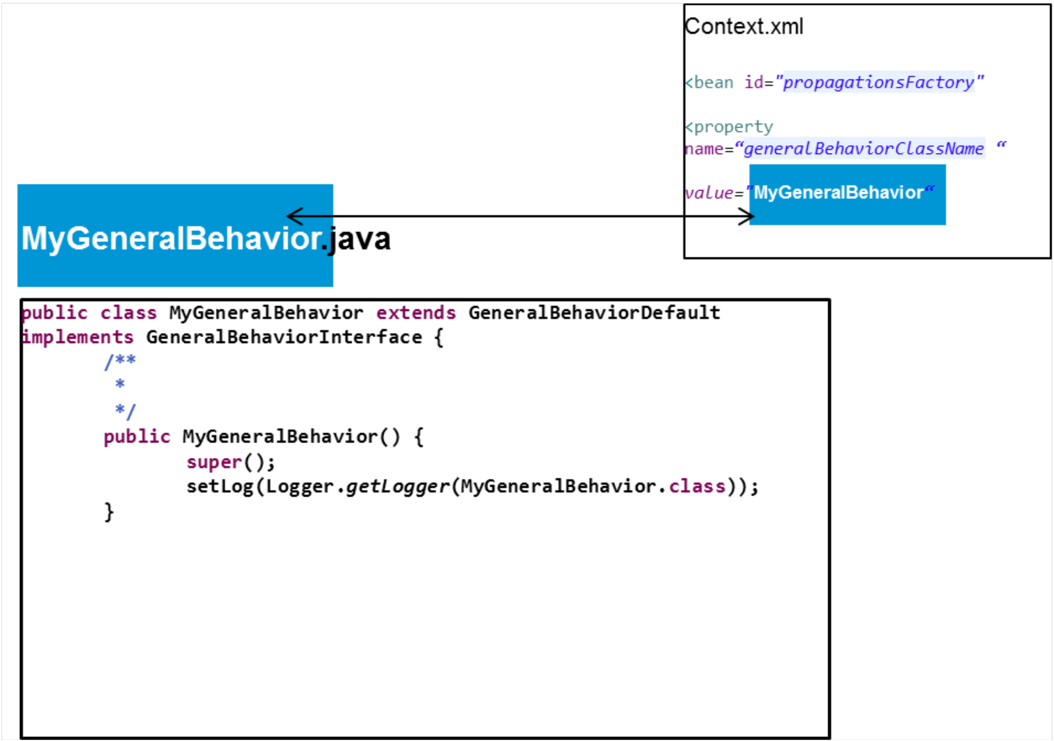


Figure 37 – TSP MyGeneralBehavior name matching

Below is an example of a MyGeneralBehavior.java class that overrides one method of the interface **GeneralBehaviorInterface**: computeSourceUniqueId().

```
public class MyGeneralBehavior extends GeneralBehaviorDefault implements
GeneralBehaviorInterface {

/**
 * Instantiates a new my general behavior.
 */
public MyGeneralBehavior() {
 super(LoggerFactory.getLogger(MyGeneralBehavior.class));
}

/*
 * (non-Javadoc)
 *
 * @see
 * com.hp.uca.expert.vp.tp.core.GeneralBehaviorDefault#computeSourceUniqueId
 * (com.hp.uca.expert.event.Event)
 */
@Override
public String computeSourceUniqueId(Event event) throws Exception {
 String ret = super.computeSourceUniqueId(event);
 return ret == null ? ret : ret.toUpperCase();
}}
```

Chapter 8

Advanced features of the Topology State Propagator

After configuration (see section 5.4), a TSP Value Pack has a default behavior.

This default behavior is a rich behavior that does not have to be altered or extended.

For the use cases where modification or extension is required, TSP offers the flexibility to change the default behavior.

The default behavior is described in section 8.1.

For information on how to change the default behavior see section 8.2

8.1 The default behavior

The Topology State Propagator framework is a set of Java libraries. To change the default behavior of TSP Value Packs, the classes of the Java libraries have to be extended and methods have to be overridden.

Each of the following methods has a default behavior that can be changed by overriding the method.

On the default behavior of these methods, consult the Javadoc. The implementation code of these methods is included in the example value pack delivered as part of the TSP Dev Kit. The code of each method is executed for every propagation and can be overridden by the value pack developer.

8.1.1 Example

Figure 26 shows how the workflow of the different methods is triggered by an Operator State Update alarm. The alarm termination is managed for the following context: alarm 1 is root cause alarm in propagation group1 of Propagation1 and in propagation group2 of Propagation2 and has no role in any of Propagation3's groups.

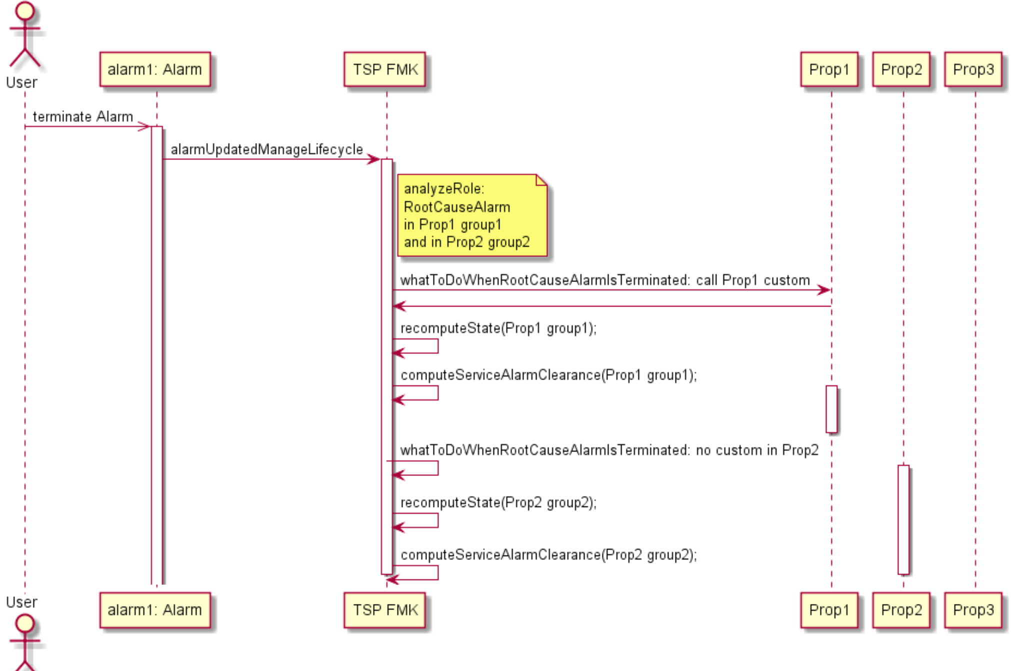


Figure 38 - Alarm termination sequence diagram example

In the topology shown in Figure 39 - Topology of the example, only Prop2 and Prop3 are connected and Prop3 has a finer grain propagation then Prop2.

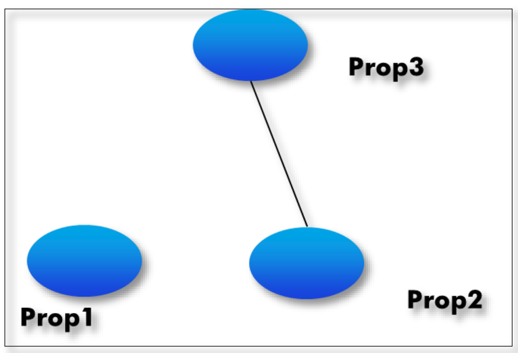


Figure 39 - Topology of the example

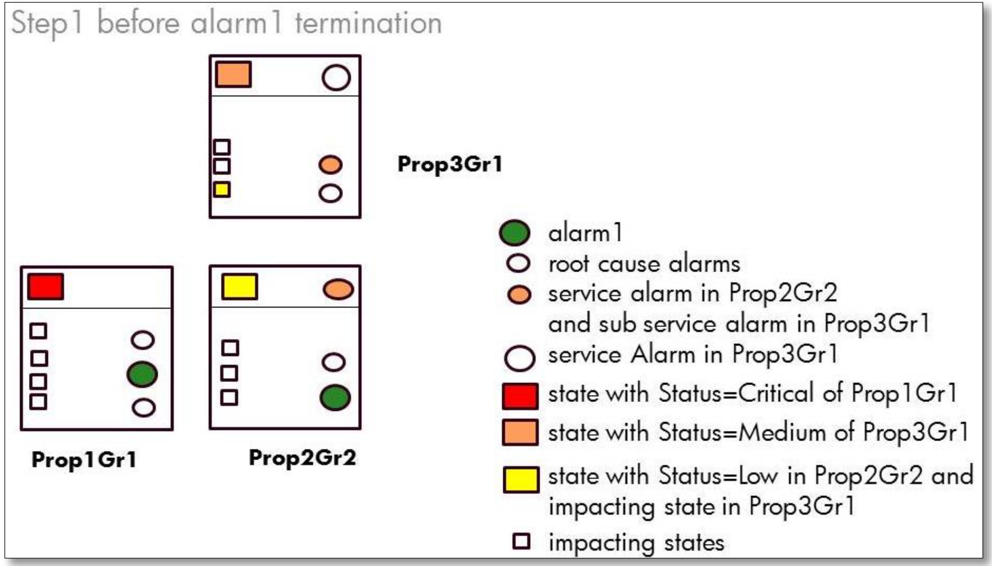


Figure 40 - TSP: Alarm termination example: TSP group updates Step1

The alarm1 termination is received and a number of methods are called by the TSP framework. As alarm1 has a role in the computation of all states in all groups, it will result in the groups' status change. Alarm1 has no impact on service Alarms computation. The propagation groups are as shown in Figure 29.

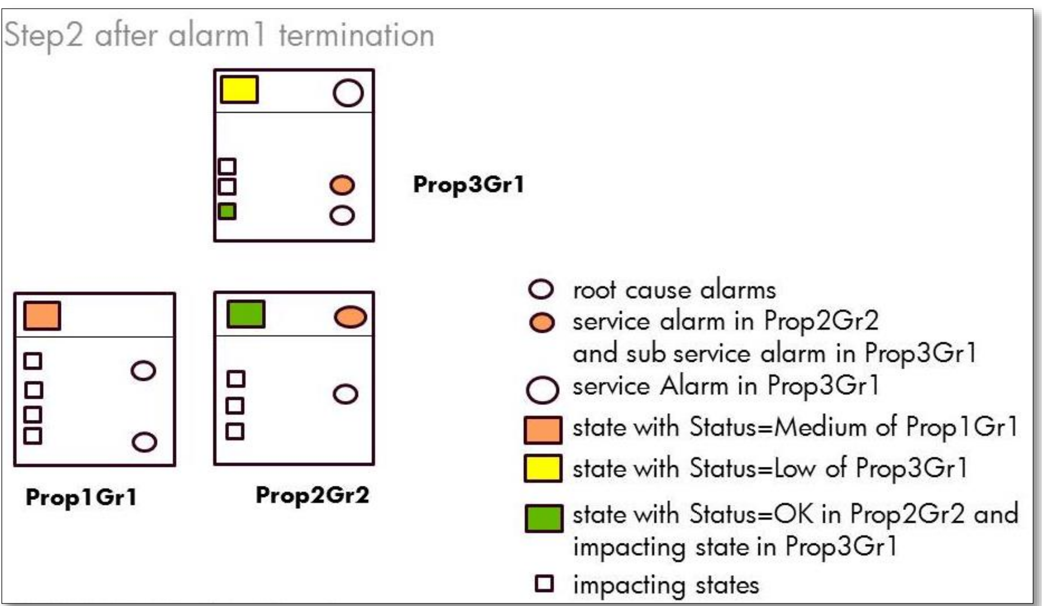
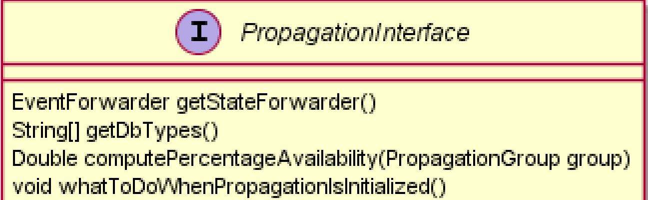
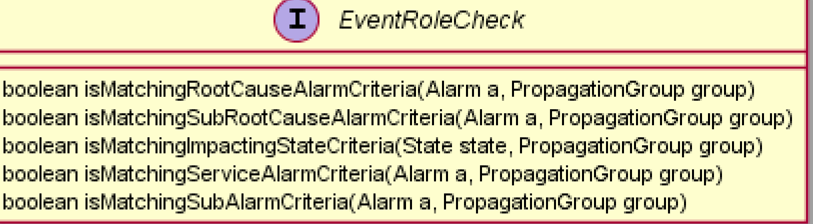


Figure 41 - TSP: Alarm termination example: TSP group updates Step2

8.1.2 Propagation Interface

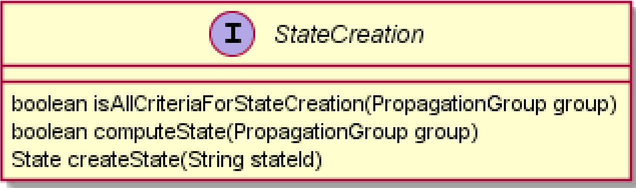


8.1.3 Event Role Check



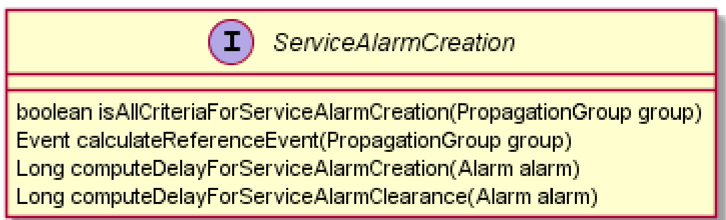
8.1.4 State Creation

Method used to check whether all criteria are met to create the State:



8.1.5 Service Alarm Creation and Clearance

Method used to check if all criteria are met to create the Service alarm:



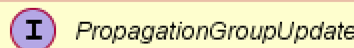
8.1.6 Common Entity Check

Methods used to calculate Information for optimizations

```
String computePropagationKey(Event event, String propagationEntity)
Long computeGroupPriority(Event event)
```

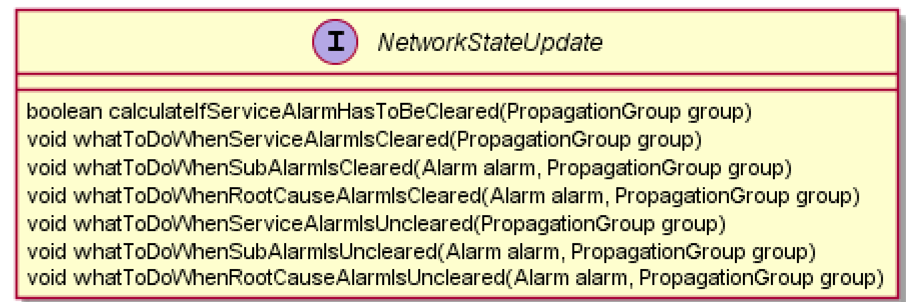
8.1.7 PropagationGroup update

Methods used to manage the propagation group life cycle, and its associated alarms and states.




```
void whatToDoWhenRootCauseAlarmIsAttachedToGroup(Alarm alarm, PropagationGroup group)
void whatToDoWhenImpactingStateIsAttachedToGroup(State state, PropagationGroup group)
void whatToDoWhenStateIsAttachedToGroup(State state, PropagationGroup group)
void whatToDoWhenStateIsUpdated(State state, PropagationGroup group)
boolean whatToDoPeriodicallyForAGroup(PropagationGroup group)
void whatToDoWhenServiceAlarmIsAttachedToGroup(PropagationGroup group)
void whatToDoWhenSubAlarmIsAttachedToGroup(Alarm alarm, PropagationGroup group)
```

8.1.8 Network State Update



Alarm Network State Changes

WM
Topology State Propagator Framework
PropagationDefault
alarm state has changed
alt [networkState = CLEARED]
loop [for all groups where alarm is ServiceAlarm]
whatToDoWhenServiceAlarmIsCleared(group)
loop [for all groups where alarm is SubAlarm or SubServiceAlarm]
whatToDoWhenSubAlarmIsCleared(alarm, group)
possibly compute ServiceAlarm clearance for the group
loop [for all groups where alarm is RootCauseAlarm]
whatToDoWhenRootCauseAlarmIsCleared(alarm,group)
possibly compute ServiceAlarm clearance for the group
[networkState = NOT_CLEARED]
loop [for all groups where alarm is ServiceAlarm]
whatToDoWhenServiceAlarmIsUncleared(group)
loop [for all groups where alarm is SubAlarm or SubServiceAlarm]
whatToDoWhenSubAlarmIsUncleared(alarm, group)
possibly compute ServiceAlarm clearance for the group
loop [for all groups where alarm is RootCauseAlarm]
whatToDoWhenRootCauseAlarmIsUncleared(alarm,group)
possibly compute ServiceAlarm clearance for the group
WM
Topology State Propagator Framework
PropagationDefault

Figure 42 - Alarm networks state change flow

8.1.9 Operator State Update

Methods used to manage the life cycle of a

- ServiceAlarm
- SubAlarm
- RootCauseAlarm

And its consequences

OperatorStateUpdate

void whatToDoWhenServiceAlarmIsTerminated(PropagationGroup group)
void whatToDoWhenServiceAlarmIsAcknowledged(PropagationGroup group)
void whatToDoWhenServiceAlarmIsUnacknowledged(PropagationGroup group)
void whatToDoWhenSubAlarmIsTerminated(Alarm alarm, PropagationGroup group)
void whatToDoWhenSubAlarmIsAcknowledged(Alarm alarm, PropagationGroup group)
void whatToDoWhenSubAlarmIsUnacknowledged(Alarm alarm, PropagationGroup group)
void whatToDoWhenRootCauseAlarmIsTerminated(Alarm alarm, PropagationGroup group)
void whatToDoWhenRootCauseAlarmIsAcknowledged(Alarm alarm, PropagationGroup group)
void whatToDoWhenRootCauseAlarmIsUnacknowledged(Alarm alarm, PropagationGroup group)

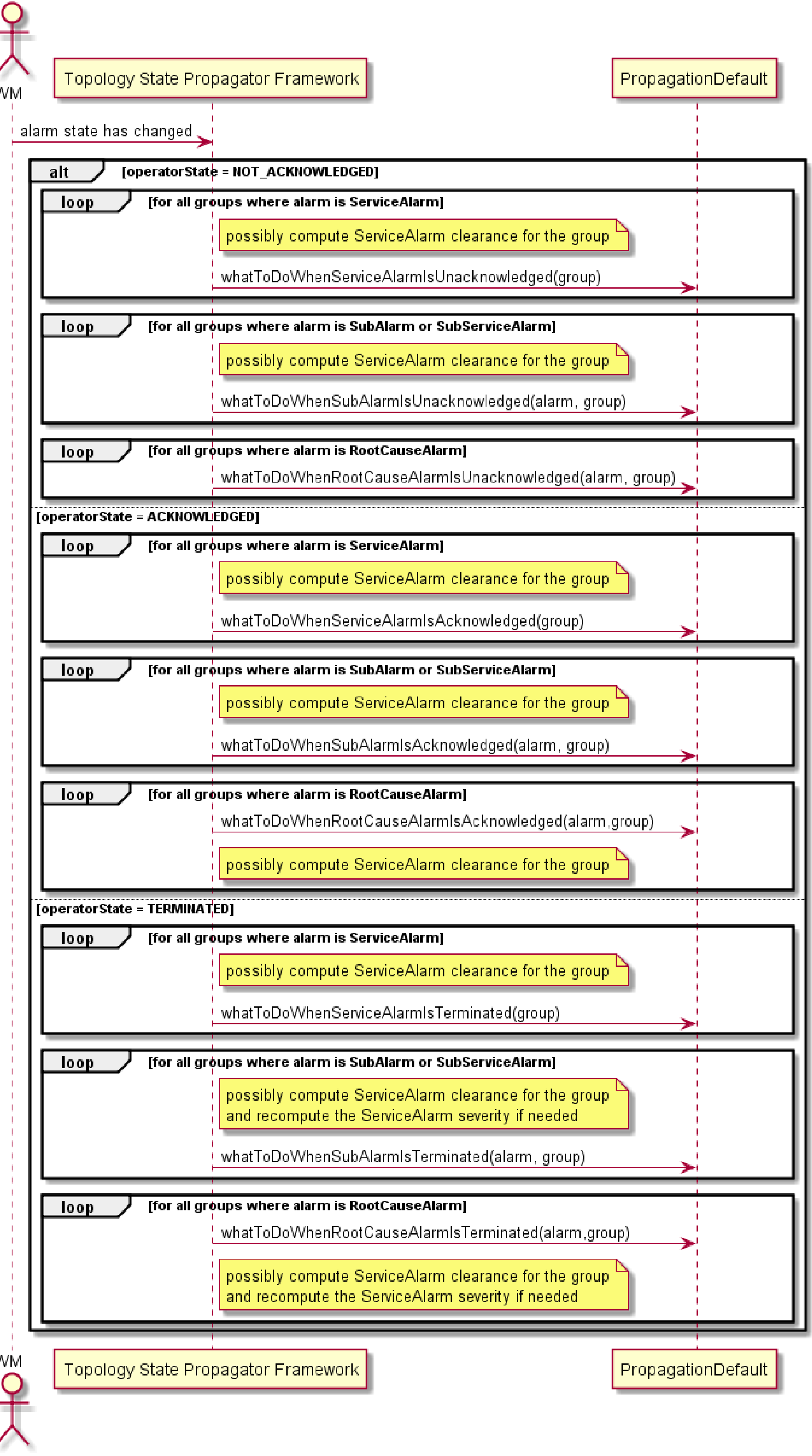


Figure 43 - Alarm operator state change flow

8.1.10 Alarm Attribute Update

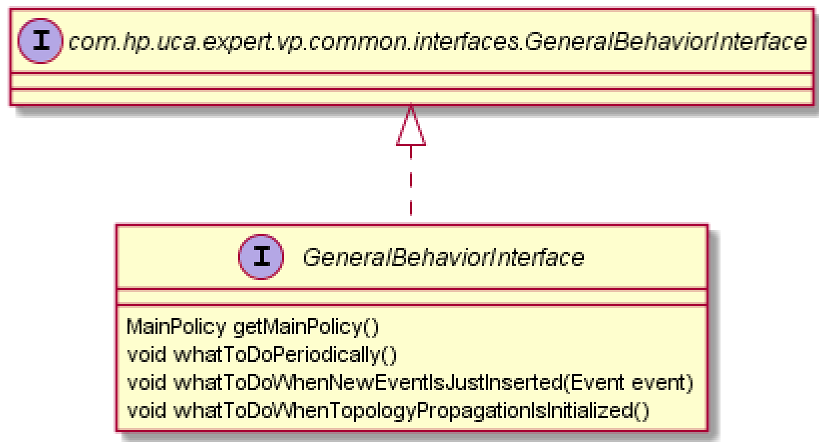
Methods used to manage the Severity or an Attribute Update of

- ServiceAlarm
- RootCause Alarm
- SubAlarm

, and its consequences

```
void whatToDoWhenRootCauseAlarmSeverityHasChanged(PropagationGroup group, Alarm alarm)
void whatToDoWhenServiceAlarmSeverityHasChanged(PropagationGroup group, Alarm alarm)
void whatToDoWhenSubAlarmSeverityHasChanged(PropagationGroup group, Alarm alarm)
void whatToDoWhenServiceAlarmAttributeHasChanged(PropagationGroup group, AttributeChange attributeChange)
void whatToDoWhenSubAlarmAttributeHasChanged(Alarm alarm, PropagationGroup group, AttributeChange attributeChange)
void whatToDoWhenRootCauseAlarmAttributeHasChanged(Alarm alarm, PropagationGroup group, AttributeChange attributeChange)
```

8.1.11 Periodic Check and General Behavior



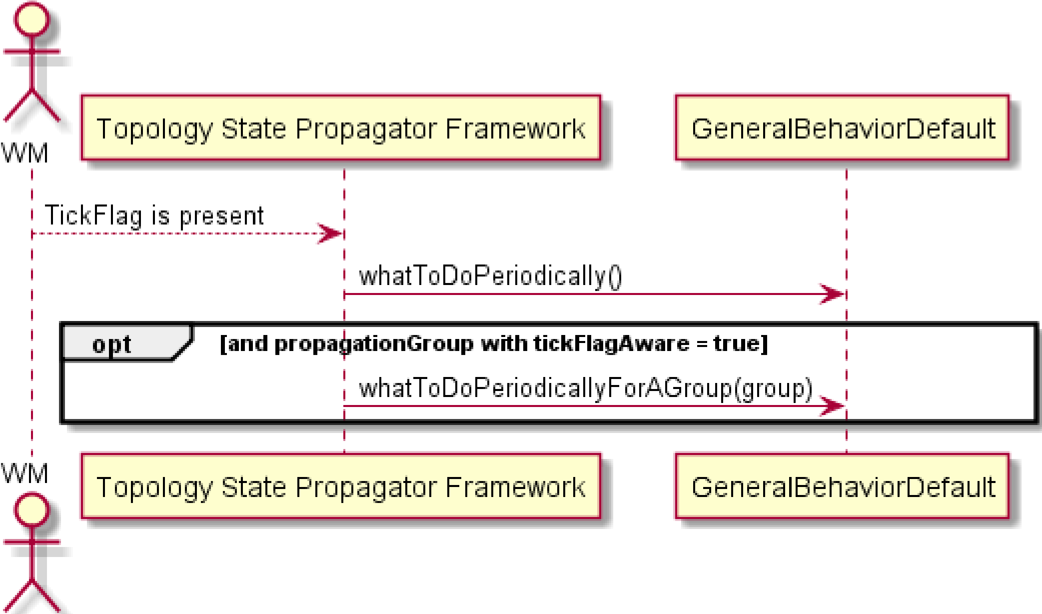
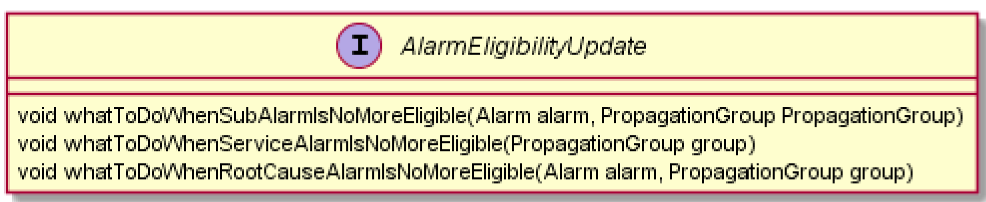


Figure 44 - Periodic check and general behavior

8.1.12 Alarm Eligibility Update

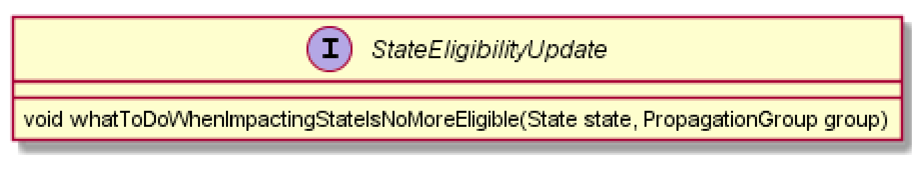


Alarm Eligibility Update

WM
Topology State Propagator Framework
PropagationDefault
alarm has been tagged to be retracted
loop [for all propagation groups where alarm is ServiceAlarm]
compute ServiceAlarm clearance for the group
whatToDoWhenServiceAlarmIsNoMoreEligible(group)
loop [for all propagation groups where alarm is SubAlarm or SubServiceAlarm]
whatToDoWhenSubAlarmIsNoMoreEligible(alarm, group)
compute ServiceAlarm clearance for the group
recompute the ServiceAlarm severity if needed
opt [ServiceAlarm of the group is cleared]
whatToDoWhenServiceAlarmIsNoMoreEligible(group)
loop [for all propagation groups where alarm is RootCauseAlarm]
whatToDoWhenRootCauseAlarmIsNoMoreEligible(alarm,group)
compute ServiceAlarm clearance for the group
recompute the ServiceAlarm severity if needed
opt [ServiceAlarm of the group is cleared]
whatToDoWhenServiceAlarmIsNoMoreEligible(group)
loop [for all propagation groups where alarm was present]
opt [if group has become useless]
retract the group
retract the alarm
WM
Topology State Propagator Framework
PropagationDefault

Figure 45 - Alarm eligibility update

8.1.13 State Eligibility Update



State Eligibility Update

WM

Topology State Propagator Framework

PropagationDefault

state has been tagged to be retracted

loop [for all propagation groups where state is present as impacting state]

whatToDoWhenImpactingStateIsNoMoreEligible(state,group)

compute ServiceAlarm clearance for the group
recompute the ServiceAlarm severity if needed

loop [for all propagation groups where state was present]

opt [if group has become useless]

retract the group

retract the state

WM

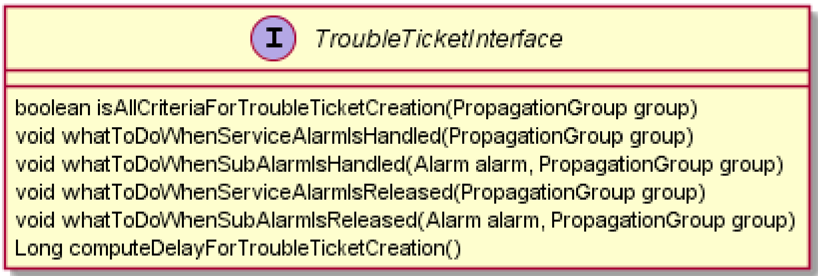
Topology State Propagator Framework

PropagationDefault

Figure 46 - State eligibility update

8.1.14 TroubleTicket update

Methods used to manage the Trouble Ticket life cycle when related to a ServiceAlarm or a SubAlarm, and its consequences.



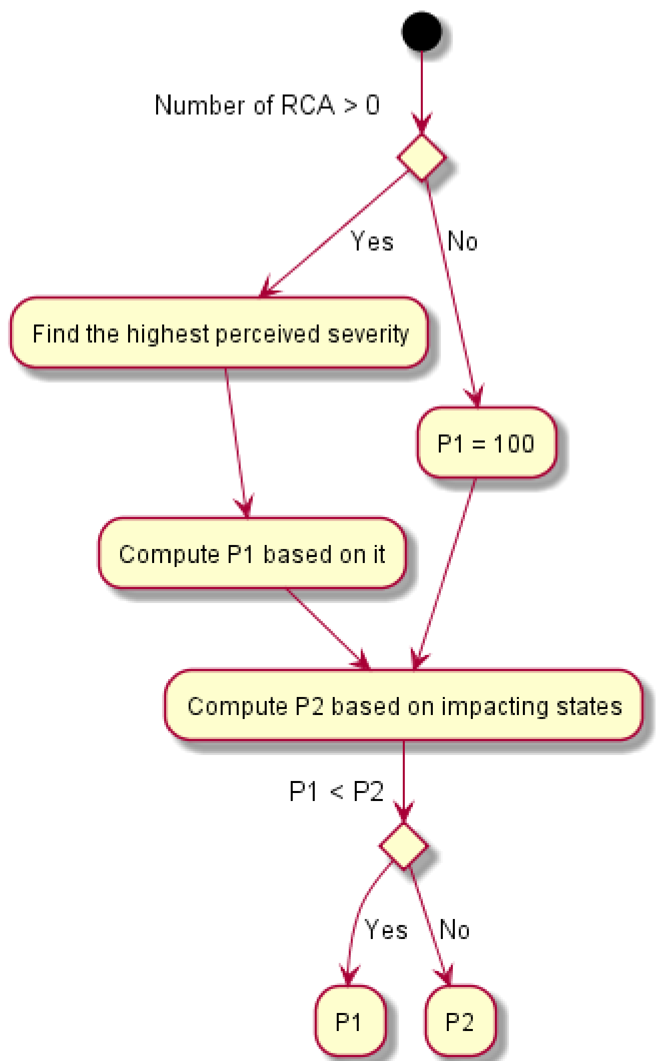
8.2 Computing State

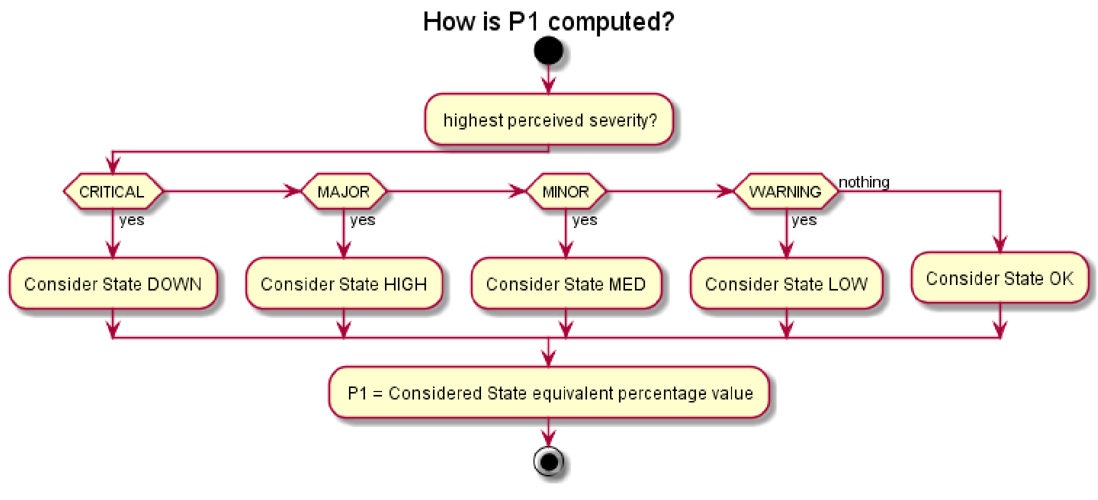
The default state computation is based on service provided by `TP_Service_StateCalculation.`*`computePercentageAvailability ().`*

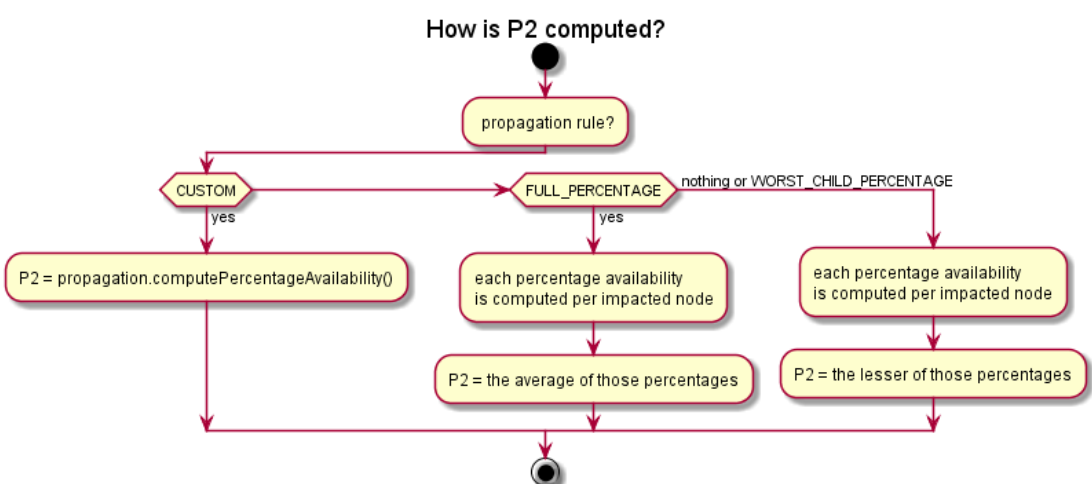
Depending on the percentage of availability of the impacted node, a state is associated to that value, given that the limits are:

- OK = 100% available
- DOWN = 0% availabale

A percentage in between those limits will be associated to a status like LOW, MED, HIGH, and CRITICAL or a customized status.







8.3 Customizing the default behavior

A TSP VP example is provided with the IM SDK, described in Annex E.

8.3.1 Java customization

The default behavior of Topology State Propagator Value Packs can be changed by overriding some of the Java methods listed in section 8.1. There are three levels of customization:

- Per propagation (described in section 8.2.1)
- For a set of or for all propagations (described in section 8.3.2)
- For non-propagation specific matters (described in section 8.3.3)

The methods that can be overridden to customize the propagation specific behavior of a Topology State Propagator Value Pack are listed in the `PropagationInterface` Java interface.

The methods that can be overridden to customize the non-propagation specific behavior of a Topology State Propagator Value Pack are listed in the `GeneralBehaviorInterface` Java interface.

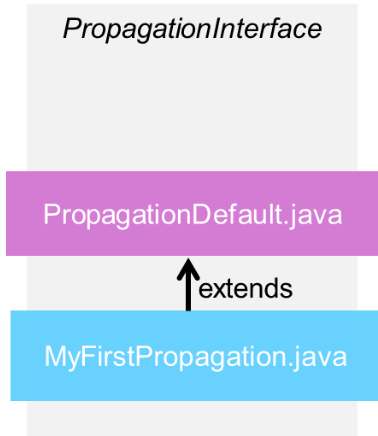


Figure 47 – One propagation specific customization

PropagationDefault.java is the class that implements the methods of the ProblemInterface. It defines the default behavior of Topology State Propagator Value Packs.

To override a method of the PropagationInterface, one customization class per propagation must be created. The customization class extends PropagationDefault**Error! Reference source not found.**.

See below the "`MyPropagation.java`" class created by the Eclipse plug-in. It is located in *src/main/java/[com.hp.uca.expert.vp.tp.core].*

```
/**
 * This Propagation is empty and ready to define methods to customize the
PropagationDefault
 */
package com.hp.uca.expert.vp.tp.core;

import org.slf4j.LoggerFactory;

import com.hp.uca.expert.vp.tp.core.PropagationDefault;
import com.hp.uca.expert.vp.tp.interfaces.PropagationInterface;

public class MyPropagation extends PropagationDefault implements
PropagationInterface {

 public MyPropagation() {
   super(LoggerFactory.getLogger(MyPropagation.class));
 }

}
```

Note that the name of the class (MyPropagation in the above example) must be changed to the name of the propagation for which the behavior is to be customized.

The following equation must be true:

Name of the customization class for propagation X = name of propagation X as defined in TopologyPropagation_filters.xml file.

For example, if the extract of *`TopologyPropagation_filters.xml`* is:

```
<topFilter name="Propagation_PhoneService">
```

Then the class *Propagation_PhoneService.java* must be declared in the following way:

```
public final class Propagation_PhoneService extends
PropagationDefault implements PropagationInterface {
```

Below, the same file is renamed as MyFirstPropagation.java, and overrides both the *calculateAlarmOperatorNote()* and *calculateAlarmOtherAttribute()* methods.

```
/**
 * The Class MyFirstPropagation extends PropagationDefault and overrides <li>
 * {@link #calculateAlarmOperatorNote(GroupBase, Event)}<li>
 * {@link #calculateAlarmOtherAttribute(GroupBase, Action, Event)}
 */
public class MyFirstPropagation extends PropagationDefault implements
                                                       PropagationInterface {

/**
 *
 */
public MyPropagation() {

 super(LoggerFactory.getLogger(MyPropagation.class));
 setPublishAttributeForDebug(true);
}
/*
 * (non-Javadoc)
 *
 * @see
 * com.hp.uca.expert.vp.tp.core.PropagationDefault#calculateAlarmOperatorNote
 * (com.hp.uca.expert.group.GroupBase, com.hp.uca.expert.event.Event)
 */
@Override
public String calculateAlarmOperatorNote(GroupBase group,
Event referenceEvent) throws Exception {

if (log.isTraceEnabled()) {

 LogHelper.enter(log, "calculateAlarmOperatorNote()",
group.getName());
}
StringBuilder buf = new StringBuilder();
Boolean first = true;
Set<Alarm> wholeST = ((PropagationGroup) group).getWholeSubTreeRootCauses();

if (wholeST != null && !wholeST.isEmpty()) {

 for (Alarm s : wholeST) {
  if (!first) {
   buf.append(" | ");

  first = false;
  }
  buf.append(s.getIdentifier());
 }
}
String ret = buf.toString();
if (log.isTraceEnabled()) {
 LogHelper.exit(log, "calculateAlarmOperatorNote()", ret);
}
return ret;

}
```

```
/*
 * (non-Javadoc)
 *
 * @see
*com.hp.uca.expert.vp.tp.core.PropagationDefault#calculateAlarmOtherAttribute
 * (com.hp.uca.expert.group.GroupBase,
 * com.hp.uca.mediation.action.client.Action, com.hp.uca.expert.event.Event)
 */
@Override
public void calculateAlarmOtherAttribute(GroupBase group, Action action,
Event referenceEvent) throws Exception {

if (log.isTraceEnabled()) {
LogHelper.method(log, "calculateAlarmOtherAttribute()",
group.getName());
}
Map<String, String> otherAttributes = new HashMap<String, String>();
otherAttributes.put("ucaCustomField5",
String.format("dbNodeId:<%s>",
((PropagationGroup) group).getDbId()));
action.getVar().put("otherAttributes", otherAttributes); }
```

The Topology State Propagator framework will automatically invoke the methods `whatToDoWhenXXX(…)` listed in section 8.1, at predefined phases of the life cycle of alarms (depending on propagation context).

For instance, when the alarm 'alarm1', which is a root cause in '`propagationGroup1`', is cleared, the `TopologyStatePropagator` framework will invoke the method `whatToDoWhenRootCauseAlarmIsCleared`(`alarm1`, `propagationGroup1`).

If '`alarm1`' belongs to only one propagation "`Propagation A`", then the Topology State Propagator framework will invoke the method `whatToDoWhenRootCauseAlarmIsCleared (alarm1…)` from the customization class of "`PropagationA1`". If the method `whatToDoWhenRootCauseAlarmIsCleared ( )` has not been overridden for "`PropagationA1`", the default method is invoked.

If '`alarm1`' belongs **also** to "`Propagation B`", and is a root cause alarm for `PropagationB` as well, the Problem Detection framework will **also** invoke the method `whatToDoWhenRootCauseAlarmIsCleared (alarm1 …)`, if it is present in the customization class of "`Propagation B`", otherwise the default method is invoked.

Depending on the life cycle phase the alarm is in, the Topology State Propagator framework decides which method(s) `whatToDoWhenXXX(..)` to invoke.

8.3.2 My PropagationDefault

`PropagationDefault` class can also be extended like the `ProblemDefault` class (see section 7.4.3). By extending the `PropagationDefault` class, the default behavior for all propagations or for a set of propagations can be modified.

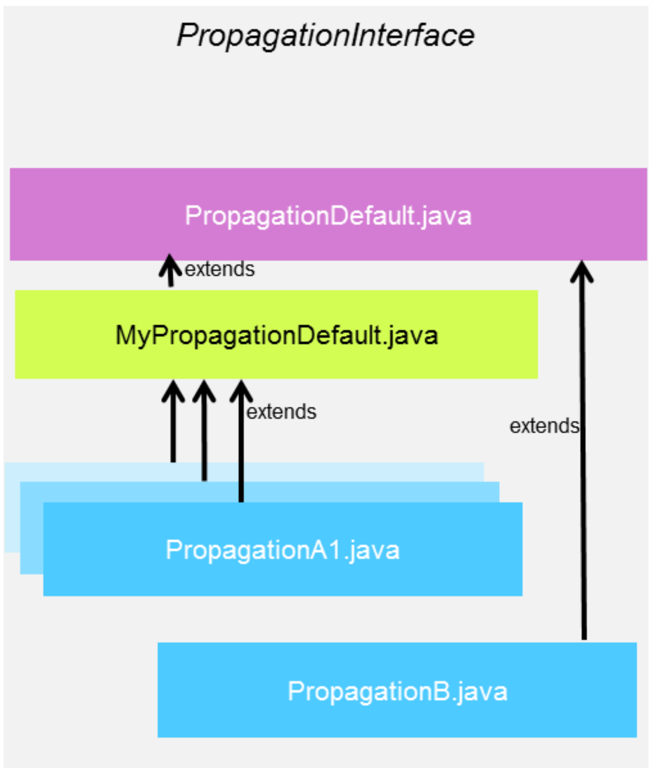


Figure 48 - MyPropagationDefault: a customization for a group of propagations

In Figure 48 - MyPropagationDefault: a customization for a group of propagations `MyPropagationDefault.java` implements some or all of the methods of the ***PropagationInterface***. Each propagation customization class that extends `MyPropagationDefault.java` will benefit from the implementation of those methods. In the diagram, by default, `PropagationA1`, `PropagationA2` and `PropagationA3` (the latter two hidden behind `PropagationA1`) will use the methods implemented in `MyPropagationDefault.java`. This happens only because the different propagation Java classes (`PropagationA1` to `A3`) extended in their java code the `MyPropagationDefault`. `PropagationB` will use the methods implemented in `PropagationDefault.java`, unless these methods are overridden in `PropagationB.java`.

For a comprehensive diagram showing the advanced possibilities and subtleties gained by extending `PropagationDefault.java`, see Annex F.

Propagations initialization

Propagations are initialized from the `PropagationXmlConfig.xml` defined in the `<propagationPolicy>` tag in the following way: all the policies defined in the `PropagationDefault` propagation policy (can be `MyPropagationDefault`) are applied to all other Propagations, unless they are overwritten by their respective custom propagation policy. Furthermore, for the policies seen in Table 24, Strings, Longs, Booleans (which contain a sequence of String, Long and Boolean types) defined in the `PropagationDefault` are valid for all the other Propagations. Even if defined in a custom `propagationPolicy`, they are added to those defined in the sequence, and are not overwritten. This applies also to topology policies: Nodes, `PoiCategories` and Threshold values seen in 5.4.2.2. To customize a specific behavior for each of the Propagations, it is better to delete the `PropagationDefault` configuration and redefine it for each of the custom propagation policies. It is recommended to identify what all

propagations have in common and define it only once in the PropagationDefault configuration.

The PropagationDefault (can be MyPropagationDefault) configuration is used to initialize a propagation whose policy is not defined in *the* PropagationXmlConfig.xml, but is defined as a top filter in a <topFilter> tag of TopologyPropagation_filters.xml file. Also, if no PropagationDefault policy tag is defined in the PropagationXmlConfig.xml file, the default values are applied from the PropagationDefault.java class.

In the following example, the PropagationDefault policies will apply for all other propagations defined. For example, by default, the enableServicealarmCreation is set to false (for Propagation_Switch and Propagation_Server), but is set to true when overwritten (in Propagation_PhoneService). The String “dummy” will apply for all propagations, but each propagation adds its own strings to this list. The node dbType location and the poi Location and RC will be found in all propagations, but for example the node dbType callServer and phonePool are added to this list for Propagation_PhoneService, as well as the poiCategory Service. The propagationRule is WorstChildPercentage for all propagations. The threshold values are set as in PropagationDefault for all propagations except for Propagation_Switch.

```
…
<propagationPolicy name="PropagationDefault">
  <serviceAlarm>
   <enableServiceAlarmCreation>false</enableServiceAlarmCreation>
   <delayForServiceAlarmCreation>0</delayForServiceAlarmCreation>
   <attachWholeSubTreeRootCauses>true</attachWholeSubTreeRootCauses>
 </serviceAlarm>
 <groupTickFlagAware>false</groupTickFlagAware>
 <propagationRule>
  <rule>WorstChildPercentage</rule>
   </propagationRule>
 <nodes>
  <dbType>
   <key><![CDATA[location]]></key>
  </dbType>
  </nodes>
 <poiCategories>
 <poiCategory>
  <key><![CDATA[LOCATION]]></key>
  </poiCategory>
 <poiCategory>
 <key><![CDATA[RC]]></key>
 </poiCategory>
 </poiCategories>
  <thresholdValues>
   <OK name="OK">
  <perceivedSeverity>CLEAR</perceivedSeverity>
 <availabilityPercentage>100.0</availabilityPercentage>
 <poiImportance>None</poiImportance>
 </OK>
 <LOW name="LOW">
 <perceivedSeverity>WARNING</perceivedSeverity>
 <availabilityPercentage>99.99999999</availabilityPercentage>
 <poiImportance>Low</poiImportance>
 </LOW>
 <MEDIUM name="MED">
 <perceivedSeverity>MINOR</perceivedSeverity>
 <availabilityPercentage>75.0</availabilityPercentage>
 <poiImportance>Medium</poiImportance>
  </MEDIUM>
 <HIGH name="HIGH">
 <perceivedSeverity>MAJOR</perceivedSeverity>
 <availabilityPercentage>50.0</availabilityPercentage>
```

```
 <poiImportance>High</poiImportance>
 </HIGH>
 <CRITICAL name="CRITICAL">
   <perceivedSeverity>CRITICAL</perceivedSeverity>
   <availabilityPercentage>25.0</availabilityPercentage>
  <poiImportance>Critical</poiImportance>
 </CRITICAL>
  <DOWN name="DOWN">
  <perceivedSeverity>CRITICAL</perceivedSeverity>
  <availabilityPercentage>0.0</availabilityPercentage>
  <poiImportance>Critical</poiImportance>
  </DOWN>
  </thresholdValues>
 <booleans />
  <strings>
  <p1:string key="dummy">
  <p1:value><![CDATA[ffff]]></p1:value>
 </p1:string>
  </strings>
 <longs />
</propagationPolicy>
…
<propagationPolicy name="Propagation_Server">
 <serviceAlarm></serviceAlarm>
 <groupTickFlagAware>false</groupTickFlagAware>
 <propagationRule>
  <rule>WorstChildPercentage</rule>
 </propagationRule>
<nodes>
<dbType>
 <key><![CDATA[switch]]></key>
   </dbType>
  </nodes>
  <booleans />
  <strings>
  <p1:string key="propagationObject">
    <p1:value><![CDATA[Server]]></p1:value>
   </p1:string>
   <p1:string key="statusName">
  <p1:value><![CDATA[state]]></p1:value>
  </p1:string>
  <p1:string key="percentageAvailabilityKey">
   <p1:value><![CDATA[percAvailability]]></p1:value>
  </p1:string>
  </strings>
  <longs />
</propagationPolicy>

<propagationPolicy name="Propagation_PhoneService">
 <serviceAlarm>
  <enableServiceAlarmCreation>true</enableServiceAlarmCreation>
 </serviceAlarm>
 <groupTickFlagAware>false</groupTickFlagAware>
 <propagationRule></propagationRule>
<nodes>
  <dbType>
   <key><![CDATA[phonePool]]></key>
 </dbType>
 <dbType>
   <key><![CDATA[callServer]]></key>
 </dbType>
  </nodes>
  <poiCategories>
   <poiCategory>
    <key><![CDATA[SERVICE]]></key>
   </poiCategory>
  </poiCategories>
  <booleans />
  <strings>
   <p1:string key="propagationObject">
     <p1:value><![CDATA[PhoneService]]></p1:value>
   </p1:string>
   <p1:string key="statusName">
    <p1:value><![CDATA[state]]></p1:value>
```

```
 </p1:string>
  <p1:string key="percentageAvailabilityKey">
 <p1:value><![CDATA[percAvailability]]></p1:value>
 </p1:string>
 </strings>
<longs />
</propagationPolicy>

<propagationPolicy name="Propagation_Switch">
 <serviceAlarm></serviceAlarm>
 <groupTickFlagAware>false</groupTickFlagAware>
<propagationRule>
 <rule>WorstChildPercentage</rule>
 </propagationRule>
<nodes>
<dbType>
<key><![CDATA[switch]]></key>
</dbType>
</nodes>
<poiCategories>
<poiCategory>
<key><![CDATA[SERVICE]]></key>
</poiCategory>
</poiCategories>
<thresholdValues>
<OK name="Normal">
<perceivedSeverity>CLEAR</perceivedSeverity>
<availabilityPercentage>100.0</availabilityPercentage>
<poiImportance>None</poiImportance>
</OK>
  <LOW name="LowDegraded">
 <perceivedSeverity>WARNING</perceivedSeverity>
  <availabilityPercentage>99.99999999</availabilityPercentage>
 <poiImportance>Low</poiImportance>
 </LOW>
 <MEDIUM name="MedDegraded">
  <perceivedSeverity>MINOR</perceivedSeverity>
   <availabilityPercentage>75.0</availabilityPercentage>
  <poiImportance>Medium</poiImportance>
 </MEDIUM>
 <HIGH name="HighDegraded">
 <perceivedSeverity>MAJOR</perceivedSeverity>
  <availabilityPercentage>50.0</availabilityPercentage>
 <poiImportance>High</poiImportance>
 </HIGH>
  <CRITICAL name="CriticalLyDegraded">
    <perceivedSeverity>CRITICAL</perceivedSeverity>
   <availabilityPercentage>25.0</availabilityPercentage>
      <poiImportance>Critical</poiImportance>
       </CRITICAL>
      <DOWN name="Down">
    <perceivedSeverity>CRITICAL</perceivedSeverity>
   <availabilityPercentage>0.0</availabilityPercentage>
      <poiImportance>Critical</poiImportance>
   </DOWN>
 </thresholdValues>
 <booleans />
 <strings>
  <p1:string key="propagationObject">
   <p1:value><![CDATA[Switch]]></p1:value>
  </p1:string>
   <p1:string key="statusName">
   <p1:value><![CDATA[state]]></p1:value>
  </p1:string>
  <p1:string key="percentageAvailabilityKey">
   <p1:value><![CDATA[percAvailability]]></p1:value>
  </p1:string>
 </strings>
 <longs />
</propagationPolicy>
…
```

8.3.3 MyGeneralBehavior

The same principles apply for the general behavior of propagations as for the non-problem specific behavior described in section 7.4.5. The methods that can be overridden to customize the “non-propagation specific” behavior of a Topology State Propagator Value Pack are listed in the **GeneralBehaviorInterface** Java interface.

A “non-propagation-specific” behavior is a behavior that is not related to any propagation in particular.

For example, the behavior of the initialization of a Topology State Propagator Value Pack is a “non-propagation-specific” behavior.

To customize a “non-propagation-specific” behavior do the following steps:

- Create a **MyGeneralBehavior.java** (name can be different) Java class in the following directory: *src/main/java/[com.hp.uca.expert.vp.tp.core]*.
- Ensure that the value of the property *generalBehaviorClassName* in the file *context.xml* in the *src/main/resources/valuepack/conf/* folder matches MyGeneralBehavior , *a*s shown in Figure 32.
- Override the methods of the **GeneralBehaviorInterface** for which the behavior has to be customized.

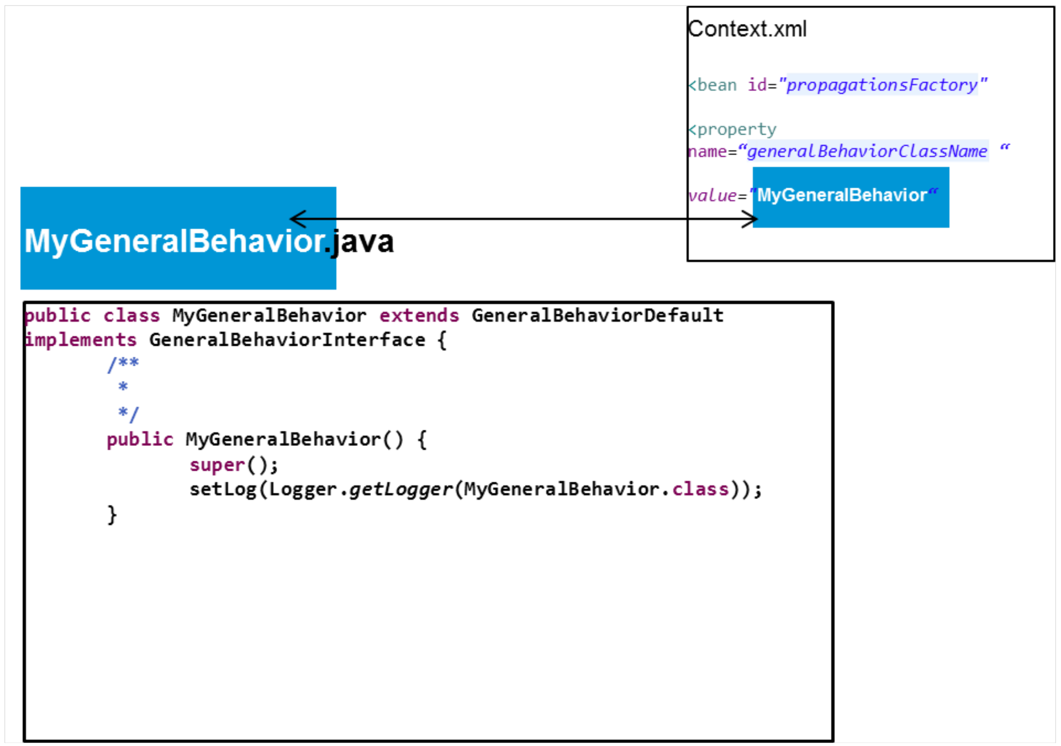


Figure 49 – TSP MyGeneralBehavior name matching

Below is an example of a MyGeneralBehavior.java class that overrides one method of the interface **GeneralBehaviorInterface**: computeSourceUniqueId().

```
public class MyGeneralBehavior extends GeneralBehaviorDefault implements
GeneralBehaviorInterface {

/**
 * Instantiates a new my general behavior.
 */
public MyGeneralBehavior() {
 super(LoggerFactory.getLogger(MyGeneralBehavior.class));
}

/*
 * (non-Javadoc)
 *
 * @see
 * com.hp.uca.expert.vp.tp.core.GeneralBehaviorDefault#computeSourceUniqueId
 * (com.hp.uca.expert.event.Event)
 */
@Override
public String computeSourceUniqueId(Event event) throws Exception {
 String ret = super.computeSourceUniqueId(event);
 return ret == null ? ret : ret.toUpperCase();
}}
```

Chapter 9

Troubleshooting

9.1 Logging

The logging configuration for an Inference Machine Value Pack (as for any UCA for EBC Value Pack) has to be done in the `${UCA_EBC_INSTANCE}/conf/uca-ebc-log4j.xml` file
on the UCA for EBC server.

The list of specific IM loggers is listed below:

Logger	Description
com.hp.uca.expert.vp.common.actions.db	Controls the execution of DB requests
com.hp.uca.expert.vp.common.actions.temip	Controls the execution of HP TeMIP actions (and TroubleTicket actions)
com.hp.uca.expert.vp.common.actions.GroupingKeys	Controls the execution of grouping keys computation
com.hp.uca.expert.vp.common.lifecycle	Controls Inference Machine Internals
com.hp.uca.expert.vp.common.services	Controls Inference Machine Services

Problem Detection specific loggers:

Logger	Description
com.hp.uca.expert.vp.pd.config.ProblemPropertie s	Controls the extraction of values from the XML configuration files
com.hp.uca.expert.vp.pd.core.XmlProblem	Controls the parsing of the XML of the XmlProblem customization
com.hp.uca.expert.vp.pd.core.ProblemDefault	Controls the execution of the default implementation of Problem Detection behavior
com.hp.uca.expert.vp.pd.core.internal.PD_AlarmR ecognition	Controls the decoding and setting of the roles of alarms
com.hp.uca.expert.vp.pd.core. internal.PD_Lifecycle	Controls the states propagation methods
com.hp.uca.expert.vp.pd.core. internal.PD_TroubleTicket	Controls the emission of Trouble Ticket requests
com.hp.uca.expert.vp.pd.core. internal.PD_Navigation	Controls the requests for updates on alarms
com.hp.uca.expert.vp.pd.core. internal.PD_Process	Controls the execution of operations of PD at a high level, (attaching a Sub-Alarm to a group, creating a Trouble Ticket, and so on)

com.hp.uca.expert.vp.pd.core.internal.ProblemDetection	Controls the execution of operations of PD at the highest level: the methods invoked directly from the rules
com.hp.uca.expert.vp.pd.problem	Controls the customization of classes
com.hp.uca.expert.vp.pd.im.lifecycle	Controls Problem Detection Internals
com.hp.uca.expert.vp.pd.services.PD_Service_Lifecycle	Controls Problem Detection Services for Life cycle
com.hp.uca.expert.vp.pd.services.PD_Service_ProblemAlarm	Controls Problem Detection Services for Problem Alarm
com.hp.uca.expert.vp.pd.services.PD_Service_Util	Controls Problem Detection Miscellaneous Services
com.hp.uca.expert.vp.pd.services.PD_Service_Navigation	Controls Problem Detection Services for Navigation
com.hp.uca.expert.vp.pd.services.PD_Service_Action	Controls Problem Detection Services for Actions
com.hp.uca.expert.vp.pd.services.PD_Service_TroubleTicket	Controls Problem Detection Services for Trouble Tickets

Topology State Propagator specific loggers:

Logger	**Description**
com.hp.uca.expert.vp.tp.config.PropagationProperties	Controls the extraction of values from the XML configuration files
com.hp.uca.expert.vp.tp.core.PropagationDefault	Controls the execution of the default implementation of Propagation behavior
com.hp.uca.expert.vp.tp.core.internal.TP_EventRecognition	Controls the decoding and setting of the roles of events
com.hp.uca.expert.vp.tp.core.internal.TP_Lifecycle	Controls the states propagation methods
com.hp.uca.expert.vp.tp.core.internal.TP_TroubleTicket	Controls the emission of Trouble Ticket requests
com.hp.uca.expert.vp.tp.core.internal.TP_Navigation	Controls the requests for updates on alarms and events
com.hp.uca.expert.vp.tp.core.internal.TP_Process	Controls the execution of operations of TSP at a high level, (attaching a subalarm to a group, creating a Trouble Ticket, …)
com.hp.uca.expert.vp.tp.core.internal.TopologyPropagation	Controls the execution of operations of TSP at the highest level: the methods invoked directly from the rules
com.hp.uca.expert.vp.tp.propagation	Controls the customization of classes
com.hp.uca.expert.vp.tp.im.lifecycle	Controls TSP Internals Life cycle
com.hp.uca.expert.vp.tp.services.TP_Service_Lifecycle	Controls TSP Services for Life cycle
com.hp.uca.expert.vp.tp.services.TP_Service_ServiceAlarm	Controls TSP Services for Service Alarm

com.hp.uca.expert.vp.tp.services.TP_Service_Util	Controls TSP Miscellaneous Services
com.hp.uca.expert.vp.tp.services.TP_Service_Navigation	Controls TSP Services for Navigation
com.hp.uca.expert.vp.tp.services.TP_Service_Action	Controls TSP Services for Actions
com.hp.uca.expert.vp.tp.services.TP_Service_TroubleTicket	Controls TSP Services for Trouble Tickets
com.hp.uca.expert.vp.tp.services.TP_Service_Group	Controls TSP Services for Grouping
com.hp.uca.expert.vp.tp.services.TP_Service_PointOfInterest	Controls TSP Services for Point Of Interest

In addition to these Inference Machine (PD, TSP, and common library) loggers, it is recommended to log with the following HP UCA EBC logger

`logger name="com.hp.uca.expert.filter"` with level

- DEBUG to trace why an alarm does not pass
- TRACE to trace why an alarm passes

Chapter 10

Annexes

Annex A. Migration steps

A.1. Migration steps from Version 3.2 to 3.3

The interface listed below is no more supported.

Type	API	Replaced by
Method	SupportedTroubleTicketActions.closeTroubleTicket(Action action, Scenario scenario, CommonActionInterface problemOrPropagation, String troubleTicketIdentifer)	closeTroubleTicket(Action action, Scenario scenario, **GroupBase group**, CommonActionInterface problemOrPropagation, String troubleTicketIdentifer)

Table 29 - Deprecated APIs in IM 3.3

Type	API	Deprecated by
Method	ProblemDefault.computeDelayForTroubleTicketCreation(Event event)	ProblemDefault.computeDelayForTroubleTicketCreation(Event event, **Group group**)
Method	PropagationDefault.computeDelayForTroubleTicketCreation()	PropagationDefault.computeDelayForTroubleTicketCreation(**PropagationGroup group**)

See [R12] *Unified Correlation Analyzer for EBC Inference Machine Release Notes*

A.2. Migration steps from Version 3.1 to 3.3

Since version 3.2, PB is now part of the Inference Machine, which embeds PD and TSP products. As PB and TSP have the exact same needs to execute actions on NMS (create alarm, clear alarm, group alarms, and so on), it has been decided to use a common ActionsFactory for this.

This common ActionsFactory is now part of a common library, which is delivering its own namespace.

As this namespace is different, the compatibility is broken, but, it brings some improvements:

- The logic of actions is separated from PD and TSP
- It is reusable as it is : the same ActionsFactory can be used across PD and TSP
- It is easier to understand

Deprecated APIs

The methods, classes, and packages listed in the following table are deprecated with this version and will be removed in next major update.

This is mainly due to the fact that most of these methods are now contained in the `uca-evp-common.jar` that is used also by the Topology State Propagator for Service Impact toolkit.

Table 30 - Deprecated APIs in PD 3.2

Type	API	Deprecated by
Package	com.hp.uca.expert.vp.pd.core.exception	com.hp.uca.expert.vp.common.exceptions
Method	ProblemDefault.computeDelayForTroubleTicketCreation(Alarm alarm)	ProblemDefault.computeDelayForTroubleTicketCreation(Event event)
Method	ProblemDefault.computeDelayForProblemAlarmCreation(Alarm alarm)	ProblemDefault.computeDelayForProblemAlarmCreation(Event event)
Method	ProblemDefault.computeDelayForProblemAlarmClearance(Alarm alarm)	ProblemDefault.computeDelayForProblemAlarmClearance(Event event)
Method	ProblemDefault.computeTimeWindow(Alarm alarm)	ProblemDefault.computeTimeWindow(Event event)
Method	PD_Service_Enrichment.setAlarmIsMissingInformation(Alarm a, String problemName)	PD_Service_Enrichment.setEventIsMissingInformation(Event e, String problemName)
Method	PD_Service_Enrichment.setAlarmIsNoMoreMissingInformation(Alarm a, String problemName)	PD_Service_Enrichment.setEventIsNoMoreMissingInformation(Event e, String problemName)
Method	PD_Service_Enrichment.isAlarmMissingInformation(Alarm a, String problemName)	PD_Service_Enrichment.isEventMissingInformation(Event e, String problemName)
Method	PD_Service_Enrichment.requestAlarmComputation(Scenario scenario, Alarm a)	PD_Service_Enrichment.requestEventComputation(Scenario scenario, Event e)
Method	PD_Service_Group.calculateLeadGroup(CollectionGroup groups)	PD_Service_Group.calculateLeadGroup(Collection<Group> groups, Boolean sorted)
Method	PD_Service_Group.isLeadGroup(Group potentialLeaderGroup, CollectionGroup groups)	PD_Service_Group.isLeadGroup(Group potentialLeaderGroup, Collection<Group> groups, Boolean sorted)
Method	PD_Service_Lifecycle.cloneAlarmToBeReEvaluated(Alarm alarm)	PD_Service_Lifecycle.cloneEventToBeReEvaluated(Event event)
Method	PD_Service_Util.extractSubString()	com.hp.uca.expert.vp.common.services.UtilService.extractSubString()
Method	PD_Service_Util.retrieveBeanFromContextXml()	com.hp.uca.expert.vp.common.services.UtilService.retrieveBeanFromContextXml()
Method	PD_Service_Util.fileFromResourceName()	com.hp.uca.expert.vp.common.services.UtilService.fileFromResourceName()
Method	PD_Service_Util.storeProblemInfosInAlarmLocalVariable(ProblemContext problemContext, Alarm alarm, ListProblemInfo problemInfos)	PD_Service_Util.storeProblemInfosInEventLocalVariable(ProblemContext problemContext, Event event, List<ProblemInfo> problemInfos)
Method	PD_Service_Util.retrieveProblemInfosFromAlarmLocalVariable(ProblemContext problemContext, Alarm alarm)	PD_Service_Util.retrieveProblemInfosFromEventLocalVariable(ProblemContext problemContext, Event event)
Class	TestUtils	com.hp.uca.expert.vp.common.testmaterial.TestUtils

Migrating PD VP 3.0/3.1 to 3.3

Problem Detection v3.3 does not provide any automatic migration tool for Java files.

However, the SDK provides an XLST (eXtensible Stylesheet Language Transformation) file that you can use to migrate the PD configuration file.

Java code

Removed classes

The following imports generate compilation errors because the classes do not exist anymore. The listed classes for v3.1 have to be replaced with the corresponding classes listed for v3.3.

Table 31 - Java classes removed in PD 3.3

Class in v3.1	Class in v3.3
import com.hp.uca.expert.vp.pd.config.Action	import com.hp.uca.expert.vp.im.config.Action
import com.hp.uca.expert.vp.pd.config.Actions	import com.hp.uca.expert.vp.im.config.Actions
import com.hp.uca.expert.vp.pd.config.BooleanItem	import com.hp.uca.expert.vp.im.config.BooleanItem
import com.hp.uca.expert.vp.pd.config.Booleans	import com.hp.uca.expert.vp.im.config.Booleans
import com.hp.uca.expert.vp.pd.config.LongItem	import com.hp.uca.expert.vp.im.config.LongItem
import com.hp.uca.expert.vp.pd.config.Longs;	import com.hp.uca.expert.vp.im.config.Longs
import com.hp.uca.expert.vp.pd.config.StringItem;	import com.hp.uca.expert.vp.im.config.StringItem
import com.hp.uca.expert.vp.pd.config.Strings	import com.hp.uca.expert.vp.im.config.Strings
Import com.hp.uca.expert.vp.pd.config.TroubleTicketAction	import com.hp.uca.expert.vp.im.config.TroubleTicketActio n
import com.hp.uca.expert.vp.pd.config.TroubleTicketActions	import com.hp.uca.expert.vp.im.config.TroubleTicketActio ns
import com.hp.uca.expert.vp.pd.core.exception.InvalidSuppo rtedActions	import com.hp.uca.expert.vp.common.exceptions.InvalidS upportedActions
import com.hp.uca.expert.vp.pd.core.exception.InvalidSuppo rtedTroubleTicketActions	import com.hp.uca.expert.vp.common.exceptions.InvalidS upportedTroubleTicketActions
import com.hp.uca.expert.vp.pd.interfaces.ActionsFactoriesS election	import com.hp.uca.expert.vp.common.interfaces.ActionsF actoriesSelection
import com.hp.uca.expert.vp.pd.interfaces.SupportedActions	import com.hp.uca.expert.vp.common.interfaces.Supporte dActions
Import com.hp.uca.expert.vp.pd.interfaces.SupportedTroubl eTicketActions	import com.hp.uca.expert.vp.common.interfaces.Supporte dTroubleTicketActions

Customized ProblemDefault

If you override the listed methods from *ProblemDefault*, they need to be changed because they do not exist anymore.
The listed methods for v3.1 have to be replaced with the corresponding methods listed for v3.3.

Table 32 - ProblemDefault method changes in PD 3.3

Method in v3.1	Method in v3.3
chooseSupportedActions(Alarm alarm, ProblemInterface problem)	chooseSupportedActions(Event event, CommonActionInterface problemOrPropagation)
chooseSupportedTroubleTicketActions(Alarm alarm, ProblemInterface problem)	chooseSupportedTroubleTicketActions(Event event, CommonActionInterface problemOrPropagation)

Customized *ActionsFactory*

If you override the following methods from *ActionsFactory*, they need to be changed because they do not exist anymore.
The listed methods for v3.1 have to be replaced with the corresponding methods listed for v3.3.

Table 33 - ActionsFactory method changes in PD 3.3

Method in v3.1	Method in v3.2
createProblemAlarm(Action action, Scenario scenario, Group group, ProblemInterface problem, Alarm referenceAlarm)	createAlarm(Action action, Scenario scenario, GroupBase group, CommonActionInterface problemOrPropagation, Event referenceEvent)
terminateAlarm(Action action, Scenario scenario, Alarm alarm, ProblemInterface problem)	terminateAlarm(Action action, Scenario scenario, Alarm alarm, CommonActionInterface problemOrPropagation)
clearAlarm(Action action, Scenario scenario, Alarm alarm, ProblemInterface problem)	clearAlarm(Action action, Scenario scenario, Alarm alarm, CommonActionInterface problemOrPropagation)
acknowledgeAlarm(Action action, Scenario scenario, Alarm alarm, ProblemInterface problem)	acknowledgeAlarm(Action action, Scenario scenario, Alarm alarm, CommonActionInterface problemOrPropagation)
unacknowledgeAlarm(Action action, Scenario scenario, Alarm alarm, ProblemInterface problem)	unacknowledgeAlarm(Action action, Scenario scenario, Alarm alarm, CommonActionInterface problemOrPropagation)
associateAlarmsForHistoryNavigation(Action action, Scenario scenario, Group group, Collection Alarm children, ProblemInterface problem)	associateAlarmsForHistoryNavigation(Action action, Scenario scenario, GroupBase group, Collection Alarm children, CommonActionInterface problemOrPropagation)
dissociateAlarmsForHistoryNavigation(Action action, Scenario scenario, Group group, Collection Alarm children, ProblemInterface problem)	dissociateAlarmsForHistoryNavigation(Action action, Scenario scenario, GroupBase group, Collection Alarm children, CommonActionInterface problemOrPropagation)
setHistoryNavigation(Action action, Scenario scenario, Alarm alarm, Qualifier qualifier)	setHistoryNavigation(Action action, Scenario scenario, Alarm alarm, QualifierInterface qualifier)
setGenericAttribute(Action action, Scenario scenario, Alarm alarm, Command command)	setGenericAttribute(Action action, Scenario scenario, Alarm alarm, Command command)

Customized TroubleTicketActionsFactory

If you override the following methods from *TroubleTicketActionsFactory*, they need to be changed because they do not exist anymore.
The listed methods for v3.1 have to be replaced with the corresponding methods listed for v3.3.

Table 34 - TroubleTicket method changes in PD 3.3

Method in v3.1	Method in v3.3
createTroubleTicket(Action action, Scenario scenario, Group group, ProblemInterface problem, Alarm referenceAlarm, List Alarm alarmsToAssociate)	createTroubleTicket(Action action, Scenario scenario, GroupBase group, CommonActionInterface problemOrPropagation, Alarm referenceAlarm, List Alarm alarmsToAssociate)
closeTroubleTicket(Action action, Scenario scenario, ProblemInterface problem, String troubleTicketIdentifer)	closeTroubleTicket(Action action, Scenario scenario, CommonActionInterface problemOrPropagation, String troubleTicketIdentifer)

associateTroubleTicket(Action action, Scenario scenario, Group group, ProblemInterface problem, List Alarm alarmsToAssociate, String troubleTicketIdentifer)	associateTroubleTicket(Action action, Scenario scenario, GroupBase group, CommonActionInterface problemOrPropagation, List Alarm alarmsToAssociate, String troubleTicketIdentifer)
dissociateTroubleTicket(Action action, Scenario scenario, Group group, ProblemInterface problem, List Alarm alarmsToDissociate, String troubleTicketIdentifer)	dissociateTroubleTicket(Action action, Scenario scenario, GroupBase group, CommonActionInterface problemOrPropagation, List Alarm alarmsToDissociate, String troubleTicketIdentifer)

XML configuration

The *ProblemXMLConfig.xml* file, or its equivalent, needs to be modified to make use of the new *http://config.im.vp.expert.uca.hp.com/* namespace for certain elements of the file, like:

- Actions
- TroubleTicketActions
- Booleans
- Longs
- Strings

You can use the `ProblemXmlConfig-Migration-to-V32.xslt` file, which is part of the Inference Machine SDK, to transform your current `ProblemXmlConfig.xml` version 3.1 to version 3.3.

Using Eclipse, perform the following steps:

1. Select the `ProblemXmlConfig.xml` file.
2. Right-click and choose **Run As > XSL Transformation**.
3. Clicking Add External Files to add the input file.
4. Select the xslt file provided under `${UCA_EBC_DEV_HOME}/schemas`

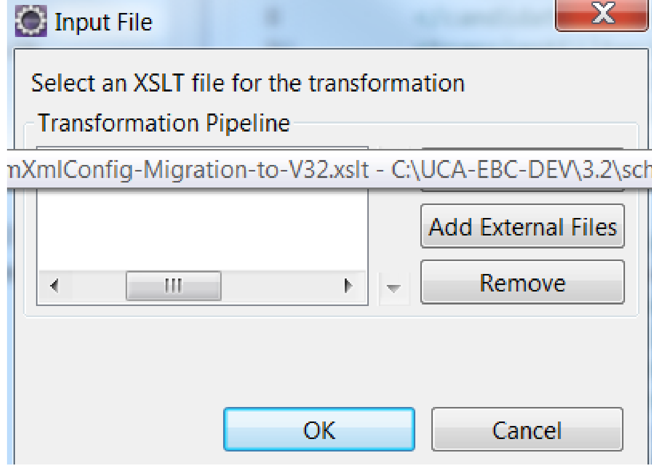


Figure 50 - Selecting the XSLT transformation file

5. Click OK.

If you receive errors like `Namespace for prefix 'p1' has not been declared`, the probable cause is that you are not using the right processor to transform your XML. In such a case:

1. Choose Run configurations.
2. Choose the last run.
3. Click on Processor tab.
4. Use specific processor: Xalan or Saxon (depending on your settings).
5. Click Run.

Annex B. Problem Detection Value Pack example

As part of the Inference Machine Development Kit, an example Value Pack project, called `pd-example`, is available.

If deployed, the pd-example Value Pack is able to recognize four problems:

- `Problem_BitError`
- `Problem_Synch`
- `Problem_Power`
- `XmlGeneric_Synch`

Each of these problems have specific filters.

`Problem_BitError`, `Problem_Synch` and `Problem_Power` are problems extending the ProblemDefault Java class, by overriding some of its methods. `XmlGeneric_Synch` is also an extended problem, but customized through XML (in the `src/main/resources/valuepack/conf/ProblemXmlConfig.xml` file)

Alarm enrichment, Action Factory and Trouble Ticket Action Factory examples are also provided for each problem. In addition a sample tests file is provided that can be executed with JUnit. These tests simulate the deployed behavior of the pd-example Value Pack without having to actually deploy it. Alarms are injected in the Value Pack as though they came from the network.

This chapter describes the contents and structure of the Value Pack example pack.

Contents of the src/main/java directory of pd-example

B.1. The src/main/java directory of the Problem Detection Value Pack example contains code customization.

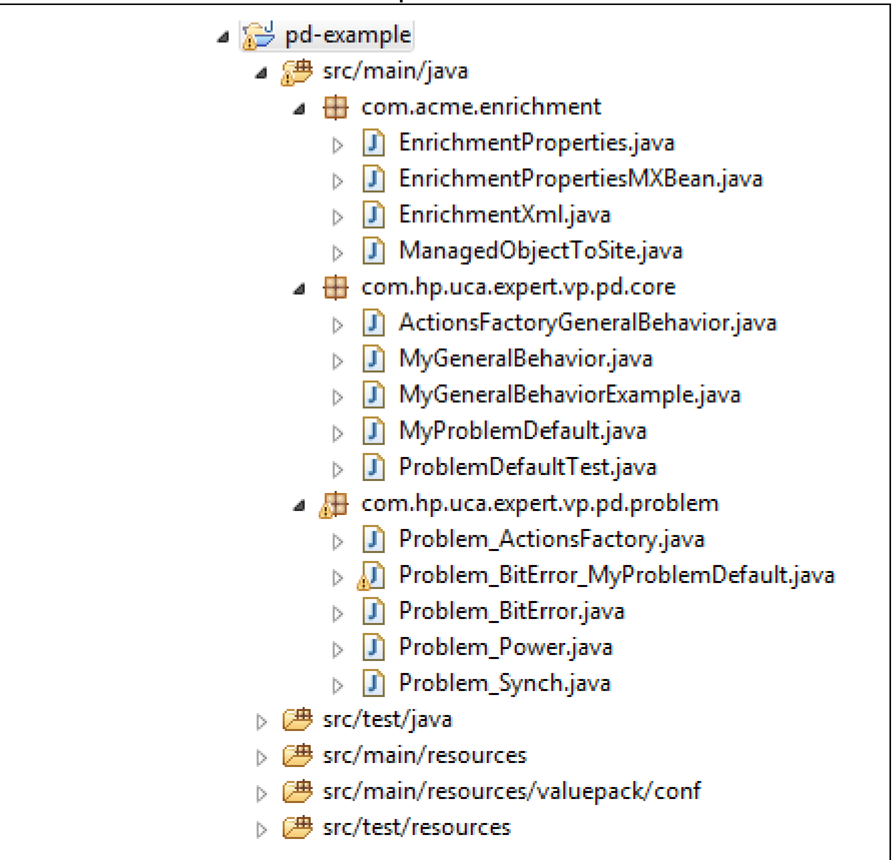


Figure 51 – pd-example src/main/java directory contents

com.acme.enrichment package

This package contains classes used to read the Enrichment.xml XML file called present in src/main/resources/valuepack/conf.
It contains information to enrich alarms and acts as an association table. Locating the managedObject of an alarm its associated site can be identified.

Example code from Enrichment.xml is as follows:

```
      <managedObjectToSite>
<managedObject>motorola_omcr_system [...] 5 btssitemgr 0 msi 18 mms
0 </managedObject>
     <site>bsc khorfakkan_bsc24 bts bridippm_6185</site>
      </managedObjectToSite>
```

- The MissingInfoAlarmPowerTest.java file is present in src/test/java/ft/enrichment is a test file sending alarms. Sent alarms belong to the Problem_Power problem and need to be enriched with site information.
- EnrichmentProperties.java is the class that contains methods to read the Enrichment.xml file.
- EnrichmentPropertiesMXBean.java is the interface implemented by EnrichmentProperties.java
- EnrichmentXml.java and ManagedObjectToSite.java create data structures to store the enrichment information.

com.hp.uca.expert.vp.pd.core package

- `ActionsFactoryGeneralBehavior.java` contains an example of the `whatToDoWhenAlarmIsJustInserted()` method that needs to be overrided for enrichment.
- `MyGeneralBehavior.java` & `MyGeneralBehaviorExample.java` also contain example override methods of the GeneralBehaviorInterface. See 7.4.5 MyGeneralBehavior
- `MyProblemDefault.java` illustrates override methods of the ProblemInterface for a subset of problems. See 7.4.3 My ProblemDefault.

com.hp.uca.expert.vp.pd.problem package

The `com.hp.uca.expert.vp.pd.problem` package contains problem customization classes for the four sample problems supplied by pd-example.

Table 35 - Overrides provided for pd-example problems

File	Overrides
Problem_BitError.java	calculateProblemAlarmAdditionalText computeProblemEntity isAllCriteriaForProblemAlarmCreation
Problem_Sync.java	calculateProblemAlarmAdditionalText computeProblemEntity isAllCriteriaForProblemAlarmCreation calculateProblemAlarmEventTime
Problem_Power.java	calculateProblemAlarmAdditionalText computeProblemEntity isAllCriteriaForProblemAlarmCreation calculateProblemAlarmSeverity isInformationNeededAvailable isMatchingProblemAlarmCriteria
Problem_BitError_MyProblemDefault.java	calculateProblemAlarmAdditionalText computeProblemEntity isAllCriteriaForProblemAlarmCreation calculateProblemAlarmSeverity
Problem_ActionsFactory.java	calculateProblemAlarmAdditionalText computeProblemEntity isAllCriteriaForProblemAlarmCreation isMatchingSubAlarmCriteria isMatchingTriggerAlarmCriteria

Contents of the of src/test/java directory of pd-example

This directory contains the source code of JUnit tests used to simulate the behavior of the pd-example value pack. It also contains Actions Factory customization examples.

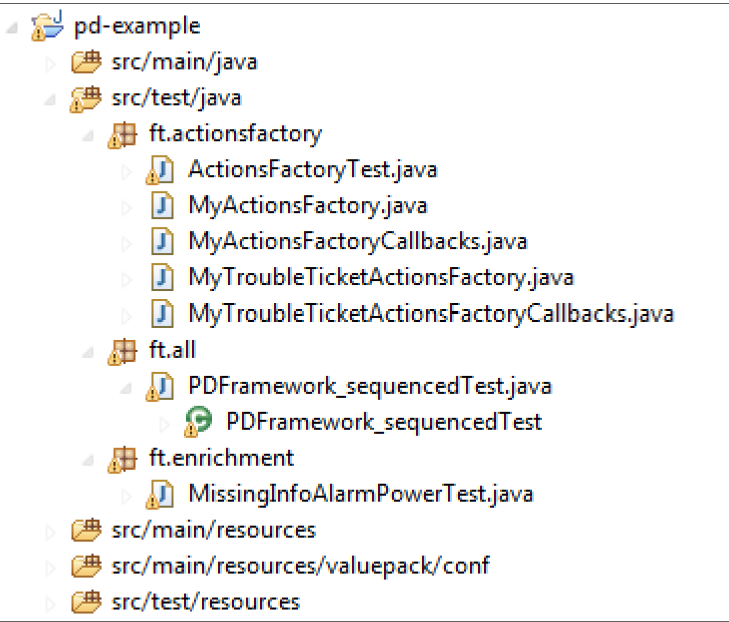


Figure 52 - pd-example src/test/java directory contents

ft.actionsfactory package

A Problem Detection Value Pack receives alarms from a Network Management System (NMS), performs alarm processing, then requests the NMS to execute actions.

The list of supported actions is defined in the `SupportedActions` Java interface. The SupportedActions interface defines methods such as `createProblemAlarm(), terminateAlarm(), clearAlarm()`

- The `ActionsFactory.java` class is a simple implementation of the `SupportedActions` interface.
- If an NMS other than HP TeMIP is used, an implementation of the `SupportedActions` interface must be created on the model of MyActionsFactory.java.

  Problem Detection provides `TeMIPActionsFactory.java`, a working implementation of `SupportedActions` for the scenario when HP TeMIP is the NMS.
- `MyActionsFactoryCallback.java` contains the callbacks methods that the NMS must call after executing some of the actions.

A Problem Detection Value Pack can create and manage trouble tickets. The possible interactions between the Problem Detection Value Pack and a trouble ticketing system are listed in the `SupportedTroubleTicketActions.java` interface. The `SupportedTroubleTicketActions` interface defines methods such as `createTroubleTicket(), closeTroubleTicket().`

- The `TroubleTicketActionsFactory.java` class is a simple implementation of the `SupportedTroubleTicketActions` interface.

- If a trouble ticketing system other than HP Service Manager (part of HP TeMIP) is used, an implementation of the `SupportedTroubleTicketActions` interface must be created on the model of `MyTroubleTicketActionsFactory.java`.

  Problem Detection provides `TeMIPTroubleTicketActionsFactory.java`, a working implementation of `SupportedTroubleTicketActions` for the scenario when HP Service Manager is the ticketing system in use.

- `MyTroubleTicketActionsFactoryCallback.java` contains the callbacks methods that the trouble ticketing system must call after executing some of the requests.
- *`ActionsFactoryTest.java`* is a test file that simulates sending alarms and then checks that the necessary actions are performed.

ft.all package

`PDFramework_sequencedTest.java` is a test file. It sends alarms corresponding to the problems provided by pd-example.

It checks the following:

- Problems are detected
- Problem Alarms are created
- Sub-Alarms are tagged
- The number of groups created is correct
- The number of actions executed is correct

ft.enrichment package

`MissingInfoAlarmPowerTest.java` is a test file. It sends alarms that need to be enriched and checks whether the enrichment was successful.

Contents of the src/main/resources directory of pd-example

The `src/main/resources` directory of the Problem Detection Value Pack example contains configuration files.

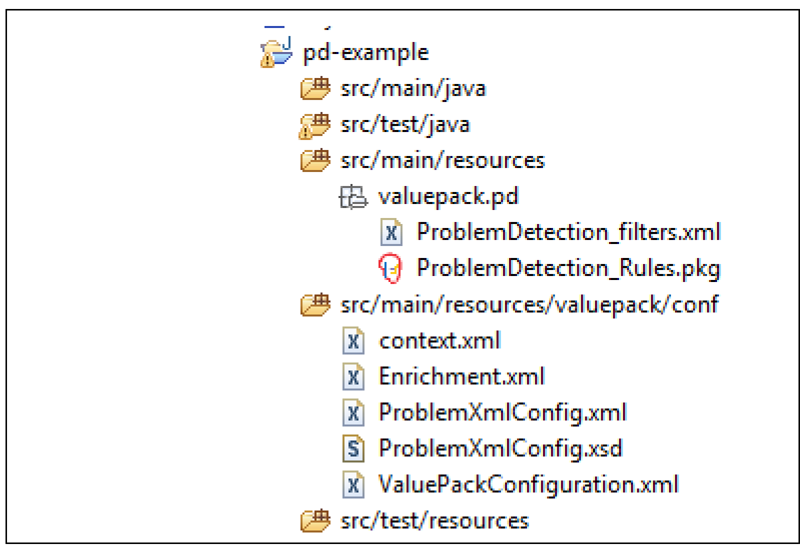


Figure 53 - pd-example src/main/resources directory contents

Filters

Filters are available in:
`src/main/resources/valuepack/pd/ProblemDetection_filters.xml`

The top filters correspond to the four problems delivered by the Value Pack example:

- Problem_Synch
- Problem_Power
- Problem_BitError
- XmlGeneric_Synch

```
<topFilter name="Problem_Synch">
<topFilter name="Problem_Power">
<topFilter name="Problem_BitError">
<topFilter name="XmlGeneric_Synch">
```

Rules

Rules are available in:
`src/main/resources/valuepack/pd/`*`ProblemDetection_Rules.pkg`*

Configuration

Configuration files are available in `src/main/resources/valuepack/conf`

- `context.xml` declares that the Problem Detection Value Pack relies on a customization of the `GeneralBehavior`.
- `Enrichment.xml` contains data to enrich alarms belonging to `Problem_Power`
- `ProblemXmlConfig.xml` contains the main policies, for example which Actions Factory to use; and the problem specific policies, for example the time window of each problem.

- `ProblemXmlConfig.xsd` contains the XML schema of `ProblemXmlConfig.xml`
- `ValuePackConfiguration.xml` defines the configuration of the Value Pack and its scenarios, the scenario policies, and the mediation flows.

Contents of the src/test/resources directory of pd-example

The `src/test/resources` directory of the Problem Detection Value Pack example contains test configuration files.

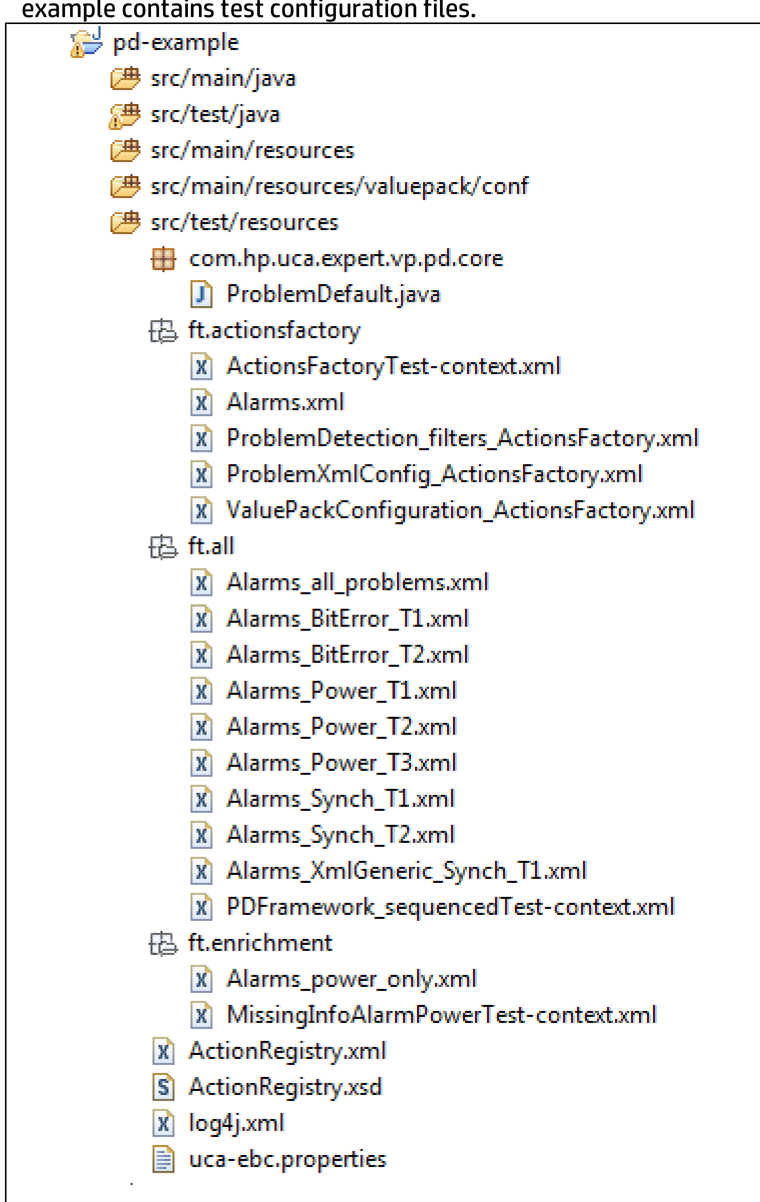


Figure 54 - pd-example src/test/resources directory contents

com.hp.uca.expert.vp.pd.core package

Contains `ProblemDefault` implementation located in `src/test/resources/com/hp/uca/expert/vp/pd/core/`

ft.actionsfactory

Each JUnit test must be executed with a specific configuration for the Value Pack. For example the JUnit test file `ActionsFactoryTest.java`, must use `ActionsFactoryTest-context.xml` as its context file. The naming scheme for the context file is `<test file name>-context.xml`.

This context file points at `ProblemXmlConfig_ActionsFactory.xml`, which is the policies configuration file, and at the main Value Pack configuration file: `ValuePackConfiguration_ActionsFactory.xml`.

This main configuration file points to the filters file: `ProblemDetection_filters_ActionsFactory.xml`.

`Alarms.xml` is the file describing the simulated alarms to be sent by the test `ActionsFactoryTest.java`.

ft.all

This package contains the alarms files used by the JUnit test file `PDFramework_sequencedTest.java`. The JUnit test file sends alarms in sequence from each alarms file one by one.

It is also possible to send all alarms simultaneously by using the `Alarms_all_problems.xml` file.

- `Alarms_BitError_T1.xml` contains alarms belonging to `Problem_BitError` and grouped in a group different from the group where alarms coming from `Alarms_BitError_T2.xml` are gathered
- `Alarms_BitError_T2.xml` contains alarms belonging to `Problem_BitError` and grouped in a group different from the group where alarms coming from `Alarms_BitError_T1.xml` are gathered
- `Alarms_Power_T1.xml` contains alarms belonging to `Problem_Power` and grouped in a group different from the groups where alarms coming from `Alarms_ Power _T2.xml` and `Alarms_ Power _T3.xml` are gathered
- `Alarms_Power_T2.xml` contains alarms belonging to `Problem_ Power` and grouped in a group different from the groups where alarms coming from `Alarms_ Power _T1.xml` and `Alarms_ Power _T3.xml` are gathered
- `Alarms_Power_T3.xml` contains alarms belonging to `Problem_ Power` and grouped in a group different from the groups where alarms coming from `Alarms_ Power _T1.xml` and `Alarms_ Power _T2.xml` are gathered
- `Alarms_Synch_T1.xml` contains alarms belonging to `Problem_Synch` and grouped in a group different from the group where alarms coming from `Alarms_Synch_T2.xml` are gathered.
- `Alarms_Synch_T2.xml` contains alarms belonging to `Problem_Synch` and grouped in a group different from the group where alarms coming from `Alarms_Synch_T1.xml` are gathered
- Alarms_`XmlGeneric_Synch_T1.xml` contains alarms belonging to problem `XmlGeneric_Synch`.

- `PDFramework_sequencedTest-context.xml` contains the context file of the `PDFramework_sequencedTest.java` test file.

ft.enrichment

- `Alarms_power_only.xml` contains alarms sent by `MissingInfoAlarmPowerTest.java`
- `MissingInfoAlarmPowerTest-context.xml` is the context file of the `MissingInfoAlarmPowerTest.java` test file.

Like any HP UCA for EBC Value Pack, the pd-example Value Pack, if deployed, can send action requests to be executed by the mediation layer associated with UCA for EBC Server: HP OSS Open Mediation V6.0. for example.

The actions are executed by a Channel Adapter (specific to a target application) on the mediation layer. Action replies are then returned to the pd-example Value Pack.

HP UCA for EBC Value Pack scenarios use web services to communicate with the Action Service web service of a Channel Adapter, typically the HP UCA for EBC Channel Adapter.

For these actions to be properly routed to the mediation layer and then to the correct Channel Adapter and target application, the file `ActionRegistry.xml` must be configured correctly.

For details on how to configure the `ActionRegistry.xml` file see the [R11] *UCA for EBC Administration, Configuration and Troubleshooting Guide*, and in particular the 'uca-ebc.properties file configuration' chapter.

ActionRegistry.xsd

Contains the XML schema for ActionRegistry.xml.

log4j.xml

Contains the different log levels that can be configured for the entire set of JUnit tests of the pd-example Value Pack.

uca-ebc.properties

Contains the different properties that can be configured for HP UCA -EBC Server. This file generally does not need to be modified. For more details, see the [R11] *UCA for EBC Administration, Configuration and Troubleshooting Guide*, and in particular the 'ActionRegistry.xml file configuration' chapter.

Annex C.

Problem Detection Advanced customization

Problem Detection behavior customization

As seen in section 7.4 Customizing default behavior it is possible to modify the default behavior of Problem Detection Value Packs.

The behavior can be modified in the following aspects:

- Per problem
- Per family of problems
- For all problems
- For non problem specific matters

Per problem

To modify the behavior of Problem Detection for a given problem, an override must be defined on some *ProblemInterface* methods in the customization class of the problem.

Per family of problems

To modify the default behavior of Problem Detection for a set of problems:

1st step -- Create a customization class (for example MyFamilyOfProblems)that implements some override methods over the *ProblemInterface.*

2nd step – For each problem in the family, create a customization class that extends the MyFamilyOfProblems customization class for the problem.

For all problems

Modifying the default behavior of Problem Detection for all problems is identical as doing it for a family of problems. The only difference is that the customization class of each problem must extend one class (for example MyAllProblemsDefault” (this name is given as an example) class

For non problem specific matters

The Problem Detection framework offers the possibility to modify system behavior not linked to problems, through the creation of a customization class (for example MyGeneralBehavior), and overriding methods of the *GeneralBehaviorInterface* interface such as `whatToDoWhenProblemDetectionIsInitialized( )`, `whatToDoWhenNewAlarmIsJustInserted( )`

The `context.xml` file in the `src/main/resources/valuepack/conf/` folder needs to be modified to specify Problem Detection that the customized

implementation of the methods of the GeneralBehaviorInterface methods are available **and only available** in the MyGeneralBehavior class.

Therefore it is not required to override any GeneralBehaviorInterface method anywhere else other than the class specified in the `context.xml` file.

GeneralBehaviorInterface defines methods such as "`whatToDoWhenProblemDetectionIsInitialized()`" which are not specific to any problem, and are not invoked by the Problem Detection framework on a problem object. Therefore it is not required to provide an implementation of those methods in the customization class of the problems.

The figure below shows an example of the following:

- a "*per problem*" customization in Problem1.java
- a "*per family of problems*" customization inMyFamilyOfProblems.java for Problem 2 and Problem 3
- a "*non problem specific*" customization in MyGeneralBehavior.java

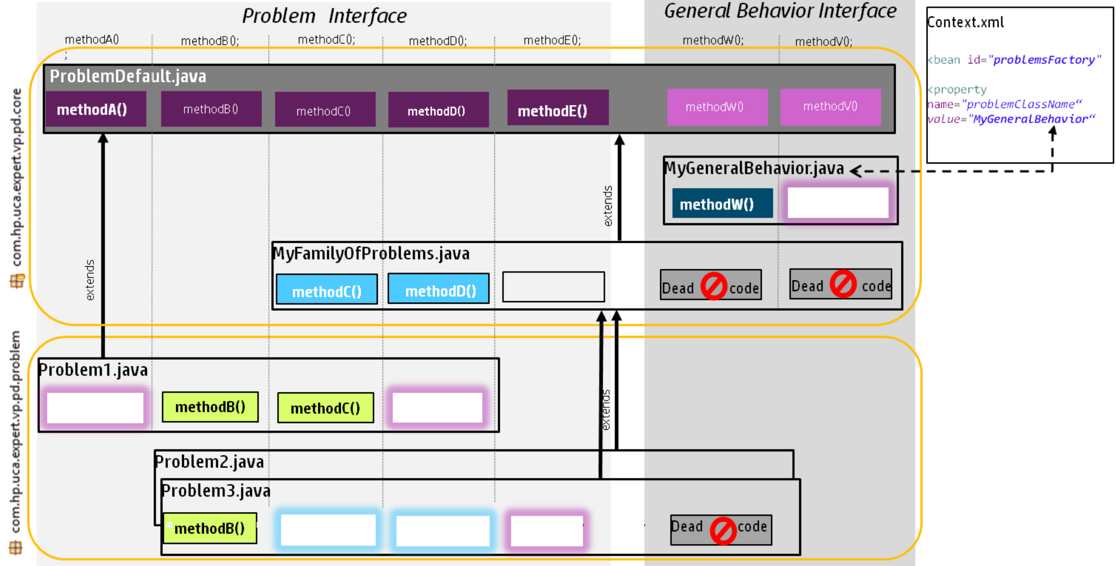
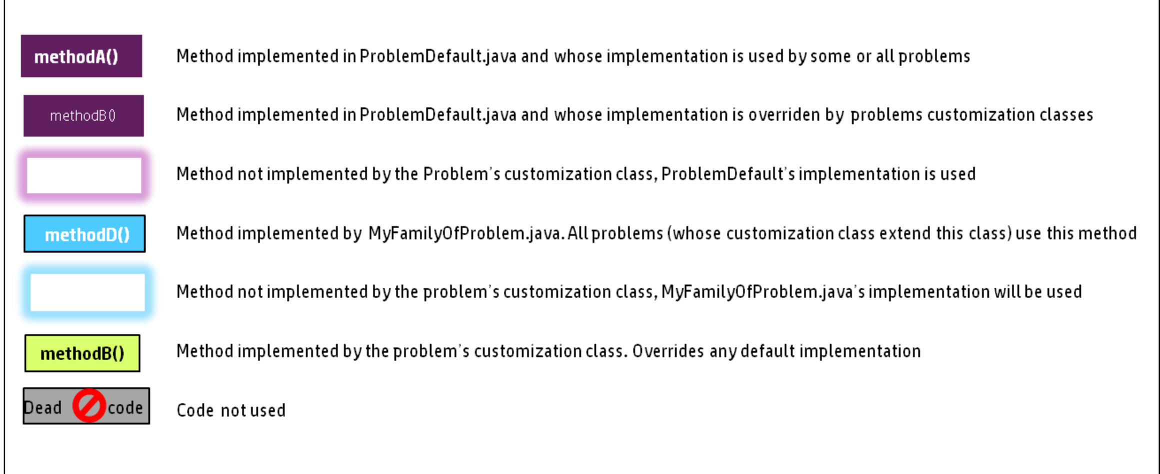


Figure 55 - Implementation schema of the main Problem Detection interfaces



Problem Entity, Multiple Problem Entities, Problem key

Problem Entity / Problem Entities definition

For each alarm passing the filters, Problem Detection calculates one or multiple problem entities. These problem entities represent the affected modules, elements and services.

For example:

1) An Alarm reporting the crash of a processor can be related to processor ID as the problem entity.

2) An Alarm reporting the fact that a server is unavailable can be related to the server name as the problem entity.

3) An Alarm reporting a pipe cut between two machines (machine A and B) can be related to machine A and machine B as the problem entity.

Problem Key definition

Each alarm passing the filters can have one or multiple problem entities. A problem key is associated with each problem entity.

The problem key defines a perimeter equal or larger than the problem entity. All alarms that pass the same filter, and share the same problem key, are considered for potential grouping.

Table 36 - Example Problem Keys

Problem	Problem entity	Problem key
An alarm reporting the crash of a processor	Processor ID	The server in which the processor is located
An alarm reporting a server is unavailable	Server name	Server name
An alarm reporting a pipe cut between two machines (machine A and B)	Machine A and Machine B	Site containing machine A Site containing machine B

Role of Problem Entity / Problem Entities / Problem Key in grouping

When alarms are grouped, the problem entity of the alarms is taken into account.

Example 1: All the alarms have the same problem entity and problem key.

Table 37 - Problem key grouping example 1

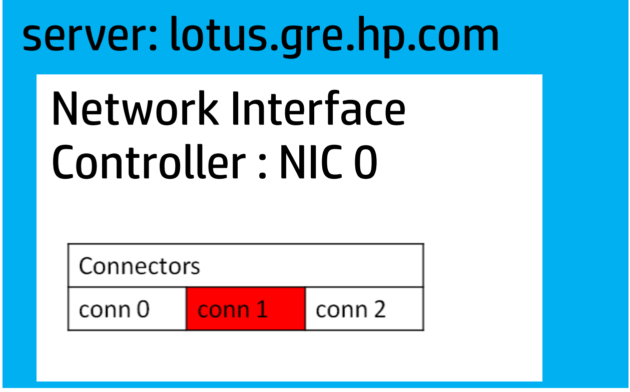
Alarm	Problem entity	Problem key
Destination Host Unreachable	lotus.gre.hp.com	lotus.gre.hp.com
Server down	lotus.gre.hp.com	lotus.gre.hp.com
Fans stopped working	lotus.gre.hp.com	lotus.gre.hp.com

In this case, all alarms have the same problem key, so they are considered for grouping. They also have the same problem entity so they will be grouped. The group receives the same problem entity as the included alarms.

Example 2: All alarms have the same problem key and a similar problem entity

Table 38 - Problem key grouping example 2

Alarm	Problem entity	Problem key
Destination Host Unreachable	lotus.gre.hp.com	lotus.gre.hp.com
Network Interface Controller down (Trigger alarm)	lotus.gre.hp.com__NIC_0	lotus.gre.hp.com
8B8C connector down	lotus.gre.hp.com__NIC_0__conn1	lotus.gre.hp.com



In this case, all alarms have the same problem key, so they will be considered for grouping. They also have a similar problem entity: all problem entities are a superstring or a substring to the problem entity of the trigger alarm.

The method to override `compareProblemEntities` decides for each alarm whether to be part of the group.
The group receives the problem entity of the trigger alarm: lotus.gre.hp.com__NIC_0

Example 3: Some alarms have multiple problem entities.

Table 39 - Problem key grouping example 3

Alarm no.	Alarm	Problem entity	Problem key
1	Remote site not accessible	Site GRE	lotus.gre.hp.com
2	Broken pipe	Site GRE Site VBE	lotus.gre.hp.com nenufar.vbe.hp.com
3	Remote site not accessible	Site VBE	nenufar.vbe.hp.com

The connection between the two machines lotus and nenufar, and therefore the connection between the two sites GRE and VBE, is broken.

The `sameGroupForAllProblemEntities` property controls grouping behavior:

- If the `sameGroupForAllProblemEntities` property is set to false (default value), two groups will be created:

  Group 1 contains alarm 1 and alarm 2:

```
(groupname = <p>problem name</p> <e>lotus.gre.hp.com</e>
   group keys = <p>problem name</p> <k>site GRE</k>
```

Group 2 contains alarm 3 and alarm 3:

```
(groupname = <p>problem name</p> <e>nenufar.vbe.hp.com</e>
   group keys = <p>problem name</p> <k>site VBE</k>
```

- If the `sameGroupForAllProblemEntities` property is set to true, only one group will be created containing all alarms. The problem name is a random choice from the two available options.

```
Group 1 (groupname =
<p>problem name</p> <e>lotus.gre.hp.com</e>

OR

<p>problem name</p> <e>nenufar.vbe.hp.com</e>
   group keys = <p>problem name</p> <k>site GRE</k>
                <p>problem name</p> <k>site VBE</k>
```

ActionsFactory implementation

A Problem Detection Value Pack needs to send actions to the various NMS it takes alarms from. For example, a Problem Detection Value Pack informs a particular NMS to clear an Alarm, or to create a Problem Alarm.

The actions allowed to be invoked by the Problem Detection framework is defined in the SupportedActions interface.
For details, see [R6] *UCA for EBC Inference Machine – JavaDoc (%UCA_EBC_DEV_HOME%\apidoc\inference-machine\index.html)*

A Problem Detection Value Pack needs to implement the SupportedActions interface for each NMS it connects to.

For example, if a Problem Detection Value Pack receives alarms from HP TeMIP, SCOM and SMARTS, it has to provide one SupportedActions interface for each.

The SupportedActions interface implementation must be done by extending the `com.hp.uca.expert.vp.pd.actions.ActionsFactory` abstract class. This class provides common code to be extended.

HP TeMIP Actions Factory example

HP UCA EBC Problem Detection provides an implementation of the SupportedActions interface for HP TeMIP in the `uca-evp-pd-fwk.jar` file.

The following code example from the TeMIPActionsFactory class shows how the `clearAlarm()` method is implemented:

```
public class TeMIPActionsFactory extends ActionsFactory implements
                                               SupportedActions {

@Override
public Action clearAlarm(Action action, Scenario scenario, Alarm alarm,
ProblemInterface problem) throws Exception {

  action.addCommand("directiveName", "CLEARALARM");

  action.addCommand("entityName" alarm.getIdentifier());

  action.addCommand("UserId", UCA_EXPERT_ACTION_ID + action.getActionId());
```

```
  createAndSetCallback(action, scenario, TeMIPActionsFactoryCallbacks.class,
"clearAlarmCallback", scenario, action, alarm);

  return action;
}
```

Note that the method **createAndSetCallback** is defined and implemented in `com.hp.uca.expert.vp.pd.actions.ActionsFactory`

The following code example from the TeMIPActionsFactoryCallbacks class shows how the `clearAlarmCallback` method defined in the TeMIPActionsFactory class, is implemented

```
public class TeMIPActionsFactoryCallbacks {

 public static void clearAlarmCallback(Scenario scenario, Action action,
                                      Alarm referenceAlarm) {

   switch (action.getActionStatus()) {
      case Failed:
      String rawText = null;
      if (action.getListActionResponseItem() != null
      && action.getRawText() != null) {
        rawText = XmlUtils.xmlToString(action.getRawText());
                                              }

     if (rawText != null) {
       if (rawText.contains(SOURCE_OF_THE_ERROR_CLEAR_ALARM)) {
         if (LOG.isDebugEnabled()) {
LOG.debug(ALARM_WAS_ALREADY_CLEARED_FORCING_ACTION_STATUS_TO_COMPLETED);
         }
         action.acknowledgeActionFailure();
       }
       else if (rawText.contains(ENTITY_NON_EXISTENT)) {
         if (LOG.isDebugEnabled()) {
           LOG.debug(ALARM_WAS_DELETED_FORCING_ACTION_STATUS_TO_COMPLETED);
         }
         action.acknowledgeActionFailure();
       }
     }
     break;
     default:
     break;
   }
   if (LOG.isTraceEnabled()) {
     LogHelper.exit(LOG, "clearAlarmCallback()");
   }
 }
```

Non-HP TeMIP Actions Factory example

Any Actions Factory implementation class needs to implement the SupportedActions interface and extend the ActionsFactory class

Among the methods of the SupportedActions interface the role of some methods is not obvious and therefore described as follows:

associateAlarmsForHistoryNavigation

(Action action, Scenario scenario, Group group, Collection<Alarm> children, ProblemInterface problem)

This method is used to inform the NMS that all children alarms have to be grouped together under a problem alarm.

If HP TeMIP is the NMS, associateAlarmsForHistoryNavigation invokes the TeMIP directive GROUPALARMS.

If the NMS is a different product, possibly one dedicated method exists to group children alarms with a problem alarm, or possibly this is done through setting some alarms fields to be grouped.

dissociateAlarmsForHistoryNavigation

This method is the reverse of associateAlarmsForHistoryNavigation.

This method is used when the children alarms are not to be grouped any longer under the problem alarm of a given group.

setHistoryNavigation

(Action action, Scenario scenario, Alarm alarm, Qualifier qualifier)

This method sets the field of the alarm indicating whether the alarm is a sub-alarm, problem alarm, candidate alarm, or an orphan alarm.

Even if you your NMS does not require to update alarms with this information, such information must be stored in the working memory of Problem Detection.

An example Actions Factory for the MyCOolNMS NMS is described as follows:

```
public class MyCOolNMSActionsFactory extends ActionsFactory implements
                                               SupportedActions {


    @Override
    public Action createProblemAlarm(Action action, Scenario scenario, Group
group, ProblemInterface problem, Alarm alarm) throws Exception {

    String referenceAlarm = group.getTrigger().getIdentifier();
    action.addCommand("METHOD", "createProblemAlarm");   // for example only
    action.addCommand("REFERENCE_ALARM", referenceAlarm);  // for example
only

     [...]

     return action;
    }
```

The implementation of each method of the SupportedActions interface (`createProblemAlarm()` method in the above example) must fill the action to be sent to the NMS

For more details, see the Javadoc of the ActionRequest class:
[R7] Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Actions (C:\%UCA_EBC_DEV_HOME%\apidoc\uca-mediation-action-client\index.html)

The commands need to pe provided in the form of key/value pairs to the passed action object. The contents of the action and the specific commands to be provided depends on what the NMS expects.

Referencing and invoking Actions Factory

Assuming HP UCA EBC Problem Detection Value Pack is connected to two NMS systems: Smarts and SCOM, one Actions Factory is implemented for each NMS.

When an action needs to be sent, for example a Problem Alarm needs to be created, the Problem Detection framework will need to be informed which actions factory to use, and which NMS to target.

The ProblemXmlConfig.xml file of the Value Pack associates an action name and an action class to the action. An example of this file is as follows:

```
<ProblemPolicies xmlns="http://config.pd.vp.expert.uca.hp.com/">
  <mainPolicy>
       [ . . . ]
    <actions>
     <defaultActionScriptReference>Exec_localhost</defaultActionScriptRefere
nce>
      <action name="SMARTS">
        <actionReference>Smarts_Notif_localhost</actionReference>
        <actionClass>com.acme.af.SmartsActionsFactory</actionClass>
                      [ . . . ]
      </action>

      <action name="SCOM">
       <actionReference>SCOM_Alert_localhost</actionReference>
               <actionClass> com.acme.af.SCOMActionsFactory</actionClass>
            [ . . . ]
      </action>
    </actions>
    [ . . . ]
```

For a specific action to be performed on a specific alarm, the Actions Factory to invoke is found due to the method available in the ProblemDefault.java file (see the following example) or the Problem customization classes if defined there.

```
public SupportedActions chooseSupportedActions(Alarm alarm, ProblemInterface
problem)
[...]
          SupportedActions supportedActions =
getSupportedActions().get(alarm.getSourceIdentifier());
[...]
```

In the previous code snippet, the action name is taken from the "alarm.getSourceIdentier()"

In the previous example if value of the sourceIdentifier field of the alarm is SMARTS, the actions Factory containing an action called SMARTS (`<action name="SMARTS">`) is selected in the theProblemXmlConfig.xml file then the action class is `com.acme.af.SmartsActionsFactory` and the Action Reference is `Smarts_Notif_localhost`.

To identify which NMS to target, Problem Detection evaluates the contents of the ActionRegistry.xml file located at:
${UCA_EBC_INSTANCE}/conf/ActionRegistry.xml

The example content of this file is as follows:

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

<MediationValuePack MvpName="scom"
```

```
MvpVersion="1.0"
url=http://localhost:26700/uca/mediation/action/ActionService?WSDL
brokerURL="failover://tcp://localhost:10000">

<Action actionReference=" SCOM_Alert_localhost ">
<ServiceName>alertsDirective</ServiceName>
<NmsName>scom_host</NmsName>
</Action>
                    [...]
</MediationValuePack>

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

</ActionRegistryXML>
```

Trouble Ticket Actions Factory

To configure HP UCA EBC Problem Detection Value Pack sending actions to a Trouble Ticketing System, the following steps must be performed:

- Configure `ProblemXmlConfig.xml` located in the `src/main/resources/valuepack/conf/` directory in your development environment.
- Configure `${UCA_EBC_INSTANCE}/conf/ActionRegistry.xml`
- Implement a Trouble Ticket Actions Factory for your Trouble Ticketing System (if it is not HP TeMIP)
- Develop a Channel Adapter for your Trouble Ticketing System. This procedure is not covered in this document.

Configuring ProblemXmlConfig.xml

The `ProblemXmlConfig.xml` file associates a TroubleTicketAction name with the following:

- An actionReference that defines which Trouble Ticketing system to address.
- an actionClass that defines which implementation of the TroubleTicketActionsFactory is used.

The example content of this file is as follows:

```
<ProblemPolicies xmlns="http://config.pd.vp.expert.uca.hp.com/">
<mainPolicy>
[ . . . ]
<troubleTicketActions>
<troubleTicketAction name="TeMIP TT">
<actionReference>TeMIP_TT_Directives_localhost</actionReference>
<actionClass>com.hp.uca.expert.vp.pd.actions.TeMIPTroubleTicketActionsFactor
y</actionClass>
[ . . . ]
</troubleTicketAction>
</troubleTicketActions>
</mainPolicy>
```

By default, the alarm filters define the name of the TroubleTicketAction to be used for the alarm. This behavior can be overrided.

An example from the ProblemDefault.java file is found as follows. In this example the trouble ticket name is `tTActionsName`.

```
@Override
public SupportedTroubleTicketActions chooseSupportedTroubleTicketActions(
                                                                Alarm alarm,
ProblemInterface problem) throws Exception {

  Set<String> tags =
alarm.getPassingFiltersTags().get(problem.getProblemContext().getName());
  if (tags != null) {
    for (String tTActionsName : getSupportedTroubleTicketActions().keySet())
{
      if (tags.contains(tTActionsName)) {
        supportedTroubleTicketActions =
getSupportedTroubleTicketActions().get(tTActionsName);
      }
    }
  }
```

Configuring the ActionRegistry.xml

The action registry associates an actionReference with a Trouble Ticketing System name.

In the below example taken from the `ActionRegistry.xml` file this name is defined in the NmsName element.

```
<MediationValuePack MvpName="temip" MvpVersion="1.0"
url="http://localhost:18192/uca/mediation/action/ActionService?WSDL"
brokerURL="failover://tcp://localhost:10000">

[ . . . ]

<Action actionReference="TeMIP_TT_Directives_localhost">
<ServiceName>ttDirective</ServiceName>
<NmsName>localTeMIP</NmsName>
</Action>
</MediationValuePack>
```

Implementing a Trouble Ticket Actions Factory

If not HP TeMIP is the Trouble Ticketing System used, a Trouble Ticket Actions Factory needs to be created.

A Trouble Ticket Actions Factory implements the methods of the **SupportedTroubleTicketActions** interface.
See the JavaDoc for more details: [R6] *UCA for EBC Inference Machine – JavaDoc (%UCA_EBC_DEV_HOME%\apidoc\inference-machine\index.html)*

Example methods handled by this interface: `createTroubleTicket`, `closeTroubleTicket`.

The Trouble Ticket Actions Factory corresponding to the used Trouble Ticketing System must implement the SupportedTroubleTicketActions interface and extend the TroubleTicketActionsFactory abstract class containing common code

The following example shows an implementation extract of the `createTroubleTicket`() method:

```
public class MyTroubleTicketActionsFactory extends
TroubleTicketActionsFactory implements SupportedTroubleTicketActions {

  @Override
```

```
  public Action createTroubleTicket(Action action, Scenario scenario, Group
group, ProblemInterface problem, Alarm referenceAlarm, List<Alarm>
alarmsToAssociate) throws Exception {

    if (LOG.isTraceEnabled()) {
      LogHelper.enter(LOG, "createTroubleTicket()");
    }
    action.addCommand("DIRECTIVE_NAME", "CREATE_TICKET);
//
    action.addCommand("ENTITY_NAME", getTtServerEntity());
    action.addCommand("SELECTED_ALARM",
group.getProblemAlarm().getIdentifier());
```

The implementation of each method of the SupportedTroubleTicketActions interface (createTroubleTicket() method in the previous example) must fill the action to be sent to the Trouble Ticketing System.

For more details see the javadoc of the ActionRequest class:
[R7] Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Actions (C:\%UCA_EBC_DEV_HOME%\apidoc\uca-mediation-action-client\index.html)

The commands must be provided as key/value pairs, to the passed action object.

The content of the commands depends on what the Trouble Ticketing System Channel Adapter expects and supports.

Annex D.

Problem Detection Value Pack example with Events only

The Problem Detection Value Pack example with Events only is not available in IM SDK.

Annex E. Topology State Propagator Value Pack example

The TSP Value Pack is not available in IM SDK.

Annex F.

Topology State Propagator Advanced customization

As described in section 8.2, it is possible to modify the default behavior of Topology State Propagator Value Packs.

The following aspects of the behavior can be modified, similar to PD Value Packs:

- Per propagation
- Per family of propagations
- For all propagations
- For non propagation specific matters

The customization process is done similar to Problem Detection customization. See Annex C for details.

Annex G. Inference Machine Value Pack example

As part of the Inference Machine Development Kit, an example Value Pack project, named 'im-example', is available.

If deployed, the im-example Value Pack is able to recognize two problems with the Problem Detection scenario:

- Problem_SwitchDown
- Problem_PhoneUnavailable

It can also perform several propagations based on the above problems through the Topology State Propagator scenario:

- Propagation_Switch (generating Service Alarms)
- Propagation_Pool
- Propagation_Customer
- Propagation_VM
- Propagation_PhoneService (generating Service Alarms)
- Propagation_Server
- Propagation_Location
- Propagation_Service (generating Service Alarms)
- Propagation_Application
- Propagation_Shelf
- Propagation_CallServer

All of above problems and propagations have specific filters.

Problems generate Problem Alarms that are pushed to the TSP scenario.

Propagations are maintained in a hierarchy and only top-level ones create Service Alarms. The Problem and Service Alarms are stored on disk using DBActionsFactory.

It also contains sample tests file that can be run with the JUnit tool. These tests simulate the deployed behavior of the im-example Value Pack without having to actually deploy it. Alarms are injected into the Value Pack as though they came from the network.