hp Unified Correlation Analyzer

Unified Correlation Analyzer
for Event Based Correlation

Inference Machine

User Guide

Version 3.2

Edition: 1.0

April 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 Events grouping 18
2.3 Lifecycle 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 23
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 26
3.4.2 Lifecycle class for State and other Events 27
3.4.3 Interfaces 27

Chapter 4 28

The IM scenarios explained 28

4.1 Problem Detection 28
4.1.1 Its role in brief 28
4.1.2 Its main feature 28
4.1.3 Alarm state propagation 29
4.2 Topology State Propagator 30
4.2.1 Its role in brief 30
4.2.2 Its main feature 30

4.2.3 Alarm state propagation ... 31

Chapter 5 ... 32

Configuration ... 32

5.1 Value Pack ... 32
5.2 Inference Machine ... 32
5.2.1 Actions to NMS ... 33
5.2.2 Trouble Ticket Actions ... 36
5.3 Problem Detection ... 37
5.3.1 Filters, tags and mappers ... 37
5.3.2 Specific configuration ... 38
5.4 Topology State Propagator ... 45
5.4.1 Filters, tags and mappers ... 45
5.4.2 Specific configuration ... 46
5.5 Orchestra ... 50

Chapter 6 ... 51

Developing an IM Value Pack ... 51

6.1 Eclipse Plugins ... 51
6.1.1 Problem Detection only Value Pack ... 52
6.1.2 UCA EBC Topology State PropagatorTopology State Propagator only Value Pack 54
6.1.3 Inference Machine Value Pack ... 55
6.2 Understanding the Use Cases ... 55
6.3 Create a Simple PD VP ... 55
6.3.1 Analyze the problems to be detected ... 56
6.3.2 Identify the different types of alarms ... 56
6.3.3 Configure the Time Window ... 57
6.3.4 Create a Problem Alarm? ... 57
6.3.5 Create a Trouble Ticket? ... 58
6.3.6 Is the default behavior good enough? ... 58
6.3.7 Define the Filters ... 58
6.3.8 Configure Value Pack ... 62
6.3.9 Configure specific settings ... 63
6.3.10 Customize the behavior ... 63
6.4 Create a Simple TSP VP ... 64
6.4.1 Analyze the topology to be used and the propagations to be detected ... 64
6.4.2 Compute a State? ... 65
6.4.3 Identify the different types of alarms: Root Cause or Sub Alarms ... 65
6.4.4 Create a Service Alarm? ... 66
6.4.5 Create a Trouble Ticket? ... 66
6.4.6 Define the Filters ... 66
6.4.7 Configure Value Pack ... 70
6.4.8 Configure specific settings ... 72
6.5 Create a Standard IM VP ... 72

Chapter 7 ... 73

Advanced features of Problem Detection ... 73

7.1 The default behavior explained ... 73
7.1.1 Example ... 73

7.1.2 Alarm Role Check 75
7.1.3 EventRoleCheck 75
7.1.4 Problem Alarm Creation 75
7.1.5 Common Entity Check 76
7.1.6 Group update 78
7.1.7 Network State Update 79
7.1.8 Operator State Update 80
7.1.9 Problem State Update 82
7.1.10 Attribute Update 83
7.1.11 Periodic Check 84
7.1.12 Alarm eligibility update 85
7.1.13 Event eligibility update 86
7.1.14 Tags handling 87
7.2 Generic Events (other than Alarm types) are supported 89
7.3 Computing Problem Information starting V3.2 89
7.3.1 Case where Problem Detection is topology-aware 90
7.3.2 Default case (non-topology aware) 90
7.3.3 ProblemXmlConfig schema changes 90
7.3.4 ProblemDefault.computeProblemEntity(Event event) 91
7.3.5 GeneralBehaviourDefault.computeSourceUniqueId(Event event) 93
7.3.6 ProblemDefault.computeDbRecords(String dbUniqueIdReference, Event event) 93
7.3.7 ProblemDefault.computeGroupPriority(Event event) 94
7.3.8 ProblemDefault.computeTimeWindow(Event event) 95
7.4 How to customize default behavior 95
7.4.1 XML customization 95
7.4.2 Java customization 97
7.4.3 My ProblemDefault 102
7.4.4 Problems initialization starting V3.2 102
7.4.5 MyGeneralBehavior 107
7.4.6 Enrichment 108

Chapter 8 112

Advanced features of Topology State Propagator 112

8.1 The default behavior explained 112
8.1.1 Example 112
8.1.2 Event Role Check 115
8.1.3 State Creation 115
8.1.4 Service Alarm Creation and Clearance 115
8.1.5 Common Entity Check 115
8.1.6 PropagationGroup update 115
8.1.7 Network State Update 117
8.1.8 Operator State Update 118
8.1.9 Alarm Attribute Update 120
8.1.10 Periodic Check and General Behavior 121
8.1.11 Alarm eligibility update 122
8.1.12 State eligibility update 123
8.1.13 TroubleTicket update 123
8.2 How to customize default behavior 124
8.2.1 Java customization 124
8.2.2 My PropagationDefault 127

8.2.3 MyGeneralBehavior 131

Chapter 9 133

Troubleshooting 133

9.1 Logging 133

Chapter 10 136

Annexes 136

Annex A. 137

Migration steps from V3.1 to V3.2 137

Annex B. 142

PD Value Pack example 142

Annex C. 151

PD Advanced customization 151

Annex D. 163

PD Value Pack example with Events Only 163

Annex E. 164

TSP Value Pack example 164

Annex F. 165

TSP Advanced customization 165

Annex G. 166

IM Value Pack example 166

Tables

Table 1 - Software versions 10
Table 2 - Alarm state propagation from Problem Alarm to sub-alarms...29
Table 3 - Alarm state propagation from sub-alarms to Problem Alarm...29
Table 4 - IM actions configuration 33
Table 5 - IM action configuration 34
Table 6 – Specific optional IM action configuration for TeMIP 35
Table 7 – Specific optional IM action configuration for DB...36
Table 8 – IM troubleTicketActions configuration..36
Table 9 – IM troubleTicketAction configuration 37
Table 10 – Specific optional IM troubleTicketAction configuration for TeMIP 37
Table 11 – Tags for possible roles of an event within PD..38
Table 12 – Tags for possible roles of an alarm within PD 38
Table 13 – PD mainPolicy attributes...39
Table 14 –problemPolicy attributes 41
Table 15 – PD problemAlarm per-problem configuration...42
Table 16 – PD troubleTicket “per-problem” configuration 42
Table 17 – PD computeProblemEntityFromFields “per-problem” configuration...............................44
Table 18 – PD timeWindow “per-problem” configuration..44
Table 19 – PD customized “per-problem” configuration 44
Table 20 – Tags for possible roles of an alarm within TSP...45
Table 21 – TSP mainPolicy attributes..46
Table 22 – TSP serviceAlarm per-propagation configuration..47
Table 23 – TSP troubleTicket “per-propagation” configuration 48
Table 24 – TSP customized “per-propagation” configuration 50
Table 25 – PD: Possible roles for an alarm 61
Table 26 – TSP: Possible roles for an alarm 70
Table 27 - src/main/java: the customization code for the example Value Pack 143
Table 28 - src/test/java: the source code of the tests 145
Table 29 - src/main/resources: the configuration files of the example Value Pack..........................146
Table 30 - src/test/resources: the tests configuration files 148

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 (PropagationGroup) : position of Events 19
Figure 5 Group already created: example 19
Figure 6 PropagationGroup already created: example 19
Figure 7 Group to be created: example 20
Figure 8 PropagationGroup to be created is empty. 20
Figure 9 – Problem Detection solution architecture 25
Figure 10 – Topology State Propagator solution architecture 26
Figure 11 - Explanation of the candidateVisibilityTimeMode=Max 40
Figure 12 IM Orchestra configuration example 50
Figure 13 - How to create a UCA EBC project in Eclipse 51
Figure 14 – Create PD only Value Pack 52
Figure 15 - Files to edit to configure MyFirstProblemDetectionValuePack 53
Figure 16 – Create TSP only Value Pack 54
Figure 17 – Create IM Value Pack 55
Figure 18 - Time window illustration 57
Figure 19 Alarm clearance sequence diagram example 74
Figure 20 PD: Alarm clearance example: PD group updates Step1 74
Figure 21 PD: Alarm clearance example: PD group updates Step2 75
Figure 22 - One problem specific customization 97
Figure 23 - Consolidation of alarm's qualifiers 102
Figure 24 - MyProblemDefault: a customization for a group of problems 102
Figure 25 – PD MyGeneralBehavior name matching 107
Figure 26 Alarm termination sequence diagram example 113
Figure 27 Topology of the example 113
Figure 28 TSP: Alarm termination example: TSP group updates Step1 114
Figure 29 TSP: Alarm termination example: TSP group updates Step2 114
Figure 30 - One propagation specific customization 124
Figure 31 - MyPropagationDefault: a customization for a group of propagations 127
Figure 32 – TSP MyGeneralBehavior name matching 132
Figure 33 - schema of implementation of the main Problem Detection interfaces 152

Preface

The intention of this document is to provide information about HP UCA for EBC Inference Machine.

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

Product Version: 3.2

Kit Version: V3.2

Intended Audience

The intended audience of this guide is primarily developers (customers or HP consultants) wanting to understand an UCA for EBC Inference Machine Value Pack containing Problem Detection and Topology State Propagator scenarios.
This document will also be interesting for anyone wanting to know more about Inference Machine features

Prerequisites

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

The reader is advised to consult Chapter 1 & 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.
- Pathnames
- Keyboard key names

Italic Text:

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

Bold Text:

- To introduce new terms and to emphasize important words.

Associated Documents

The following documents contain useful reference information:

References

[R1] *Unified Correlation Analyzer for Event Based Correlation 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 (C:\%UCA_EBC_DEV_HOME%\apidoc\inference-machine\index.html)*

[R7] *UCA for EBC – JavaDoc (C:\%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*

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 3.2	• HP-UX 11.31 for Itanium • Red Hat Enterprise Linux Server release 5.9 & 6.5
UCA for Event Based Correlation Channel Adapter Version 3.2	• HP-UX 11.31 for Itanium • Red Hat Enterprise Linux Server release 5.9 & 6.5
UCA for Event Based Correlation Software Development Kit Version 3.2	• Windows XP / Vista • Windows Server 2007 • Windows 7 • Red Hat Enterprise Linux Server release 5.9 & 6.5
UCA for Event Based Correlation Inference Machine Software Development Kit Version 3.2	• Windows XP / Vista • Windows Server 2007 • Windows 7 • Red Hat Enterprise Linux Server release 5.9 & 6.5

Table 1 - Software versions

Support

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

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

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

Chapter 1

Inference Machine: a quick tour

1.1 Context

Okay, UCA for EBC is an expert system, as it brings already an embedded Inference Engine and the end-user can provide its own knowledge base of rules to execute.

So what is it that **Inference Machine**?

In fact, UCA EBC Inference Machine is nothing more than a framework based on top of UCA for EBC to deliver high-value Value Packs with embedded knowledge base where end-user does no more 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 SDK brought by Inference Machine is aimed at building an UCA for EBC Value Pack for the Root Cause Analysis / Service Impact Analysis pattern for all kinds of network elements.

1.2 Naming disambiguation

The name "*Inference Machine*" has different meanings in different contexts. It can be a short name for

- Inference Machine Development Kit (aka 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 UCA EBC Development Kit.
- Inference Machine Value Pack (aka IM VP):
  an UCA EBC Value Pack built using the Inference Machine Development Kit and including its libraries.

The name "*Problem Detection*" has different meanings in different contexts. It can be a short name for

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

The name "*Topology State Propagator*" has different meanings in different contexts. It can be a short name for

- Topology State Propagator Framework (aka TSP Framework):
  the set of libraries, rules, configuration files, used to develop and run a Topology State Propagator Value Pack. This framework is delivered as part of the UCA EBC Inference Machine Development Kit and packaged into any Topology State Propagator Value Pack.
- Topology State Propagator Value Pack (aka 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 of cases, a correlation engine is needed to provide root cause or/and service impact analysis.

Within UCA EBC family:

- RCA is covered by **Problem Detection** product (short named as **PD**).
- SIA is covered by **Topology State Propagator** product (short named as **TSP**).
- The conjunction of both RCA and SIA is called the **Inference Machine** (short named as **IM**). An IM Value Pack follows the RCA-SIA pattern as shown in Figure 1.

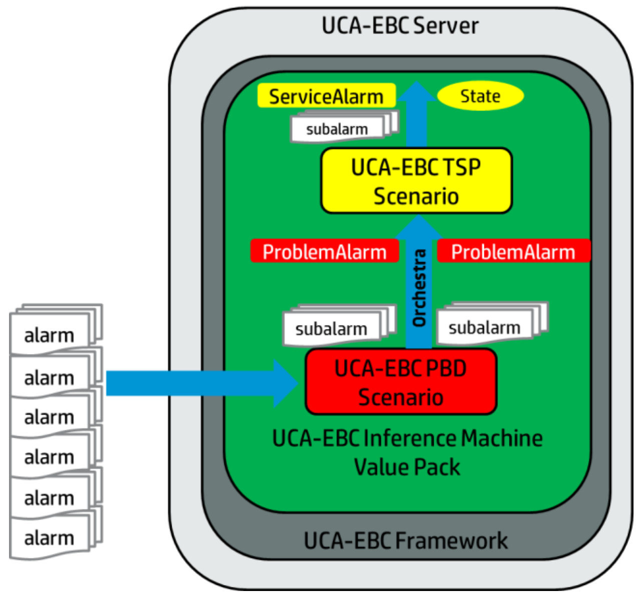


Figure 1 Inference Machine Value Pack (RCA-SIA pattern)

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 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 (*e.g.* Trouble Ticket generation, Alarms clearance), as well as cross-domain correlation and alarms enrichment.

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

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 alarms set and on the presence of certain conditions. Then, an operator readable Problem Alarm that summarizes the problem will be generated.

1.3.2.2 Problem Alarm

Another base feature of Problem Detection Value Packs is to hide all the sub-alarms in the NMS (Network Management System) display under the problem 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. 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 resource at some time *Tpb* (*Tpb* denotes time when the problem occurred), 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 those 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:

- 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 through creation of a State representing the propagation and, optionally, the creation of 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 (*e.g.* Trouble Ticket generation, Alarms clearance), as well as cross-domain correlation and alarms enrichment.

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

In a standard way, one TSP scenario will be associated to 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 trigger by either:

- a Root Cause event (usually a Problem Alarm coming from PD)
- Another state generated by TSP (*e.g.* a state generated for a sub-service).

The propagation is responsible for creating the state and optionally storing it into a DB, thanks to the 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 (or POI)

The Topology POI is an information utility feature brought by UCA-EBC V3.1. It is used in the UCA GUI graph-display tool to check what's happening 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 was 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 ServiceAlarm is an Alarm that can be created by TSP in a NMS, in order to identify the impacted propagation. It follows the same concerns as the ProblemAlarm used in PD.

As PD manages the Problem Alarm, in TSP, when the Service Alarm feature is enabled, a similar treatment happens. TSP can hide all the sub-alarms in the NMS (Network Management System) 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:

- 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 an umbrella 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 resource at some time T_{pb} (T_{pb} denotes time when the problem occurred), equipment in the neighborhood of that resource, usually generate several alarms in a time window around T_{pb}.

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

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

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

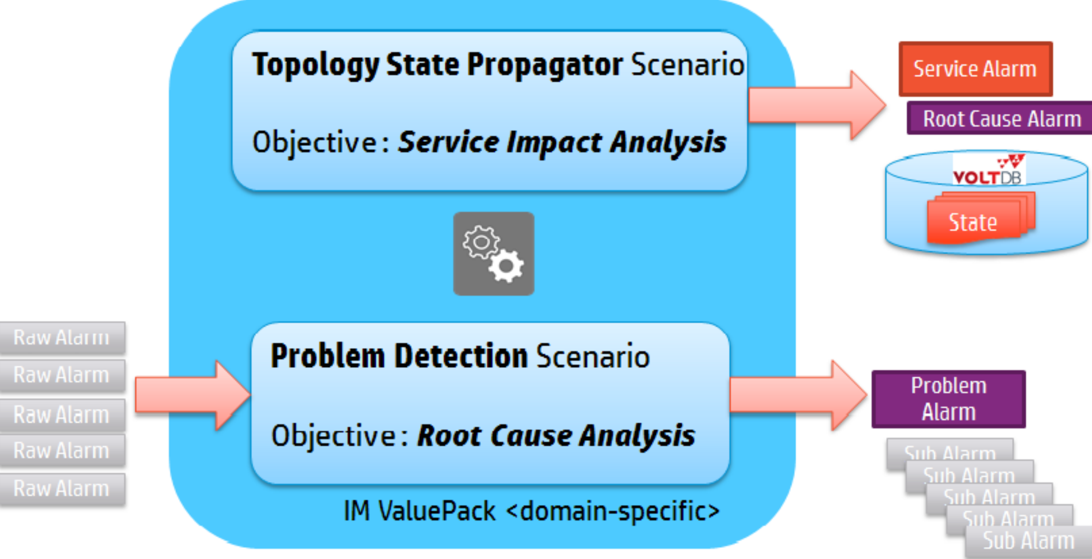


Figure 2 – RCA-SIA Pattern

When the same Network Management System (NMS) is used to handle Problem Alarms (generated by PD) and Service Alarms (generated by TSP), those 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. Refer to [R10] *HP Unified OSS Console Version 1.2.0 – User Guide* for more information.

2.2 Events grouping

A base feature of the Inference Machine Value Packs is that with both Problem Detection and with Topology State Propagator event grouping is possible under a summarized alarm which will represent 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, it is the State which represents the grouping internally in TSP.

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

To familiarize with these concepts, several schemas and diagrams are presented in this document. The following notations are respected as presented in Figure 3.

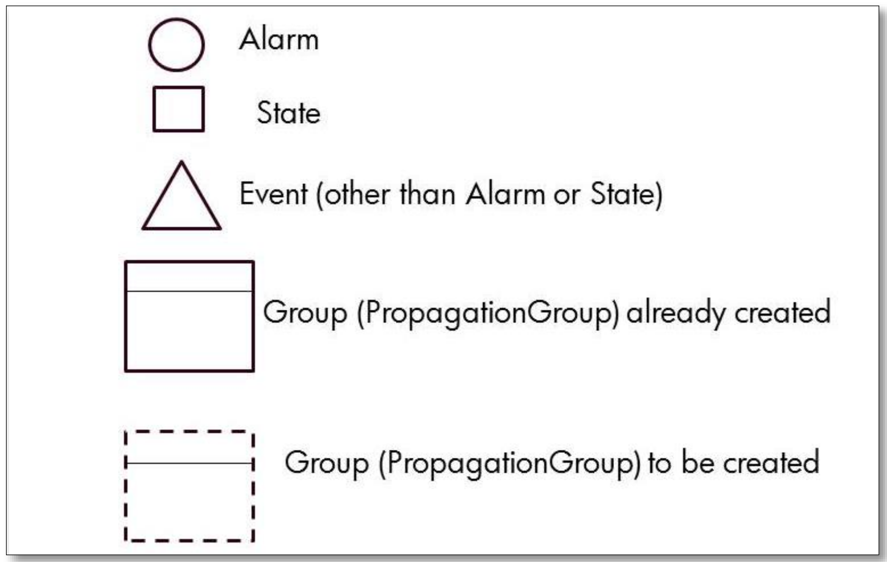


Figure 3 Notation conventions

Depending on their position in the Group and Propagation Group:

- a State can be the State representing the PropagationGroup or on impacting State of it
- an Alarm can be the ProblemAlarm of a Group, the ServiceAlarm of a PropagationGroup, or a SubAlarm in the case of Group and Propagation Group, and a RootCauseAlarm in the case of a PropagationGroup
- an Event can be a triggerEvent or a subEvent of Group.

These concepts are explained in the following figures.

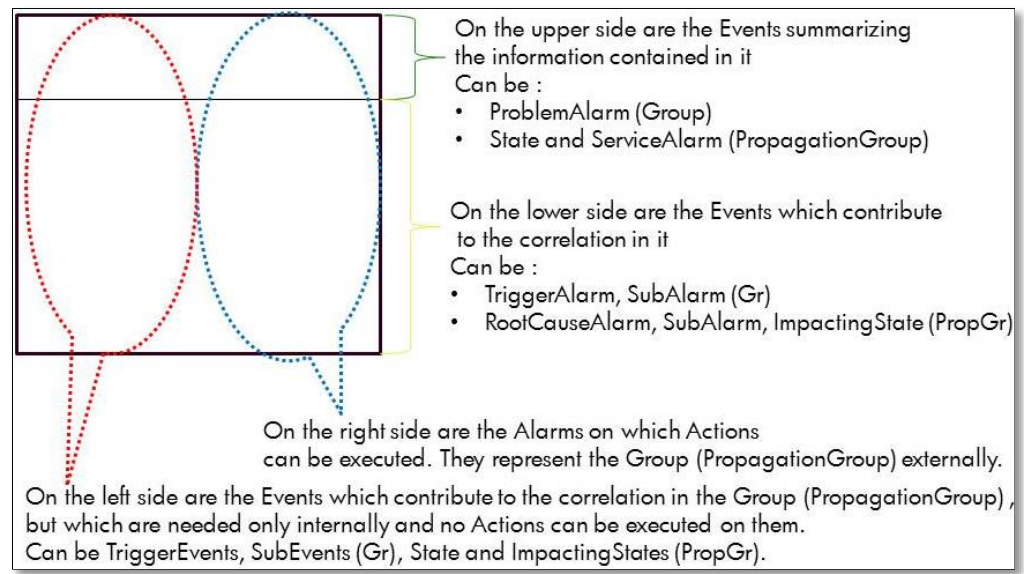


Figure 4 Group (PropagationGroup) : position of Events

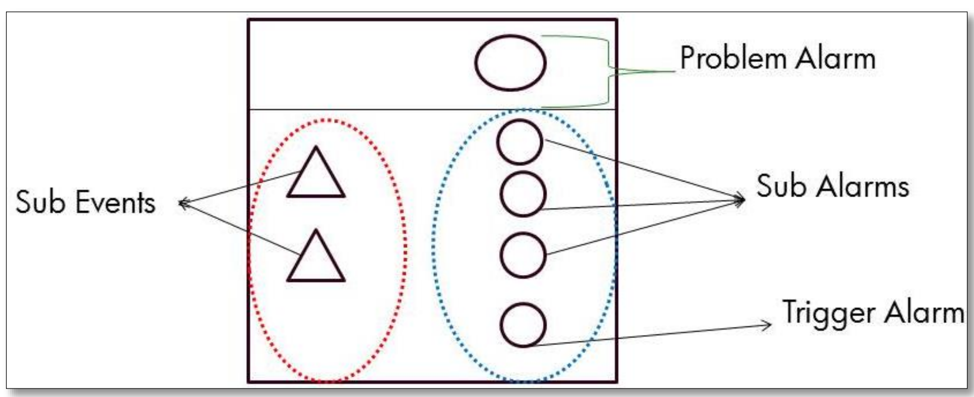


Figure 5 Group already created: example

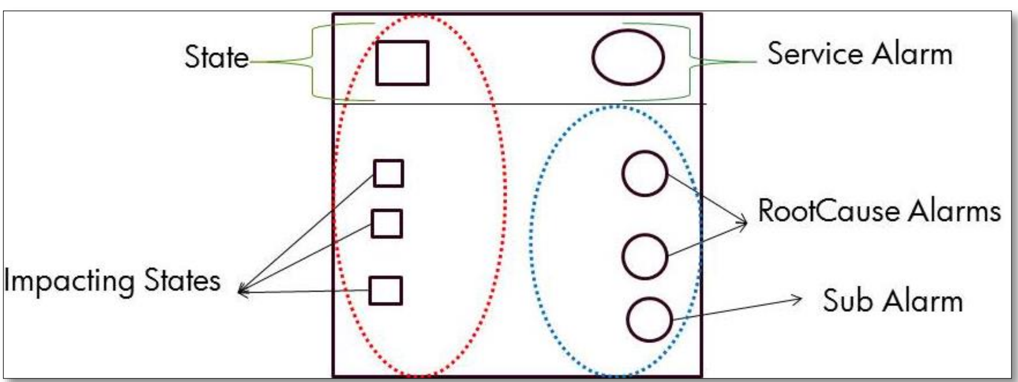


Figure 6 PropagationGroup already created: example

Depending if the Group is created or not:

- an Alarm can be a CandidateAlarm of a Group
- An Event can be a CandidateEvent of a Group.

In Figure 7 is presented 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.

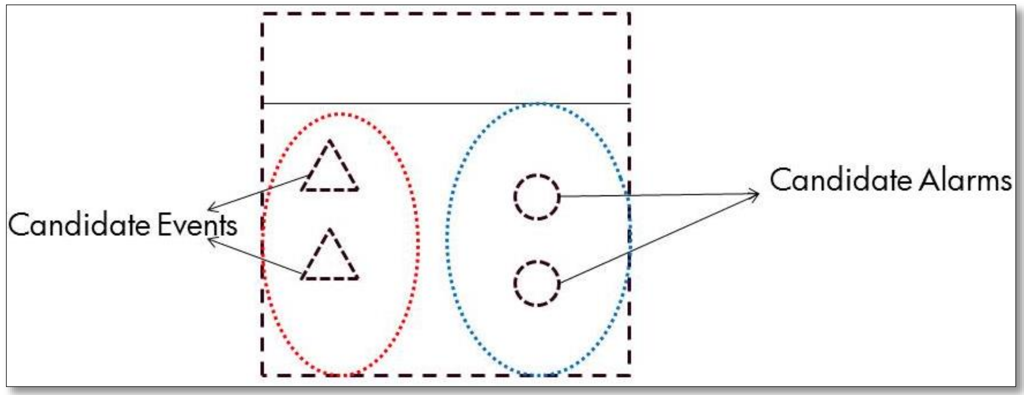


Figure 7 Group to be created: example

In comparison with the Group, in the PropagationGroup there is no notion of CandidateEvent not CandidateAlarm. Therefore, the PropagationGroup 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 PropagationGroup to be created will be empty.

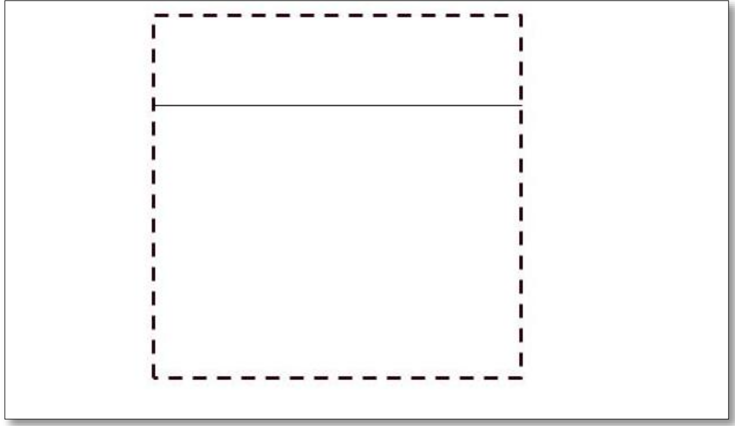


Figure 8 PropagationGroup to be created is empty.

In brief:

- PD scenario can hide all the sub-alarms in the NMS (Network Management System) display under the problem alarm.
  Since V3.2, PD is also able to group Events (not necessarily Alarms).
  You can refer to annex D to get more details on this new functionality.
- TSP scenario can aggregate State and/or Root Cause Alarms that impact the same service under a same group.
  TSP is able to group Root Cause Alarms in the NMS (Network Management System) display under a single Service alarm, if the NMS used is the same of course.

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.

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

2.3 Lifecycle

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

In case the default behavior is needed to be enhanced, the Value Pack developer can do so by writing his custom code in overridable methods or through configuration when available.

Which overridable methods will be called depend on the lifecycle of the alarm, state or other event and depending on the problem or propagation contexts.

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

2.4 Automatic actions

Besides noticing and reporting the appearance of a failure (problem), besides grouping events, Inference Machine scenarios can execute other automatic actions with respect to the lifecycle of alarms (Alarm state propagation from Problem or Service Alarm to sub-alarms and vice versa) 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 the following section.

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 could be done independently

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

2.6 Cross domain correlation

PD scenarios, as all UCA for EBC Value Packs, are able to process alarms coming from various NMS (Network Management Systems) through the OSS Open Mediation layer. The same applies to TSP scenarios, which by providing the SIA function comes to complete in the IM the RCA-SIA pattern; therefore the standard IM Valuepack will contain one PD scenario which will usually send 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 TeMIP, and are able to interact with the HP Service Manager Trouble Ticketing system through TeMIP.

Since UCA-EBC has been designed as an independent platform it is equally capable of receiving alarms and sending actions to other third party Network Management Systems and Trouble Ticketing / Incident Management Systems. By implication this applies to the PD and TSP frameworks too since they are layered on top of the 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 TeMIP)
- Any Trouble Ticketing System (in addition to HP Service Manager)

The support of additional Network Management Systems and Trouble Ticketing system will be done through 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 will be 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 NMS (Network Management System) do not contain enough information to be correlated, both 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 also possible to write Java code doing any imaginable synchronous or asynchronous request (database access, file access, HTTP request …).

2.8 Performance

Compared to a standard UCA for EBC Value Pack that would have been developed to perform correlation, an Inference Machine Value Pack is very likely to perform significantly better. The reason is that the Inference Machine Framework uses optimization based on several hash maps, which allow 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 times are close to being a linear function of the number of alarms, whereas in the case of regular 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 that it brings good performance and a sound behavior (predictable results).

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

However, an important size of memory for the JVM should 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 relatively simple and short.

If you're satisfied with the default behavior of PD or TSP scenarios, the creation of an IM Value Pack will not require any java coding or rule writing. It will only require modifying a few XML configuration files.

2.11 Simulation

It is possible and even quite easy 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. In any case, 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

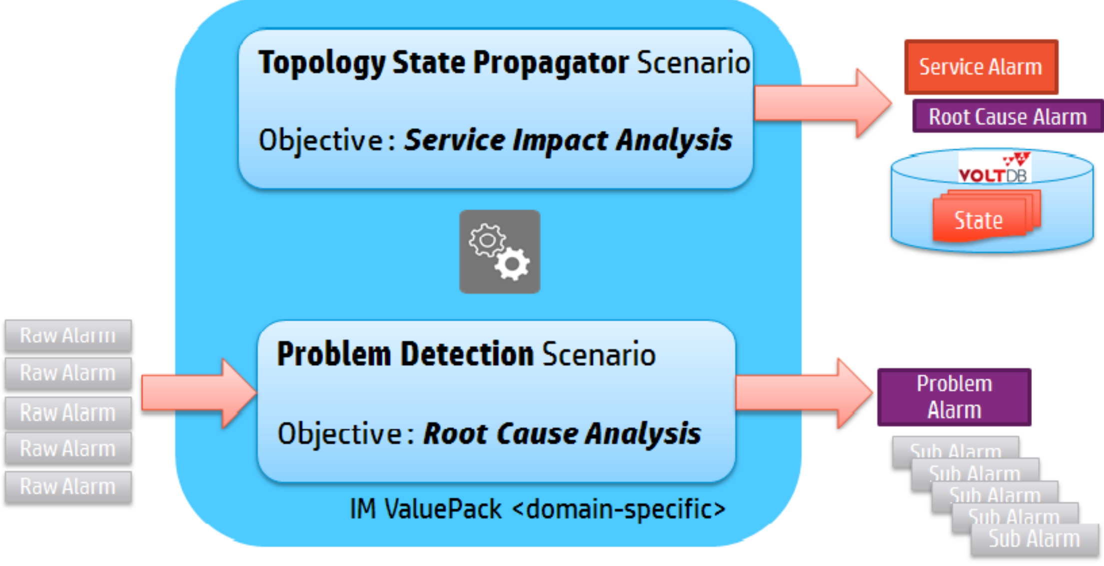
Given that UCA-EBC brings a correlation engine based on incoming events, this capability is most of the time not sufficient because end-users want to do their events analysis using an out of the box element.

Inference Machine is the cornerstone for achieving this capability. It brings a RCA-SIA pattern that should fit any customer needs. Users of Inference Machine do no more need to write correlation rules, but they need to provide configuration files and/or few customizations of Java classes.

Inference Machine is composed of 2 scenarios running in an UCA-EBC server

- 1st one is Problem Detection (PD) for doing Root Cause Analysis
- 2nd one is Topology State Propagator (TSP) for doing Service Impact Analysis

The flow of raw alarms are coming from any source (usually NMS) are treated by PD to group them and to generate correlated Root Cause Alarms. Those RCAs are eventually forwarded to TSP which in turns group them to find out what are the impacts of the network topology and eventually generate Service Alarms.



3.2 Problem Detection

The diagram below shows a Problem Detection Value Pack deployed on a UCA for EBC Server, with OSS Open Mediation connected to UCA EBC. Several Network Management Systems are connected to OSS Open Mediation.

The PD Scenario receives its alarms through Alarm Collection flow coming from one or several of the Network Management Systems connected to OSS Open Mediation. It can also receive directly alarms that come from other scenarios through Orchestra.

The Actions (to create Problem Alarms, to group sub-alarms under the Problem Alarm, etc. …) use Action Service and are routed to OSS Open Mediation to be processed by the proper Network Management System.

Contrary to other UCA for EBC Value Packs, a PD Scenario does not allow its developer to modify the set of rules as they are embedded into 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 9 – Problem Detection solution architecture.

The filters can, as per any other UCA-EBC VP, be tuned directly by end-user. Please refer to UCA-EBC Reference Guide [R1].

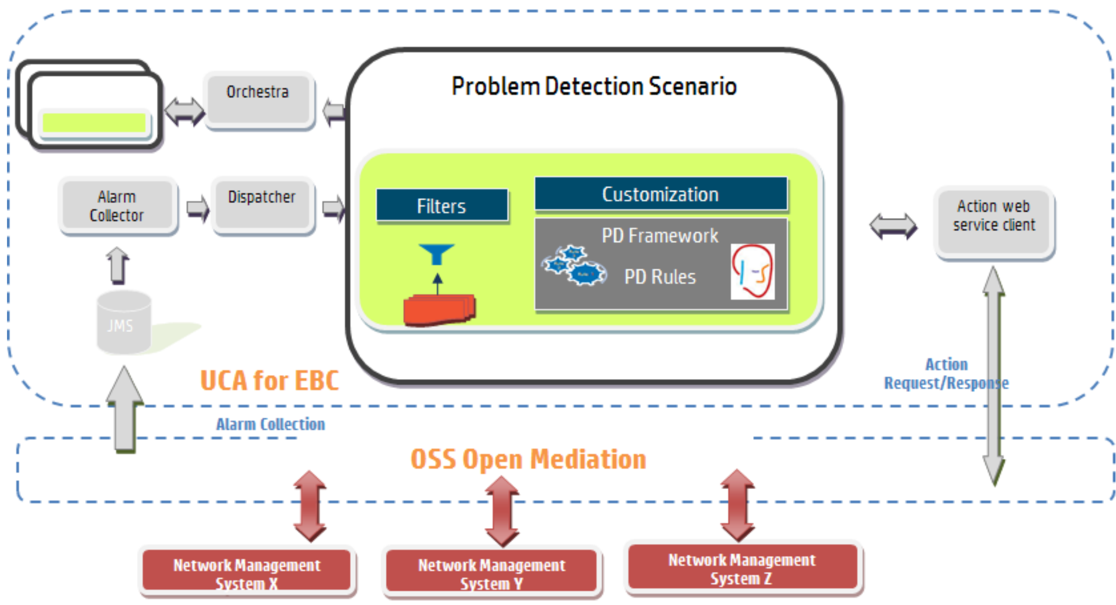


Figure 9 – Problem Detection solution architecture

3.3 Topology State Propagator

The diagram below shows a Topology State Propagator (TSP) Value Pack deployed on a UCA for EBC Server, with OSS Open Mediation connected to UCA EBC. Several Network Management Systems are connected to OSS Open Mediation.

The TSP scenario receives its alarms directly from other scenarios (PD like) through UCA EBC Orchestra. However, it can also receive alarms through UCA EBC Alarm Collection flow coming from one or several of the Network Management Systems connected to OSS Open Mediation.

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

The Actions (to create Service Alarms, to group sub-alarms under the Service Alarm, etc. ...) use UCA EBC Action Service and are routed to OSS Open Mediation to be processed by the proper Network Management System.

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

However,

- TSP Framework provides a set of Java methods that the developer can use to control the life cycle of specific Events, i.e. Alarms and States, the Service Alarm creation and so on. This is called Customization in Figure 10 – Topology State Propagator solution architecture.

The filters for defining the Propagations can be tuned directly by end-user. Please refer to UCA-EBC Reference Guide [R1].

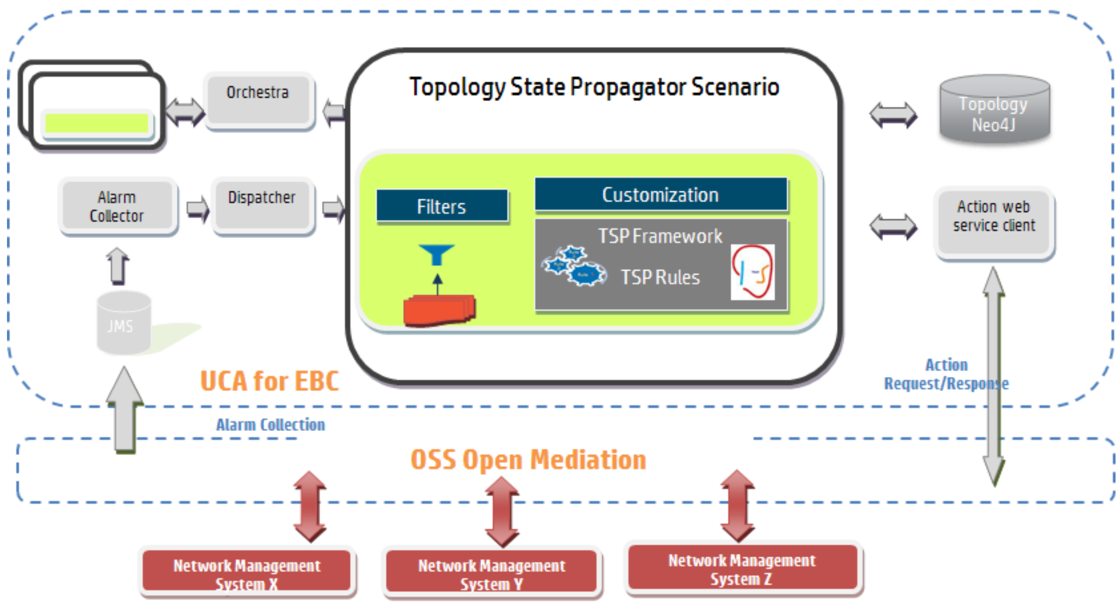


Figure 10 – Topology State Propagator solution architecture

3.4 A common library

As PD and TSP have several common needs, it has been decided to provide a common library, which is delivering its own namespace.

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

3.4.1 Actions Factory

Both TSP and PD have the needs to execute actions on NMS (*e.g.* create alarm, clear alarm, group alarms, etc...). Therefore, the Actions Factory is provided as part of

the ***uca-evp-im-common.jar*** common library. The same applied for the access to the database (DbActionsFactory.class 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 3.2 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 file. Refer to Annex A How do I migrate my PD VP 3.0/3.1 to 3.2? for more information.

In counterpart to PD incompatibility, several improvements are present:

- the logic of Actions is separated from PD and TSP
- reusable (same ActionsFactory or DbActionsFactory can be used across PD and TSP)
- easier to understand

3.4.2 Lifecycle class for State and other Events

The class *com.hp.uca.expert.vp.common.lifecycle.MixEventsAndStateLifeCycleExtended.class* has been added in the ***uca-evp-im-common.jar*** common library. This class is an enriched Alarm Lifecycle class, managing both States and others Events (Alarms and other events) lifecycle. Alarms passing just the top filter "*ReservedForGeneralBehavior*" will not be inserted in the Working Memory.

For the Topology State Propagator scenario, as well as for the PD scenario, in the IM Value Pack, there are two new classes extending this common class:

- The *com.hp.uca.expert.vp.pd.im.lifecycle.InferenceMachineLifeCycleExtended* is used as the Problem Detection scenario extended life cycle in an Inference Machine valuepack. This class handles alarms, events and states lifecycle and it will bypass service alarms received from the network.
- The *com.hp.uca.expert.vp.tp.im.lifecycle.InferenceMachineLifeCycleExtended* is used as the Topology State Propagator scenario extended life cycle in an Inference Machine valuepack. This class handles alarms, events and states lifecycle.

An IM VP example is described in Annex F.

3.4.3 Interfaces

Several interfaces are contained in the common library needed 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 IM Javadoc part of the SDK [R6]. Most of above interfaces have a default implementation which is used implicitly used by ProblemDefault or PropagationDefault Java classes.

Chapter 4

The IM scenarios explained

Inference Machine Framework brings two scenarios:

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

4.1 Problem Detection

4.1.1 Its role in brief

In short, Problem Detection is doing Root Cause Analysis.

Problem Detection aims at:

- Detecting from a numerous number of raw alarm a set of symptom alarms, and identifying the problem that those 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 Problem Detection 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 feature

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

This is to identify that a failure (problem) has occurred based on the appearance of a certain set of alarm, and on the presence of certain conditions;

And then to generate 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 has been 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"

When Problem alarm is	Change sub-alarms' state to
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 Working Memory or to remain in 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 please refer to the chapter **alarmEligibilityPolicy** in the UCA for EBC Reference Guide

When the state of all sub-alarms has been 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

4.2.1 Its role in brief

In short, Topology State Propagator is doing Service Impact Analysis.

In Topology State Propagator world:

- We call Propagation 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.

- We call State the status of that impact in the topology. For example, a service is degraded but can have different levels of degradation (low, medium, high…)

Topology State Propagator aims at:

- Detecting from one or more root cause alarm a set of propagations, and identifying the impacts that those 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 to 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 Topology State Propagator 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 alarms.

4.2.2 Its main feature

The primary role of a Topology State Propagator (TSP) scenario is Propagation Impact.

This is to identify what are the impacted services (propagation) based on the appearance of certain root cause alarms (previously correlated as Problem Alarms by PD) and based on the description of the network impacted (though Topology API).

And then to generate a State defining the status of that impacted service at a given time, and optionally to:

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

4.2.3 Alarm state propagation

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

Chapter 5

Configuration

This chapter covers the configuration of Inference Machine.

5.1 Value Pack

An UCA-EBC VP comes always with 2 configuration files.

An IM VP should provide those files already configured for running correctly.

The file "*ValuePackConfiguration.xml*" defining the configuration used by any UCA-EBC Value Pack that is 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>*	Those sections **may be modified** to support different NMS or DBs that should be considered as sources for the IM VP.

The file "*context.xml*" defining the spring beans to instantiate within the IM VP is closely related to the IM VP code itself. Particularly, it contains the beans:

"*problemsFactory*"	Present if the PD scenario is defined in "*ValuePackConfiguration.xml*" file. It **should not be modified**
"propagationsFactory"	Present if the TSP scenario is defined in "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 one **may be modified** to satisfy your DB connection needs.

For full details of above files, please refer to UCA-EBC Reference Guide [R1].

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 IM framework.

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

5.2.1 Actions to NMS

The IM Framework comes by default with the support of two Actions Factories, both of which come with default Alarm directives for handling alarms:

- in **TeMIP**, in that case <*actionClass*> should be set to *com.hp.uca.expert.vp.common.actions.temip.TeMIPActionsFactory*
- in a **DB**, and 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	Unique reference that will be used in the rule to define the routing information of a script-based Action.
action	property	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	Unique reference that will be 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. Methods such as `createAlarm, terminateAlarm, clearAlarm, …`
attributeUsedForKeyDuringRecognition	property	The Custom Field Name of the Alarm that will contain the information to identify that an Alarm is generated by the IM Framework. In other words this attribute defines the name of the field (in UCA-EBC format) of the Problem Alarm (or Service Alarm), that PD (or TSP) has to look at (when the alarms come back from the NMS) to find the useful info to attach this alarm to the right group

name	type	value
attributeUsedForKeyPbAlarmCreation	property	The custom Field of the Alarm that will contain information about the problem. This attribute defines the name of the field in the NMS format) of the Problem Alarm (or Service Alarm), in which PD (or TSP) puts the useful info (at the time of creation of that alarm) that it will read when that alarm comes back from the NMS. The useful info it contains are things like: name of the trigger alarm, name of the problem/propagation, name of the problem/propagation entity.
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 5 - IM action configuration

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

name	type	Value
maxChildrenLength	long property	Maximum size in Bytes of the alarm field "children" Default size is 15000 (15 Kb) Once the maximum size of the "children" field is reached, Problem Detection stops requesting the NMS to add potential new children to the parent alarm
useOnlyGroupingKeys	boolean property	If set to true (default false), the GROUPALARM directive is not used. This implies that "parent" and "children" field of alarms won't be filled. Only the field "grouping Keys" will be filled ; and the navigation in the TeMIP client will only be 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).

name	type	Value
copyReferenceAlarmWhenNotPbAlarm	boolean property	Useless 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	Defines 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 TeMIP

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

name	type	Value
useOnlyGroupingKeys	boolean property	If set to true (default false), the parent and children fields of an alarm are not updated. Only the field "groupingKey" will be 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 will be used.

name	type	Value
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 TeMIP.

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

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

name	type	value
troubleTicketAction	property	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	Alarms corresponding (in the filters file) to a tag matching this name will use the trouble ticket system defined in the actionReference below
actionReference	property	Unique reference that will be 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. Methods such as createTroubleTicket, closeTroubleTicket…
booleans	property (optional)	For defining multiple booleans for a specific use-case.
strings	property (optional)	For defining multiple strings for a specific use-case. Container for a set of key / value <string> specifying parameters for the interaction with the trouble ticketing system

longs	property (optional)	For defining 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 TeMIP

5.3 Problem Detection

5.3.1 Filters, tags and mappers

A PD Scenario comes usually with 3 standard UCA-EBC configuration files:
- “*ProblemDetection_filters.xml*” to define the problems and their tags
- “*ProblemDetection_filtersTags.xml*” to define the tags associated to the filters
- “*ProblemDetection_mappers.xml*” to define the different mappers and the neo4j Cypher queries to use within 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, since it defines the Java classes corresponding to a specific problem. Hence, it should not be modified except in some rare conditions, where for example some problem priority needs to be re-assessed, or to use a new mapper for computing the unique source id of an incoming event, or to update the role of a specific filtered alarm.

The “*ProblemDetection_filtersTags.xml*” is only used for GUI purpose in the filter builder panel to associate right tags to right filters.

The PD framework recognizes few predefined tags, as followed:

- When concerning Event objects

When	The role of the event is	And the definition of this role is
tag=“TriggerEvent”	Trigger event	Event which is important symptom of a problem and which triggers the creation of a group
tag=“SubEvent”	Sub event	Event which is a symptom of a problem and is grouped under a Problem alarm

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

- When concerning Alarm objects

When	The role of the alarm is	And the definition of this role is
tag=“Trigger”	Trigger alarm	Alarm which is an important symptom of a problem, and which triggers the creation of a problem alarm
tag=“SubAlarm”	Sub-alarm	Alarm which is a symptom of a problem and is grouped under a Problem alarm
tag=“ProblemAlarm”	Problem alarm	Alarm that summarizes the problem, and is readable by the operator

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

You can also associate above tags, for example:

- tag="SubAlarm,ProblemAlarm" → defines an alarm which is Problem alarm of a problem, and sub-alarm of another problem
- tag="Trigger,ProblemAlarm" →the trigger alarm should be considered as Problem alarm (no new alarm created)

The <***cypherQuery***> elements defined in the “*ProblemDetection_mappers.xml*” file are closely related to the topology loaded in Neo4j, hence it **should not be modified** except in some rare conditions. However, the <***mapper***> elements **may be changed** to handle new conditions on incoming events, but in such a case, the “*ProblemDetection_filtersTags.xml*” should be updated accordingly.

5.3.2 Specific configuration

A PD Scenario comes with a specific “*ProblemXmlConfig.xml*” file.

5.3.2.1 The Main Policy

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

It has few attributes:

name	type	value
enablePrioritySort	boolean attribute	Specifies to turn on the group sorting feature. Default value is “false”

multipleParentSupport	boolean attribute	Specifies to set the parent relationship for each group of that Problem Alarm (true) or only with the one of highest priority (false). Default value is “true”
enableTopoAccess	boolean attribute	Specifies to access topology information when computing information for Problem Alarm (hence triggering computeSourceUniqueID() and computeDBRecords()) during the workflow (true) or not (false) Default value is “false”

Table 13 – PD mainPolicy attributes

And few elements described below:

<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. Once the problem is detected (i.e. 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 types of problems. So it will remain as a candidate alarm for as long as there is a possibility that this alarm become part of one of the problems (problem A or problem B or problem C). To be part of a problem instance, an alarm must be included in a time window (see Figure 11) around the time of appearance of a trigger alarm for that problem. In diagram below if none of the trigger alarms for problem A, B and C came, it is useless for the alarm to remain candidate longer than the max value of `timeWindowBeforeTrigger` of problems A, B and C. If a trigger alarm comes after, then the alarm will necessarily be out of its time window.

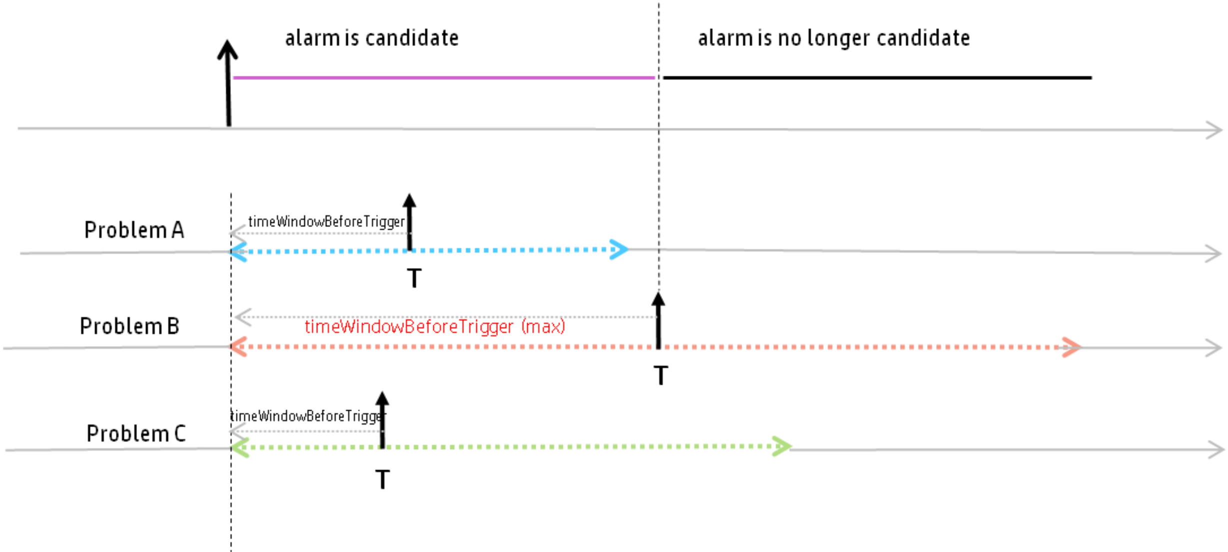


Figure 11 - Explanation of the candidateVisibilityTimeMode=Max

candidateVisibilityTimeMode=Value means that the alarm will remain as a candidate alarm no longer than the value specified by *candidateVisibilityTimeValue* (expressed in milliseconds)

candidateVisibilityTimeMode=Min means 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 should be 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 sometimes, some alarms disappear by themselves after some time; so in such situation it can be useful for a PD Value Pack to wait a little and see which alarms still exist.

When enabled with *transientFilteringEnabled=true*, the Transient Filtering feature makes the Problem Detection Value Pack, upon reception of any alarm, wait during a period (*transientFilteringDelay*) before actually processing the alarm. Maybe the alarm will have disappeared.

```
transientFilteringEnabled=true|false
```

```
transientFilteringDelay=<waiting period in milliseconds>
```

<actions>

The PD Framework is able to configure multiple actions factories in order to support multiple NMS. Refer to section 5.2.1 to get the details.

< troubleTicketActions>

The PD Framework is able to configure trouble ticket actions factories. Refer to section 5.2.2 to get the details.

Note that this element is however optional.

< counterTotalNumberAlarms>

It specifies what to count for the Problem Alarm field representing 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 The Problem Specific Policies

Problem Policies are configuration settings which are 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 few attributes:

name	type	value
enableComputeProblemEntityFromMappers	boolean attribute	Specifies to disable (when false) the use of calling mappers in computeProblemEntity(). Default value is “true”
enableComputeProblemEntityFromFields	boolean attribute	Specifies to enable (when true) calculation of fields key/value pairs in computeProblemEntity(). Default value is “false”

Table 14 –problemPolicy attributes

It has also elements described below:

<problemAlarm>

The <*problemAlarm*> element specifies behavior around ProblemAlarm.

name	type	value
delayForProblemAlarmCreation	long (optional)	Delay, expressed in milliseconds, before the creation of the associated problem alarm. Example: Setting the value: 2000 to this property applies a delay of 2000 ms (2 seconds) before creating Problem Alarms. Default value is 2000.
delayForProblemAlarmClearance	long (optional)	Delay, expressed in milliseconds, before clearing the problem alarm. Example: Setting the value: 0 (ms) to this property does not delay the clearance of Problem Alarms after all conditions are met for clearing problem Alarms. Default value is 10000.
problemAlarmCanTriggerAnotherGroupForSameProblem	boolean (optional)	It now possible to support the concept of nested problems, i.e. one alarm may have multiple roles for the same problem. It can be a ProblemAlarm for one group, but also Trigger or be attached to another group of the same problem. False (by default) → A ProblemAlarm cannot create a new group for the same problem. True → Enable the fact that a ProblemAlarm of a group can also create new group for the same problem.

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 are available that control the creation of trouble tickets for Problem Alarms:

name	type	value
automaticTroubleTicketCreation	boolean	When false, does not automate the creation of a trouble ticket once a Problem Alarm is created When true, automates the creation of a trouble ticket once a Problem Alarm is created
propagateTroubleTicketToSubAlarms	boolean	When true all sub-alarms (of the problem alarm), are associated to the trouble ticket linked with the Problem Alarm When false, sub alarms are not associated to the trouble ticket linked with the Problem Alarm
propagateTroubleTicketToProblemAlarm	boolean	When false, if one sub-alarm has a trouble ticket, the Problem Alarm will not be linked to this trouble ticket When true, if one sub-alarm has a trouble ticket, the Problem Alarm will be linked to this trouble ticket
delayForTroubleTicketCreation	long (optional)	Delay, expressed in milliseconds (after the creation of a Problem Alarm) before the associated trouble ticket is created Default is 10000.

Table 16 – PD troubleTicket "per-problem" configuration

<groupTickFlagAware>

This element of type boolean, when set to true, indicates that at regular tick intervals, the PD Value Pack, if customized for that, will execute some user code.

Hence it should not be changed unless required by VP developer.

<sameGroupForAllProblemEntities>

This property of type Boolean is optional. It only has a meaning if a trigger alarm has multiple problem entities. If a trigger alarm has several problem entities associated, and that this property is set to false, then several group will be created for the same trigger alarm; if the property is set to true, then there will be only one group created for the trigger alarm, and this group will cover all the problem entities of the trigger alarm.

<problemAlarmAbleToCreateGroup>

This property of type Boolean 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.

That generally does not cause any problem, because the lifecycle of this group will be handled.

However for some customers, the lifecycle of Problem Alarms is not handled directly (only lifecycle of non-'problem alarms' is handled), as a consequence, the lifecycle of the group will also not be handled.

For such use case, there is a property 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, will be discarded.

<enableTriggerConsistencyAfterResync>

This property of type Boolean 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. So that in such case, the problem alarm of a group is received before the original trigger that was used to create that group.

So in order to keep consistency among groups, if Problem Detection detects such a case, in which an original trigger alarm is received once the group is already created, because of the prior reception of the problem alarm of that same group, then the original trigger takes back its original role of trigger alarm for that particular, instead of the problem alarm that was in that case assumed as the trigger alarm.

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

This could be useful to disable this feature if, for example, your customization of PD framework already recomputed the trigger alarm.

<computeProblemEntityFromFields>

This element is optional.

It has an attribute "*keyValueSeparator*" that 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	Defines the field key (can be custom field or not) of an alarm used as a key/value pair for computeProblemEntity().
valueIgnored	Property (optional)	Defines an optional value to be ignored for a field during computeProblemEntity().

Note that the property key is in its turn a tuple:

name	type	Value
tagName	string	Defines the tag defining the field name to be used as key/value pair for computeProblemEntity().

fieldName	string	Defines directly the field name to be used as key/value pair for computeProblemEntity().

Table 17 – PD computeProblemEntityFromFields "per-problem" configuration

<timeWindow>

This element holds following properties:

name	type	Value
timeWindowMode	**string**	A TimeWindow is used to decide if an Alarm has to be part of a Group of Alarm depending on its alarmRaisedTime field. When **None** (by default), no time window, this is the equivalent of an infinite time window. All alarms regardless of their timestamp can be associated with a problem. When **Trigger**, a time window around the (first) trigger alarm of a problem is in place. Only alarms with timestamps inside this time window can be associated with a problem.
timeWindowBeforeTrigger	**long (optional)**	Delay, in milliseconds, before the Trigger's alarmRaisedTime field to consider an Alarm as part of the Trigger's problem. Default is 30000.
timeWindowAfterTrigger	**long (optional)**	Delay, in milliseconds, after the Trigger's alarmRaisedTime field to consider an Alarm as part of the Trigger's problem. Default 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 comes usually with 3 standard UCA-EBC configuration files:
- "*TopologyPropagation_filters.xml*" to define the propagation and their tags
- "*TopologyPropagation_filtersTags.xml*" to define the tags associated to the filters
- "*TopologyPropagation_mappers.xml*" to define the different mappers and the neo4j Cypher queries to use within PD VP, mainly specified by tags.

The <t*opFilter*> elements defined in the "*TopologyPropagation_filters.xml*" file are closely related to the TSP VP code itself, since it defines the Java classes corresponding to a specific propagation. Hence, it should not be modified except in some rare conditions, where for example some propagation priority needs to be re-assessed, or to use a new mapper for computing the unique source id of an incoming event, or to update the role of a specific filtered alarm.

The "*TopologyPropagation_filtersTags.xml*" is only used for GUI purpose in the filter builder panel to associate right tags to right filters.

The TSP framework recognizes 3 predefined tags, as followed:

When	The role of the alarm is	And the definition of this role is
tag="RootCauseAlarm"	Root Cause alarm	A Root Cause Alarm which represents a problem, and which and is attached to a specific propagation. In IM Value Pack, such a Root Cause Alarm is a Problem Alarm coming from PD.
tag="SubAlarm"	Sub-Service alarm	Alarm representing a propagation but which is associated under a higher Service alarm
tag="ServiceAlarm"	Service alarm	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 associate above tags.

The <*cypherQuery*> elements defined in the "*TopologyPropagation_mappers.xml*" file are closely related to the topology loaded in Neo4j, it should not be modified except in some rare conditions. However, the <*mapper*> elements may be changed to handle new conditions on incoming events, but in such a case, the "*TopologyPropagation_filtersTags.xml*" should be updated accordingly.

5.4.1.1 The special topFilter named ReservedForGeneralBehavior

When an event comes in TSP framework, there is a need to compute its unique source identifier. This is done by calling the *computeSourceUniqueId*() method of the GeneralBehavior class defined for all propagations.

TSP framework brings 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

When an event comes in TSP framework, once a unique source identifier has been computed, there is a need to retrieve the topology records associated to the object represented by that event, i.e. all the nodes impacted by that object.

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

The default Propagation class brought by IM framework does the following:

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

Note that for GUI filter builder purpose, usually, all the <*cypherQuery*> elements that are defined in "*TopologyPropagation_mappers.xml*" should 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 comes with a specific "*PropagationXmlConfig.xml*" file.

5.4.2.1 The Main Policy

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

It has one attribute:

name	type	value
stateSourceIdentifier	String attribute	It is used to fill up the "sourceIdentifier" field of a State event generated by the TSP framework.

Table 21 – TSP mainPolicy attributes

Also, as PD framework, it has following elements:

<actions>

The TSP Framework is able to configure multiple actions factories in order to support multiple NMS. Refer to section 5.2.1 to get the details.

Note that this element is optional.

<troubleTicketActions>

The TSP Framework is able to configure trouble ticket actions factories. Refer to section 5.2.2 to get the details.

Note that this element is optional.

<counterTotalNumberAlarms>

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

Note that this element is optional.

5.4.2.2 The Propagation Specific Policies

Propagation Policies are configuration settings which are 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)	When true, the Service Alarm is automatically created for this propagation. When false (by default), no Service Alarm is created for the propagation.
delayForServiceAlarmCreation	long (optional)	Delay, expressed in milliseconds, before the creation of the associated Service Alarm. Example: Setting the value: 10000 to this property apply a delay of 10 seconds before creating Service Alarms. Default value is 2000.
delayForServiceAlarmClearance	long (optional)	Delay, expressed in milliseconds, before clearing the service alarm. Example: Setting the value: 0 (ms) to this property does not delay the clearance of Service Alarms after all conditions are met for clearing Service Alarms. Default value is 10000.
attachWholeSubTreeRootCauses	boolean (optional)	When true, the whole sub-tree of Root Cause alarms are attached to the Service Alarm, i.e. the direct Root Cause alarms plus the Root Causes alarms part of impacting states. When 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 are available that control the creation of trouble tickets for Service Alarms:

name	type	value
automaticTroubleTicketCreation	boolean	When false, does not automate the creation of a trouble ticket once a Service Alarm is created When true, automates the creation of a trouble ticket once a Service Alarm is created

propagateTroubleTicketToSubAlarms	boolean	When true all sub-alarms (of the Service alarm), are associated to the trouble ticket linked with the Service Alarm When false, sub alarms are not associated to the trouble ticket linked with the Service Alarm
propagateTroubleTicketToMasterAlarm	boolean	When false, if one sub-alarm has a trouble ticket, the Service Alarm will not be linked to this trouble ticket. When true, if one sub-alarm has a trouble ticket, the Service Alarm will be linked to this trouble ticket
delayForTroubleTicketCreation	long (optional)	Delay, expressed in milliseconds (after the creation of a Service Alarm) before the associated trouble ticket is created Default is 10000.

Table 23 – TSP troubleTicket "per-propagation" configuration

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

<groupTickFlagAware>

This optional element of type boolean, when set to true, indicates that at regular tick intervals, the TSP Scenario, if customized for that, will execute some user code.

Hence it should not be changed unless required by VP developer.

<propagationRule>

Not used.

<nodes>

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

name	type	value
key	Property	Defines the type of the node to include

<poiCategories>

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

name	type	value
key	Property	Defines the POI category to assign

<thresholdValues>

This optional element is a sequence of 6 elements used to configure the Threshold values. Those elements are in strict order

name	type	value

OK	Property	Defines the threshold for state OK
LOW	Property	Defines the threshold for state LOW
MEDIUM	Property	Defines the threshold for state MEDIUM
HIGH	Property	Defines the threshold for state HIGH
CRITICAL	Property	Defines the threshold for state CRITICAL
DOWN	Property	Defines the threshold for state DOWN

Each of the Threshold value above should be defined using 3 elements:

name	**type**	**value**
perceivedSeverity	Property	Defines the perceived severity for that threshold value. Is one of : - INDETERMINATE - WARNING - MINOR - MAJOR - CRITICAL - CLEAR
availabilityPercentage	Property	Defines the percentage of availability of the node for that threshold value. Is a double.
poiImportance	Property	Defines the importance for the POI for that threshold value. Is one of: - None - Low - Medium - High - Critical

You can have an example in section 0

<*propagationObject* >

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

<*statusName*>

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

<*percentageAvailabilityKey*>

This optional element is a string defining the attribute name used for 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 usually brings its Orchestra configuration that should 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 12 IM Orchestra configuration example .

```
<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 12 IM Orchestra configuration example

For details on how to use the UCA-EBC V3.1 Orchestration feature, please refer to [R1] *Unified Correlation Analyzer for Event Based Correlation Reference Guide* and to [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 UCA for EBC Inference Machine Development Kit which is comprised of

- UCA for EBC Development Kit (see UCA for EBC Value Pack Development Guide)
- UCA for EBC Development Kit Inference Machine Extension

There are 4 pre-defined Value Pack that you can choose to create:

- 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)

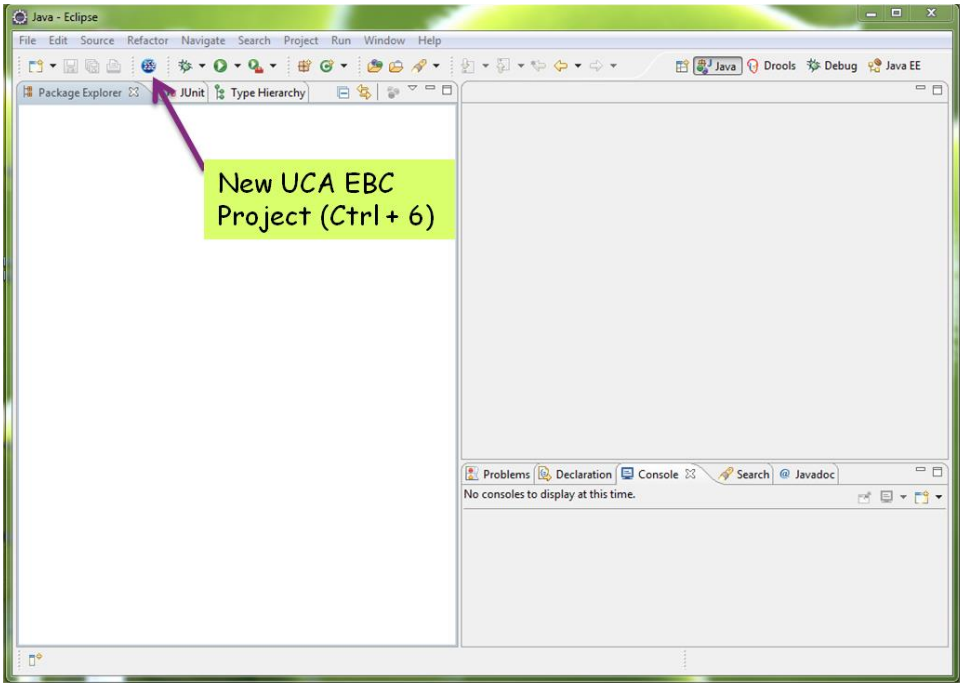


Figure 13 - How to create a UCA EBC project in Eclipse

6.1.1 Problem Detection only Value Pack

When creating, your Value Pack, you should select only "Problem Detection Scenario"

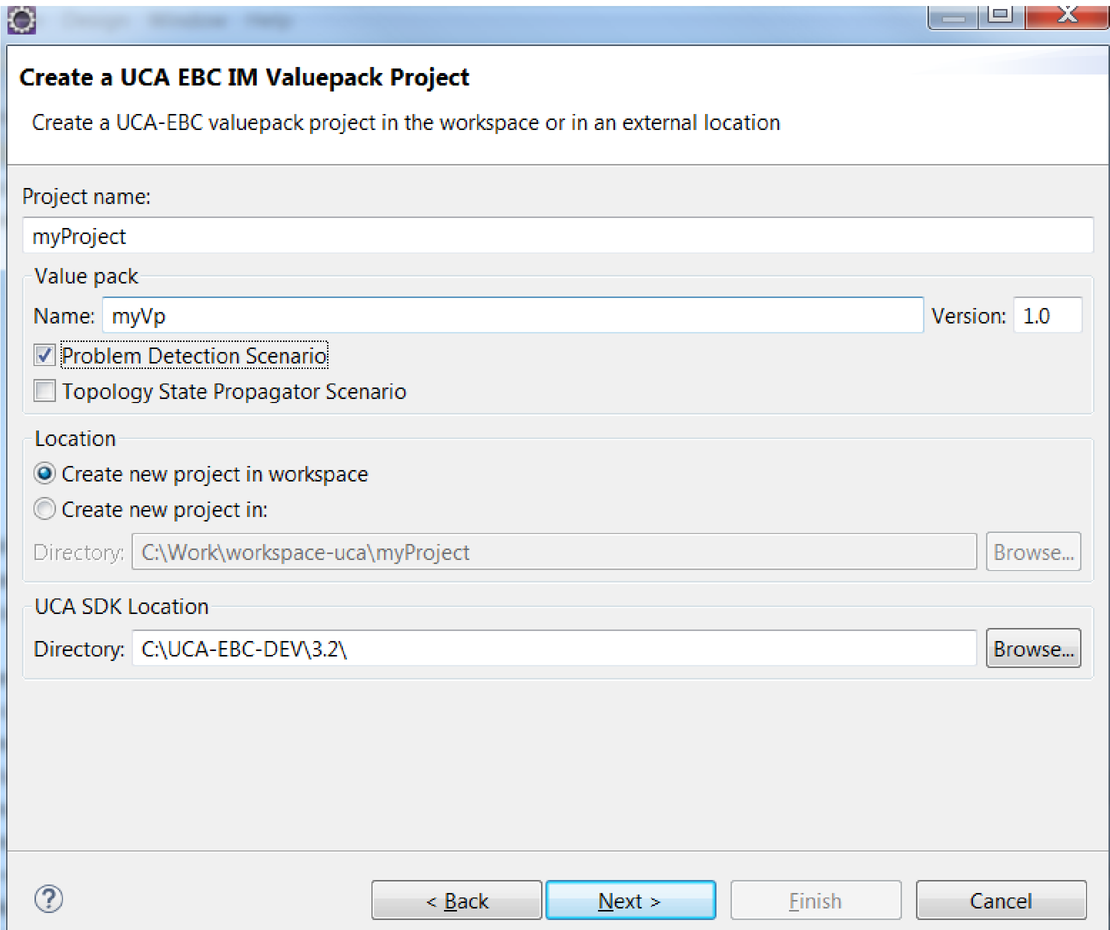


Figure 14 – Create PD only Value Pack

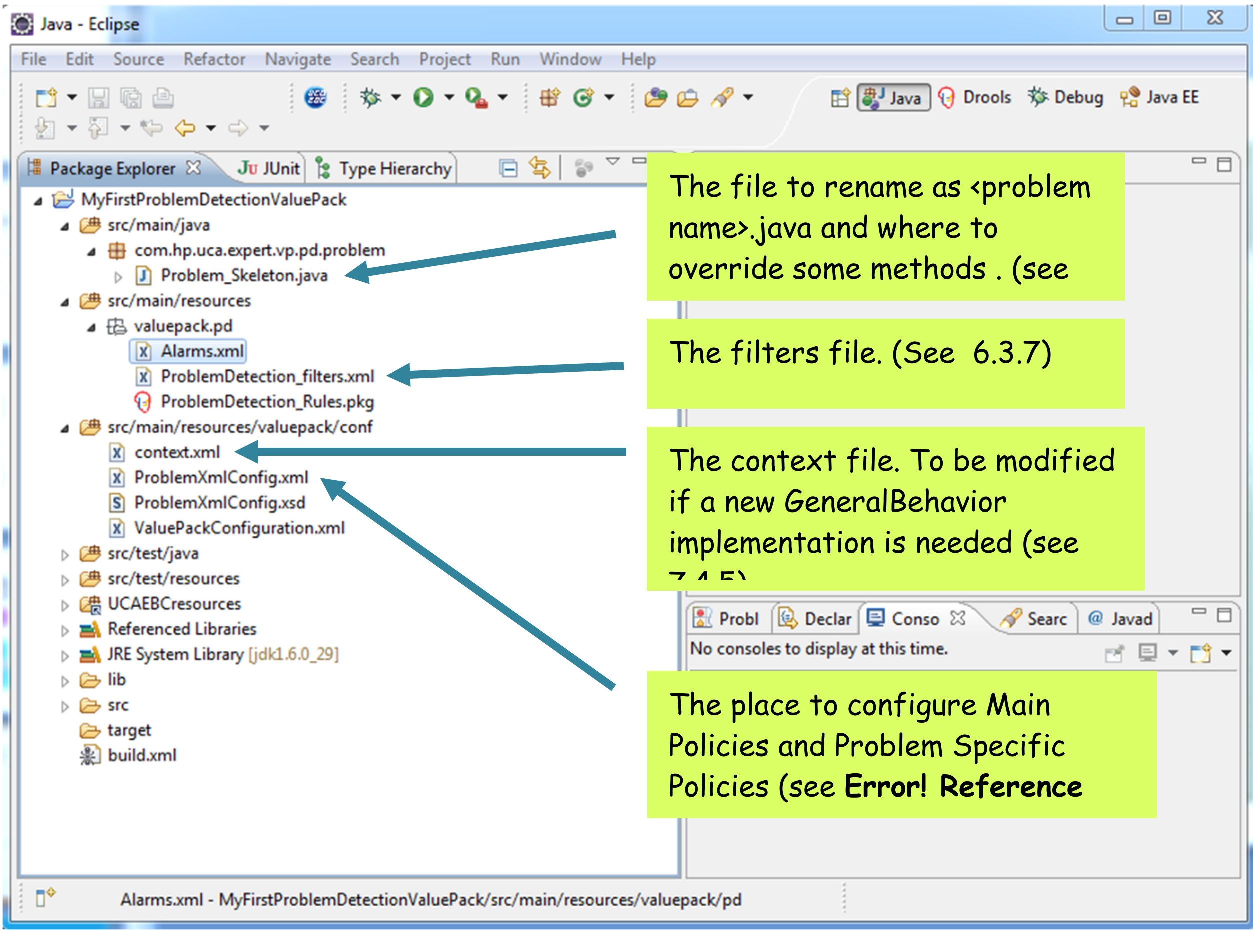


Figure 15 - Files to edit to configure MyFirstProblemDetectionValuePack

Step 3: **Mandatory steps.** Rename and edit “Problem_Skeleton.java”. Edit the filters file. 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.

6.1.2 UCA EBC Topology State PropagatorTopology State Propagator only Value Pack

When creating, your Value Pack, you should select only "Topology State Propagator Scenario"

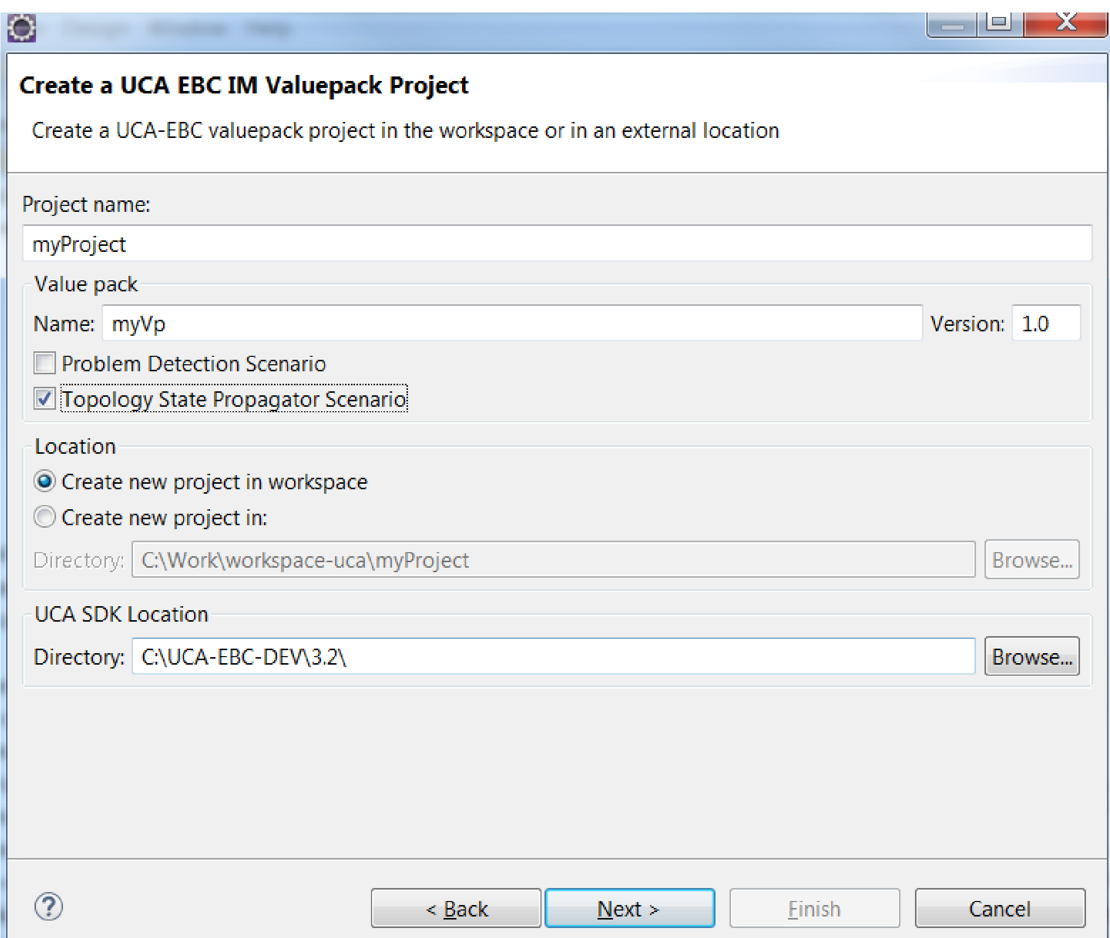


Figure 16 – Create TSP only Value Pack

6.1.3 Inference Machine Value Pack

When creating, your Value Pack, you should select both "Problem Detection Scenario" and "Topology State Propagator Scenario"

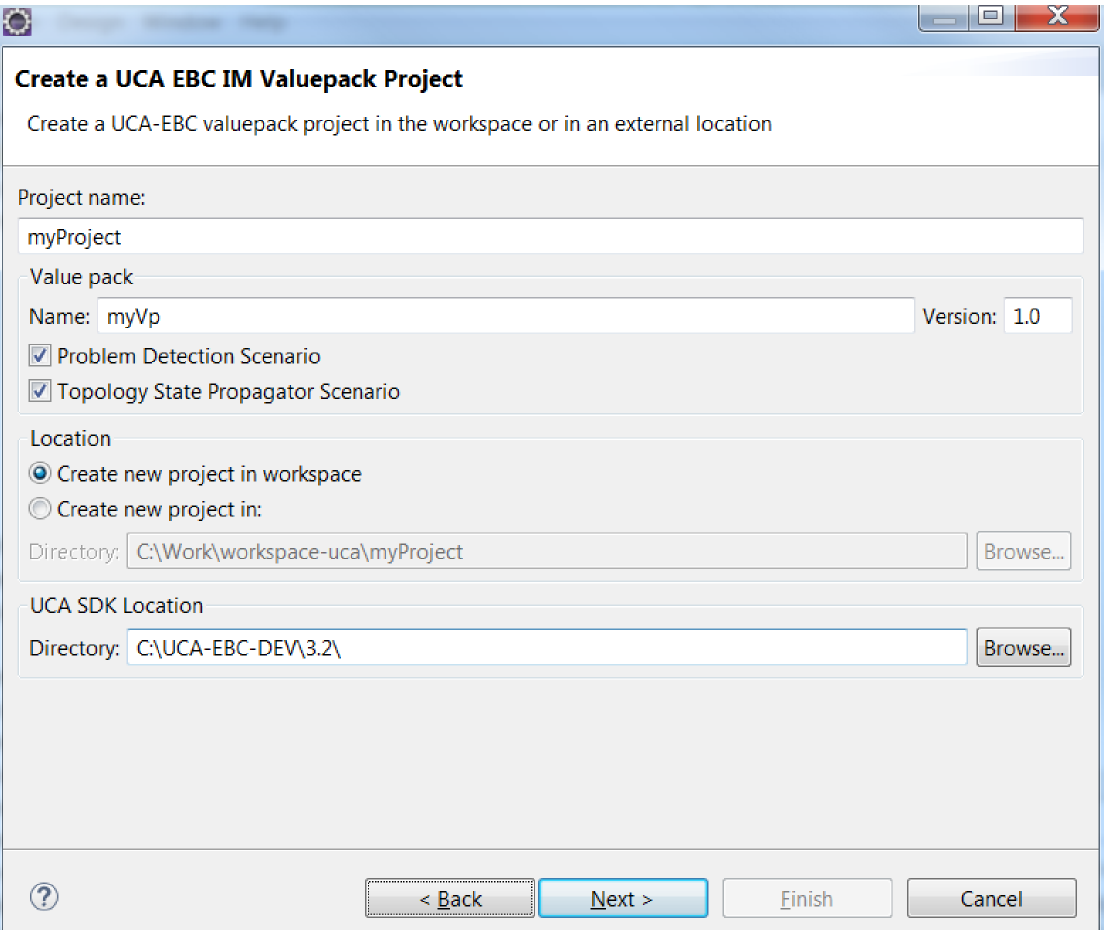


Figure 17 – Create IM Value Pack

6.2 Understanding the Use Cases

This chapter should guide you choosing the right skeleton for your VP.

Unfortunately, this chapter is not yet available.

6.3 Create a Simple PD VP

The objective of this chapter is to list and briefly explain the steps required to create a meaningful Problem Detection Value Pack. For readability reasons, in this entire chapter it is assumed that both the reader and the writer are developers of a Problem Detection Value Pack and will be referred to as "**We**".

6.3.1 Analyze 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 will be generated in the context of each problem.

To use a medical analogy:

- Alarms are the symptoms
- Problem is the disease, and the
- Problem Detection Value Pack is the physician. Based on the symptoms observed (the alarms received), she will diagnose the disease (identify the problem).

Creating a Problem Detection Value Pack first implies listing all the potential problems (and their associated alarms) that we want to identify.

To summarize, we need to:

- list all potential alarms that the NMS (Network Management System) may receive
- do the RCA analysis: list the problems that might occur in the network and that the user of a NMS is likely be interested in
- for each problem, identify which alarms are associated with the problem (please note that an alarm can be associated with several problems)

6.3.2 Identify the different types of alarms

Among all the alarms associated with a problem, we need to separate out the "trigger" alarms from the "sub-alarms". Continuing with the medical analogy made above, we want to separate the primary symptoms (trigger alarms) from the secondary symptoms (sub-alarms). Trigger alarms are called as such because they define the kind of Problem we are facing and they will trigger the creation of a Problem Alarm.

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 has been received
- at least one sub-alarm of the problem has been received

This default behavior can be customized (see sections 7.4 and 8.1.12)

Once the "trigger" alarms and "sub-"alarms have been identified,

Once we have the list of interesting problems (resulting from above step 6.3.1), the list of interesting alarms, the association between alarms and problems,

We are ready to configure the filters of our 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 (trigger alarm, sub-alarm ...)

Filters are configured in a XML file.

See detailed explanation in section 6.3.7 Define the Filters. See also Annex B.

6.3.3 Configure 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 have to 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 candidate to be associated with the problem

Note that time windows can be infinite.

The following diagram illustrates the time window, defined by `timeWindowBeforeTrigger` and `timeWindowAfterTrigger`.

`timeWindowBeforeTrigger` and `timeWindowAfterTrigger` are properties set in a configuration file. Refer to section 5.3.2.2

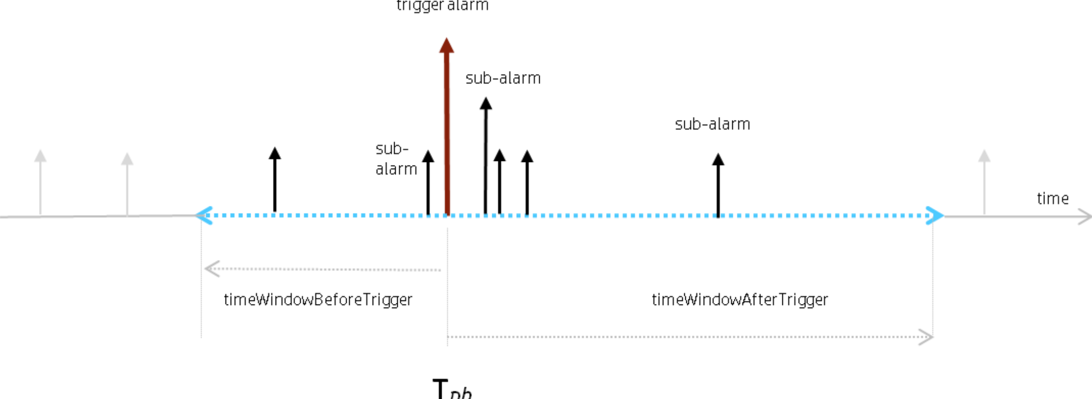


Figure 18 - Time window illustration

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.

6.3.4 Create a Problem Alarm?

For each problem, we have to decide whether, at runtime, upon occurrence of the problem, the Problem Detection Value Pack will create a Problem Alarm or re-use (promote) the trigger alarm (or one of the trigger alarms) as a Problem Alarm.

This is done in the filters XML configuration file. See detailed explanation in section 6.3.7

If we have decided that a fresh problem alarm has to be created, we need to configure an action to effectively create this problem alarm in the Network Monitoring System (NMS).

We also need to configure when the Problem Alarm will be created. Problem alarm can be created as soon as the problem is detected or after a given amount of time. See Chapter 7.

6.3.5 Create a Trouble Ticket?

For each problem, we have to decide whether, at runtime, upon occurrence of the problem, the Problem Detection Value Pack will raise a trouble ticket. . See Chapter 7.

6.3.6 Is the default behavior good enough?

Problem Detection proposes a default behavior which allows you to create a Value Pack without having to go through heavier configuration phases than the ones described in sections 6.3.1 to 6.3.5

Yet, the Problem Detection Framework is extremely open, and allows us to customize almost any behavior we would like to change.

By default, the Problem Detection Framework sets the severity of the Problem Alarm to be the severity of the sub-alarm (among all sub-alarms of the problem) having the highest severity. We may want to change that rule. The Problem Detection Framework allows you do just that.

Default behaviors and ways to customize them are detailed in See Chapter 7.

One of the default behaviors 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 could be some location information (“Paris_south _MKF2”) contained in the AdditionalText.

6.3.7 Define the Filters

Defining the filters is the primary and 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 is the role of each alarm: Problem Alarm, trigger alarm, sub-alarm.

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

The definition of filters is done in a file named "*ProblemDetection_filters.xml*" 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 example below shows the "top filter" section of a "ProblemDetection_filters.xml" file for one problem named "*Problem_BitError*".

To see an example of a filter file that contains several "top filter" sections in order to detect several problems, please consult 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 tag `<topFilter name="Problem_BitError">` signifies the beginning of the filters definition for the “Problem_BitError” problem

The tags

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

mean that conditions from block A **or** conditions from block B must be met, or both.

The tags

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

...

</allCondition>

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

The tags <anyCondition> and <allCondition> 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 tag

```
<allCondition tag="TeMIP TT">
```

means that all alarms passing all the conditions included in this tag will be associated to one given Trouble Ticket System, TeMIP TT in this case.

The possible values for the tag name are given in the <troubleTicketActions> section of file ProblemXmlConfig.xml.

Please see section 5.3.2 for more information on the ProblemXmlConfig.xml file.

The tags

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

mean that alarms having the additionalText field containing the text: "[6] Remote Alarm OOS Threshold Exceeded" will be considered trigger alarms for the "Problem_BitError" problem.

When	The role of the alarm is	And the definition of this role is
tag="Trigger"	Trigger alarm	Alarm which is an important symptom of a problem, and which triggers the creation of a problem alarm
tag="SubAlarm"	Sub-alarm	Alarm which is a symptom of a problem and is grouped under a Problem alarm
tag="ProblemAlarm"	Problem alarm	Alarm that summarizes the problem, and is readable by the operator
tag="SubAlarm,ProblemAlarm"	SubProblemalarm	Alarm which is Problem alarm of a problem, and sub-alarm of another problem

Table 25 – PD: Possible roles for an alarm

If we want a trigger alarm to be used as a Problem Alarm (instead of creating a fresh one), the tag of the trigger alarm has to be as follows: `tag="Trigger, ProblemAlarm"`.

6.3.8 Configure Value Pack

The file named "`ValuePackConfiguration.xml`" located in the `src/main/resources/valuepack/conf/` folder does not need to be modified except the highlighted part below, which concerns mediation flows. Detailed instructions are available in chapter 'Value Pack definition file' of the UCA for EBC 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 file named "`context.xml`" located in the `src/main/resources/valuepack/conf/` folder does not need to be modified, unless you want to customize the enrichment example (enrichment bean highlighted) or if you want to customize some behavior as explained in 7.4.5
For more information on context.xml, please refer to chapter 'Value Pack definition' in the UCA for EBC 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>
```

6.3.9 Configure 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 inside 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 inside the `<problemPolicy name="…">` XML tag.

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

Please note that the XML schema of this file named "*ProblemXmlConfig.xsd*" is available in the *src/main/resources/valuepack/conf/* folder.

You can also configure Transient Filtering, Actions, Trouble Tickets actions, Problem Alarm handling, etc.

For details, refer to Configuration chapter.

6.3.10 Customize the behavior

```
<problemPolicy name="XmlGeneric_Synch">
[…]
```

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

As explained in paragraph 7.4.1, it is possible to assign basic customization directives for a specific problem (XmlGeneric_Synch in above extract).

6.4 Create a Simple TSP VP

The objective of this chapter is to list and briefly explain the steps required to create a meaningful Topology State Propagator Value Pack. For readability reasons, in this entire chapter it is assumed that both the reader and the writer are developers of a Topology State Propagator Value Pack and will be referred to as "**We**".

6.4.1 Analyze 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.

Identify all the propagations that could arise induced by a state update of its underneath (from a topology point of view) propagations. 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
- Problem Detection Value Pack is the physician. Based on the symptoms observed (the alarms received), she will diagnose the disease (identify the problem)
- The correlated information containing the disease (the Problem Alarm) will be received by TSP
- TSP will analyze and, if the disease is extremely contagious, it will propagate it, resulting for example in epidemics (propagation on all upper level of the topology), if is less contagious will impact only groups with low immunity systems (propagation on part of the upper level of the topology) or not impact at all (no propagation in the topology).
- Other secondary symptoms like the fact that a large number of persons were impacted before, so they stopped coming for example to the kindergarten.
- TSP will realize Service Impact Analysis, so for example epidemics in a kindergarten can result in stopping the lessons activity (service) for a period.

Creating a Topology State Propagator Value Pack first implies listing all the potential propagations (and their associated states and alarms) that we want to identify.

To summarize, we need to:

- Set the topology to put in place and establish the nodes and relationships needed. For details on the topology extension, the [R9] *Unified Correlation Analyzer for Event Based Correlation Topology Extension Guide* can be consulted.
- Do the SIA analysis: detect the services on which the impact is wanted to be computed
- Using the IM (RCA-SIA pattern): list all potential alarm that may come from, standardly, a 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 it is encouraged to use it in conjunction with PD as an IM package.
- List the propagations that might occur in the topology
- For each propagation, identify which alarms are associated with the propagation (please note that an alarm can be associated with several propagations).

6.4.2 Compute a State?

Default State computation is performed by TSP framework. It is however possible to change this default computation to set your specific thresholds, etc...

If you want to change it in Java, then you need to override the method

```
boolean computeState(PropagationGroup group)
```

Note that the default behavior is to use the service

```
TP_Service_StateCalculation.computePercentageAvailability()
```

Which calculates the percentage of availability of the impacted node and deduces the state from the thresholdValues defined in configuration. Refer to section 5.4.2.2 for more information on those thresholdValues.

6.4.3 Identify the different types of alarms: Root Cause or Sub Alarms

As for PD an identification of the different types of alarms was needed, the same applies to TSP. Basically, as TSP is used in the IM on top of PD, a significant reduced number of alarms should be considered to be received by TSP as alarms will already be summarized before by PD into Problem Alarms. Among all the alarms associated with a propagation, we need to separate out the “root cause” alarms from the “sub-alarms”. Root Cause alarms are called as such because they have a role in the propagation, by contributing 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. Continuing with the medical analogy made above, we want to separate the primary symptoms (disease which is a root cause alarm) from the secondary symptoms (a large number of persons stopped coming to the kindergarten which are sub-alarms).

At runtime, by default, a Topology State Propagator Value Pack considers that an instance of a propagation has occurred if the following criteria is met:

The State of the propagation has been received. The computation of a State is overridable method by the VP developer so depending on the

needs, the creation of a state can be based on different criteria (like for example a certain number of root cause alarm received having their status critical or other).

This default behavior can be customized (see section 8.1.12).

If the topology is set, if the possible impacting states and root cause alarms are identified, as well as the propagations realizing the service impact analysis are set, then the filters of our 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 have to 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, service alarm.)

Filters are configured in a XML file.

See detailed explanation in section 6.4.6 Define the Filters. See also Annex E.

6.4.4 Create a Service Alarm?

In comparison with PD, in TSP creating a Service Alarm is optional. For each of the propagations, we have to decide whether, at runtime, upon occurrence of the propagation and based on several conditions meet, the TSP Value Pack will create a Service Alarm. This alarm will contain particular fields and it can only be created by the framework.

This is done in the filters XML configuration file. See detailed explanation in section 6.4.6.

As for the Problem Alarm in PD, there is the possibility for the Service Alarm to be created or cleared after a configurable time, as described in section 5.4.2.2.

6.4.5 Create a Trouble Ticket?

For each of the propagations, as for the problem in PD, we have to decide whether, at runtime, upon occurrence of the propagation, the TSP Value Pack will raise a trouble ticket, as described in section 5.4.2.2.

Is the default behavior good enough?

Topology State Propagator, as Problem Detection, proposes a default behavior which allows you to create a Value Pack without having to go through all the configuration phases described in the upper sections.

Nevertheless, TSP Framework is extremely open, and allows the customization of almost any behavior we would like to change.

Default behaviors and ways to customize them are detailed in Chapter 8.

6.4.6 Define the Filters

Defining the filters is the primary and most important step when creating a TSP Value Pack. Defining filters is not only about specifying which events (state and 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, RootCauseAlarm, Service Alarm, or SubAlarm.

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

The definition of filters is done in a file named "*TopologyPropagation_filters.xml*" 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 is defined in the "*TopologyPropagation_*filters.xml", the *ReservedForGeneralBehavior,* using the extended mappers feature of UCA EBC V3.2. The example below shows the contents of this "top filter".

```
<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 defining the *ReservedForGeneralBehavior* top filter are defined in the file "*TopologyPropagation_*tags.xml" shown in the example below.

```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 plus of the definition of the *ReservedForGeneralBehavior* top filter, propagations are defined in the "`TopologyPropagation_filters.xml`" file. The example below shows the "top filter" section of a "`TopologyPropagation_filters.xml`" file for one propagation named "*Propagation_BtsOrNodeB*".

To see an example of a filter file that contains several "top filter" sections in order to detect several propagations, please consult 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 tag `<topFilter name="Propagation_BtsOrNodeB">` signifies the beginning of the filters definition for the "*Propagation_BtsOrNodeB*" propagation

The tags

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

```
    …
</anyCondition>
```

mean that conditions from block A **or** conditions from block B must be met, or both.

The tags

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

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

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

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

Please see section 6.4.8 Configure for more information on the PropagationXmlConfig.xml file.

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

Table 26 – TSP: Possible roles for an alarm

6.4.7 Configure Value Pack

The file named "*ValuePackConfiguration.xml*" located in the *src/main/resources/valuepack/conf/* folder does not need to be modified except the highlighted part below, which concerns mediation flows. Detailed instructions are available in chapter 'Value Pack definition file' of the UCA for EBC 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 file named "`context.xml`" located in the `src/main/resources/valuepack/conf/` folder does not need to be modified, unless you want to customize the enrichment example (enrichment bean highlighted) or if you want to customize some behavior as explained in 7.4.5
For more information on context.xml, please refer to chapter 'Value Pack definition' in the UCA for EBC 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="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>
```

6.4.8 Configure 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 inside 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 inside the `<propagationPolicy name="…">` XML tag.

Policies are configured in a file named "*PropagationXmlConfig.xml*" located in *src/main/resources/valuepack/conf/*.

Please note that the XML schema of this file named "*PropagationXmlConfig.xsd*" is available in the *src/main/resources/valuepack/conf/* folder.

For details, refer to Configuration chapter 5.4.2.

6.5 Create 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.

You can refer to the IM example delivered with the IM SDK for a good example of a standard IM VP.

Chapter 7

Advanced features of Problem Detection

Once configured (see section 5.3), a Problem Detection Value Pack runs with a standard 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 The default behavior explained

The Problem Detection Framework is a set of Java libraries, with some Java classes that can be extended and methods 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.

The default behavior of all these methods is available by consulting the javadoc. The implementation code of these methods is available in the example value pack delivered as part of the Problem Detection Dev Kit (See 0 pd-example, content of src/test/resources) The code of each of these methods is executed for every problem for which that method and can be overridden by the value pack developer.

Firstly, an example is presented in 7.1.1 and secondly the different interfaces available.

7.1.1 Example

An example of how the workflow of the different methods triggered in the case of an alarm Network State Update is shown in the sequence diagram in Figure 19, where an alarm clearance is managed for the following context: alarm 1 is Problem Alarm in group1 of Problem1, sub Alarm in group2 of Problem2 and has no role for any of Problem3's groups.

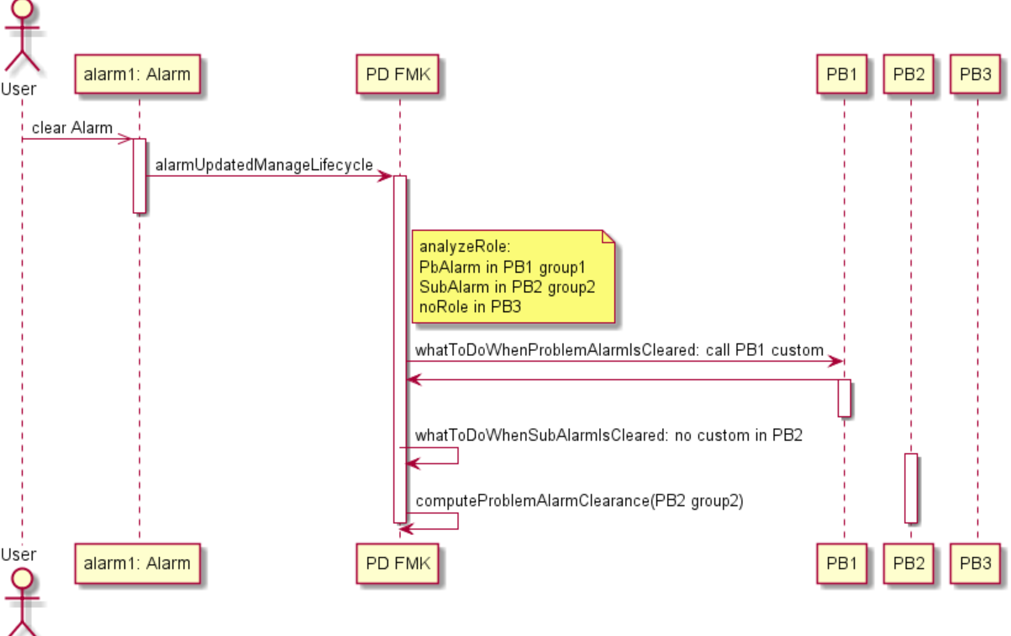


Figure 19 Alarm clearance sequence diagram example

Let's say that at before the alarm1 clearance is received, the groups are as shown in Figure 20.

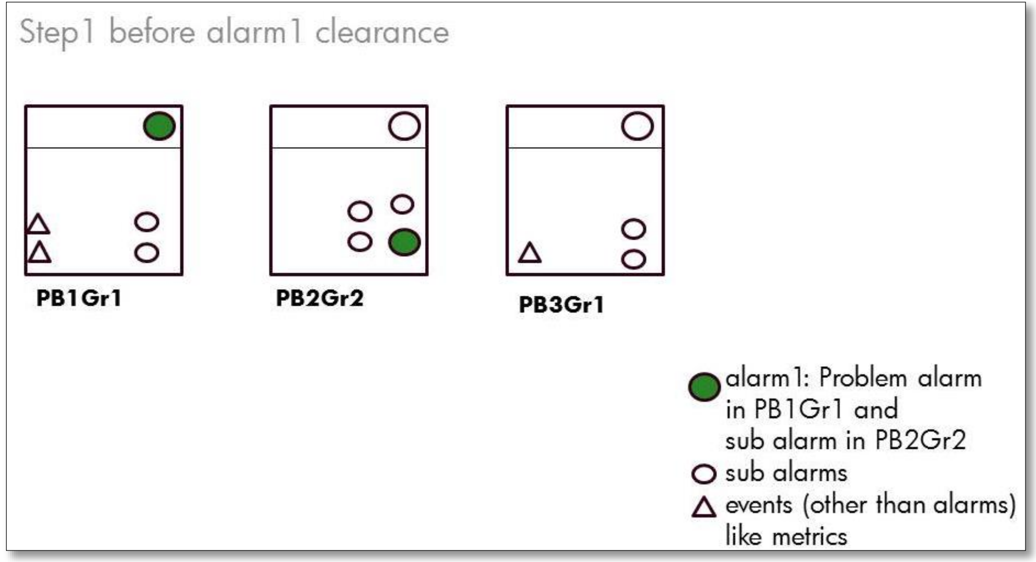


Figure 20 PD: Alarm clearance example: PD group updates Step1

The alarm1 clearance is received and, according to the sequence diagram in Figure 19Figure 26, a number of methods will be called by the PD framework. Therefore, the problem alarm in group1 of PB1 will be cleared an removed from the Working Memory. Concerning the group2 in PB2, alarm1 has a role as sub alarm, but its clearance will result in the computation of the group2 from PB2 clearance, but we assume that it will have an impact only in its severity change for example, but the problem alarm in PB2 group2 will still be present. Concerning group1 in PB3 there will be no impact. Therefore, the groups could look like represented in Figure 21.

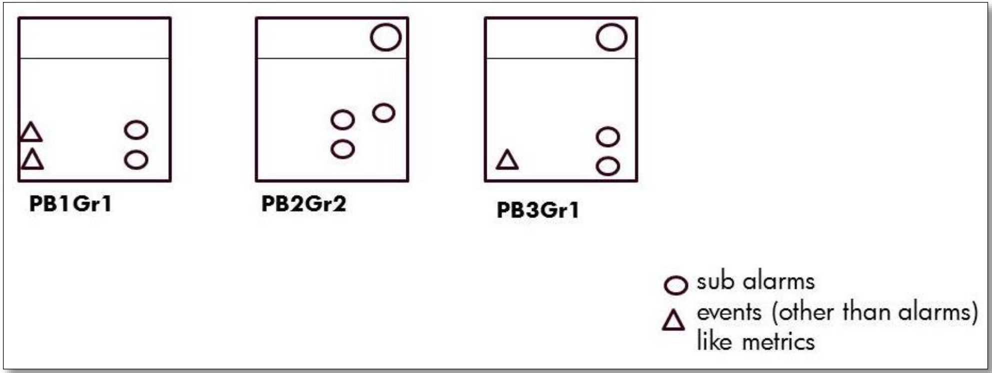
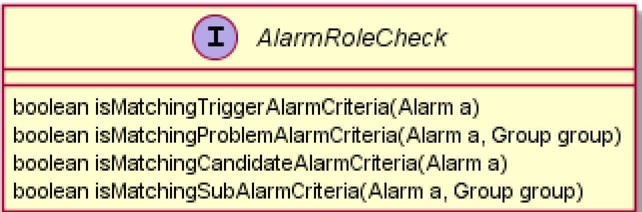
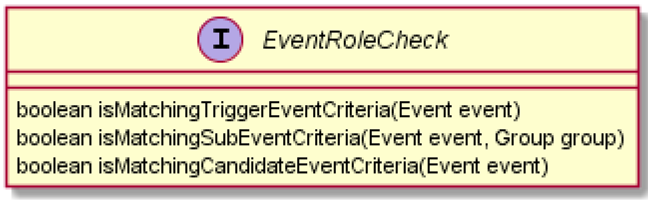


Figure 21 PD: Alarm clearance example: PD group updates Step2

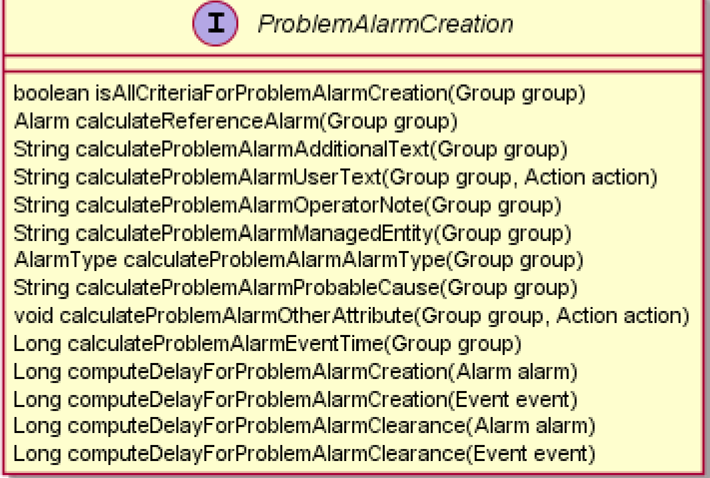
7.1.2 Alarm Role Check



7.1.3 EventRoleCheck



7.1.4 Problem Alarm Creation



Method used to check if ProblemAlarm should be created

`isAllCriteriaForProblemAlarmCreation(Group)`

Methods used during ProblemAlarm Creation

`calculateReferenceAlarm(Group)`

`calculateProblemAlarmManagedEntity(Group)`

`calculateProblemAlarmAlarmType(Group)`

`calculateProblemAlarmProbableCause(Group)`

`calculateProblemAlarmAdditionalText(Group)`

`calculateProblemAlarmOperatorNote(Group)`

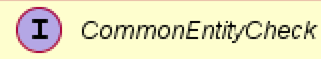
`calculateProblemAlarmUserText(Group, Action)`

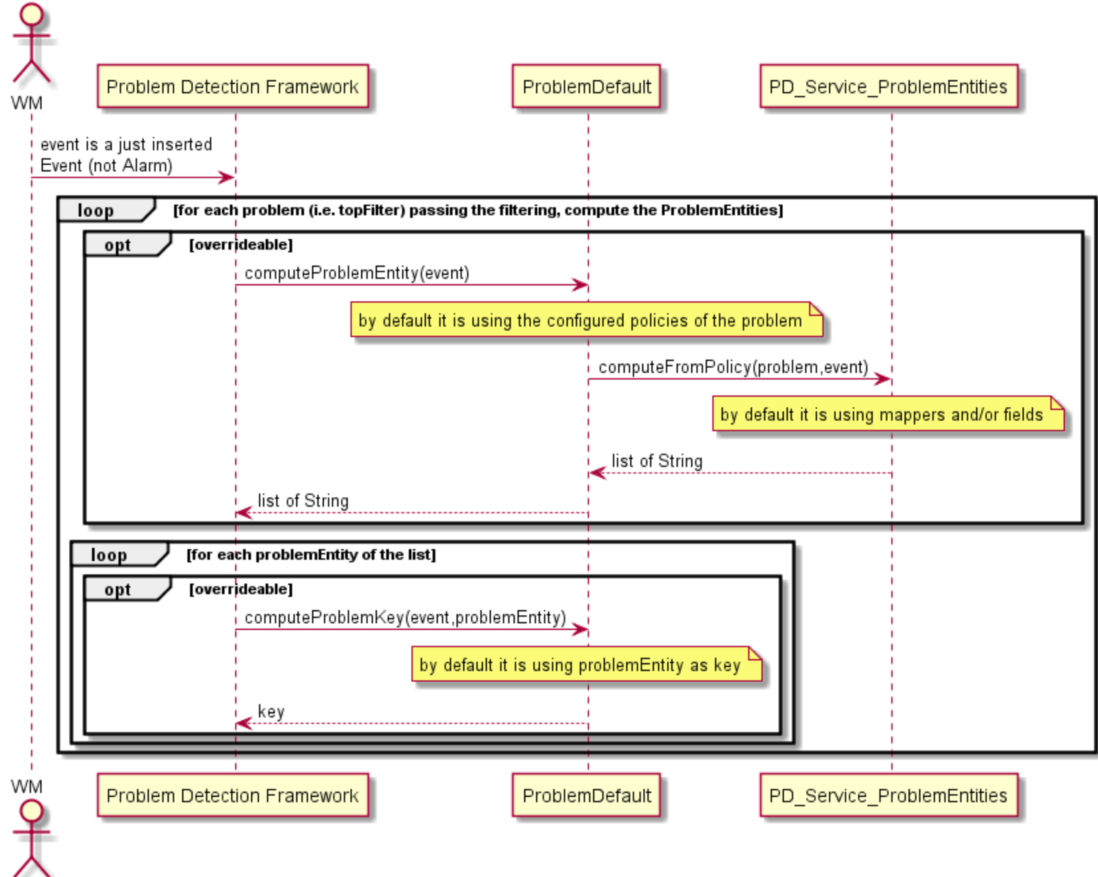
`calculateProblemAlarmEventTime(Group)`

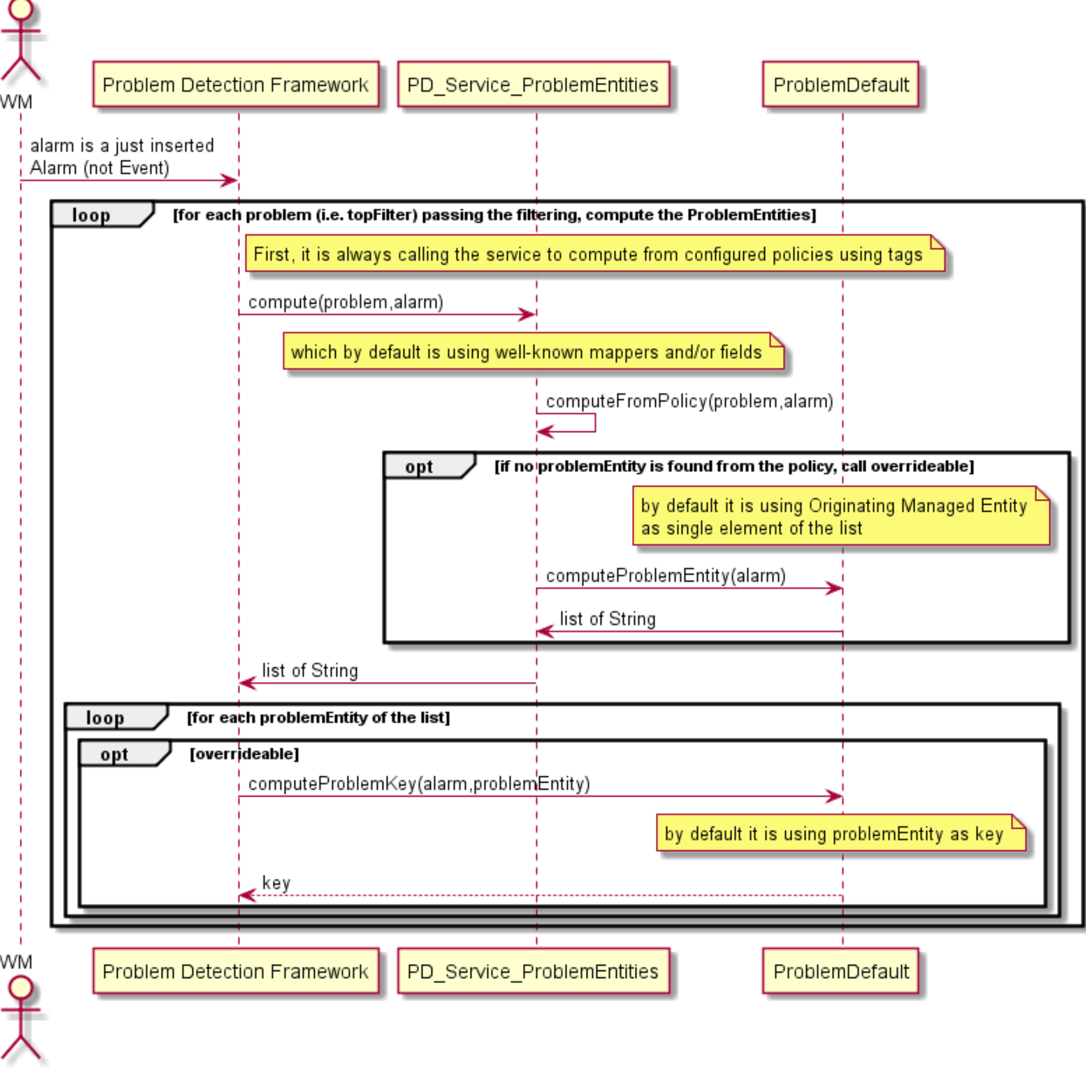
`calculateProblemAlarmOtherAttribute(Action)`

7.1.5 Common Entity Check

Methods used to calculate Information for optimizations

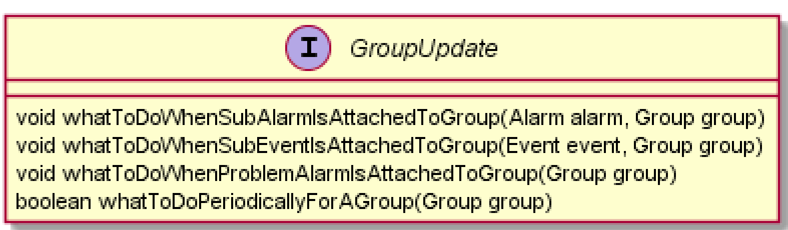



Understanding the computeProblemEntity(Event event)
WM
Problem Detection Framework
ProblemDefault
PD_Service_ProblemEntities
event is a just inserted
Event (not Alarm)
loop
[for each problem (i.e. topFilter) passing the filtering, compute the ProblemEntities]
opt
[overrideable]
computeProblemEntity(event)
by default it is using the configured policies of the problem
computeFromPolicy(problem,event)
by default it is using mappers and/or fields
list of String
list of String
loop
[for each problemEntity of the list]
opt
[overrideable]
computeProblemKey(event,problemEntity)
by default it is using problemEntity as key
key
WM
Problem Detection Framework
ProblemDefault
PD_Service_ProblemEntities

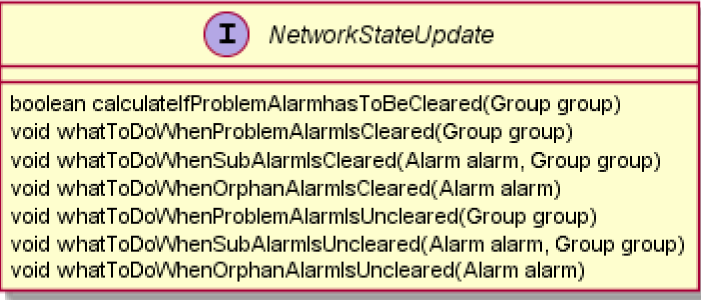


7.1.6 Group update

Methods used to manage the group lifecycle, and its associated alarms



7.1.7 Network State Update



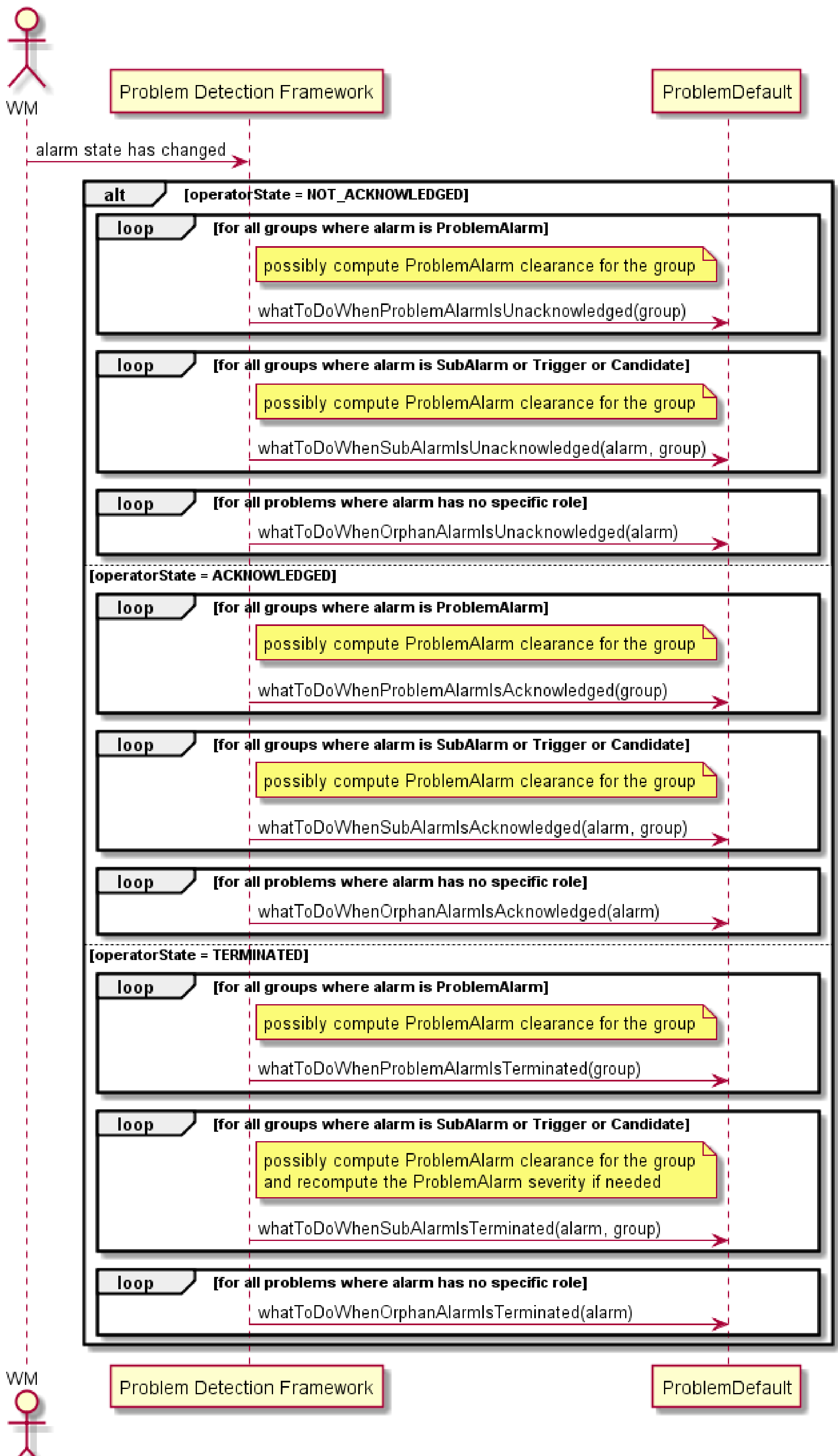
Alarm Network State Changes

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

7.1.8 Operator State Update

OperatorStateUpdate

void whatToDoWhenProblemAlarmIsTerminated(Group group)
void whatToDoWhenProblemAlarmIsAcknowledged(Group group)
void whatToDoWhenProblemAlarmIsUnacknowledged(Group group)
void whatToDoWhenSubAlarmIsTerminated(Alarm alarm, Group group)
void whatToDoWhenSubAlarmIsAcknowledged(Alarm alarm, Group group)
void whatToDoWhenSubAlarmIsUnacknowledged(Alarm alarm, Group group)
void whatToDoWhenOrphanAlarmIsTerminated(Alarm alarm)
void whatToDoWhenOrphanAlarmIsAcknowledged(Alarm alarm)
void whatToDoWhenOrphanAlarmIsUnacknowledged(Alarm alarm)

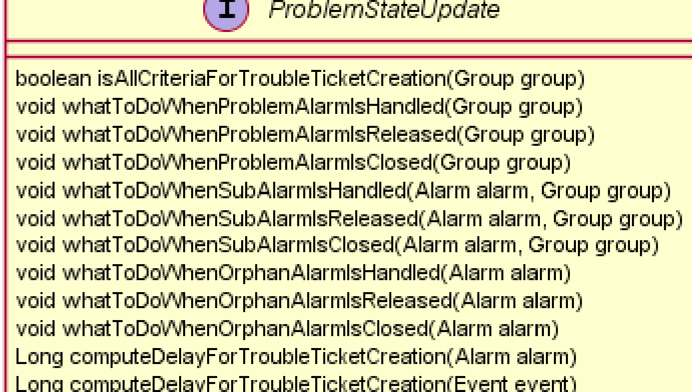

Alarm Operator State Changes
WM
Problem Detection Framework
ProblemDefault
alarm state has changed
alt
[operatorState = NOT_ACKNOWLEDGED]
loop
[for all groups where alarm is ProblemAlarm]
possibly compute ProblemAlarm clearance for the group
whatToDoWhenProblemAlarmIsUnacknowledged(group)
loop
[for all 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

7.1.9 Problem State Update

Methods used to manage the Trouble Ticket lifecycle when related to a

- Problem Alarm
- SubAlarm
- Orphan Alarm

And its consequence



7.1.10 Attribute Update

Methods used to manage a Severity or an Attribute Update of a:

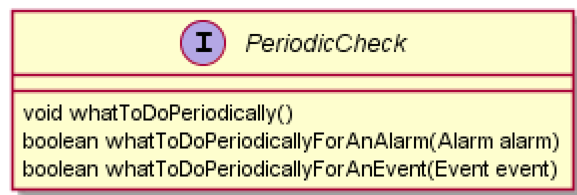
- Problem Alarm
- SubAlarm
- Orphan Alarm

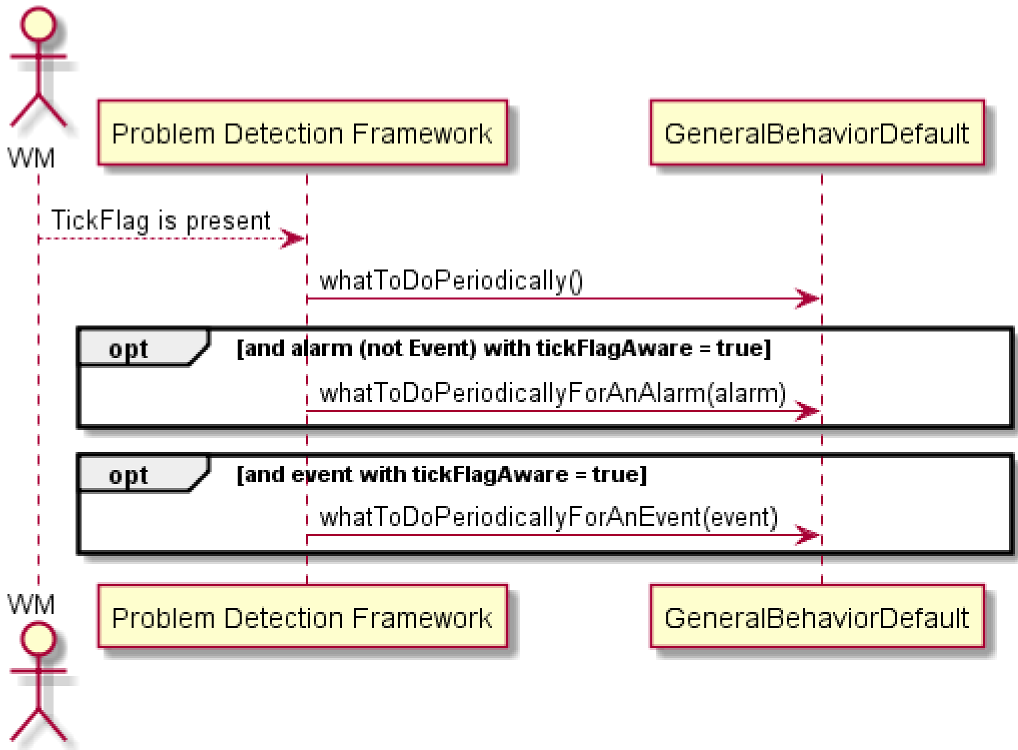
And its consequence

AttributeUpdate

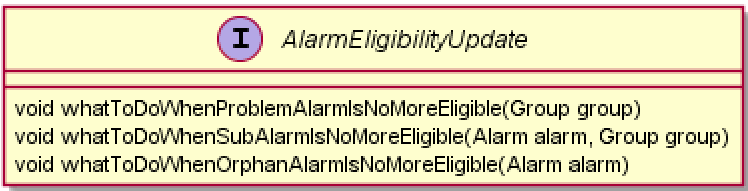
void whatToDoWhenProblemAlarmSeverityHasChanged(Group group)
void whatToDoWhenSubAlarmSeverityHasChanged(Alarm alarm, Group group)
void whatToDoWhenOrphanAlarmSeverityHasChanged(Alarm alarm)
PerceivedSeverity calculateProblemAlarmSeverity(Group group)
void whatToDoWhenProblemAlarmAttributeHasChanged(Group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

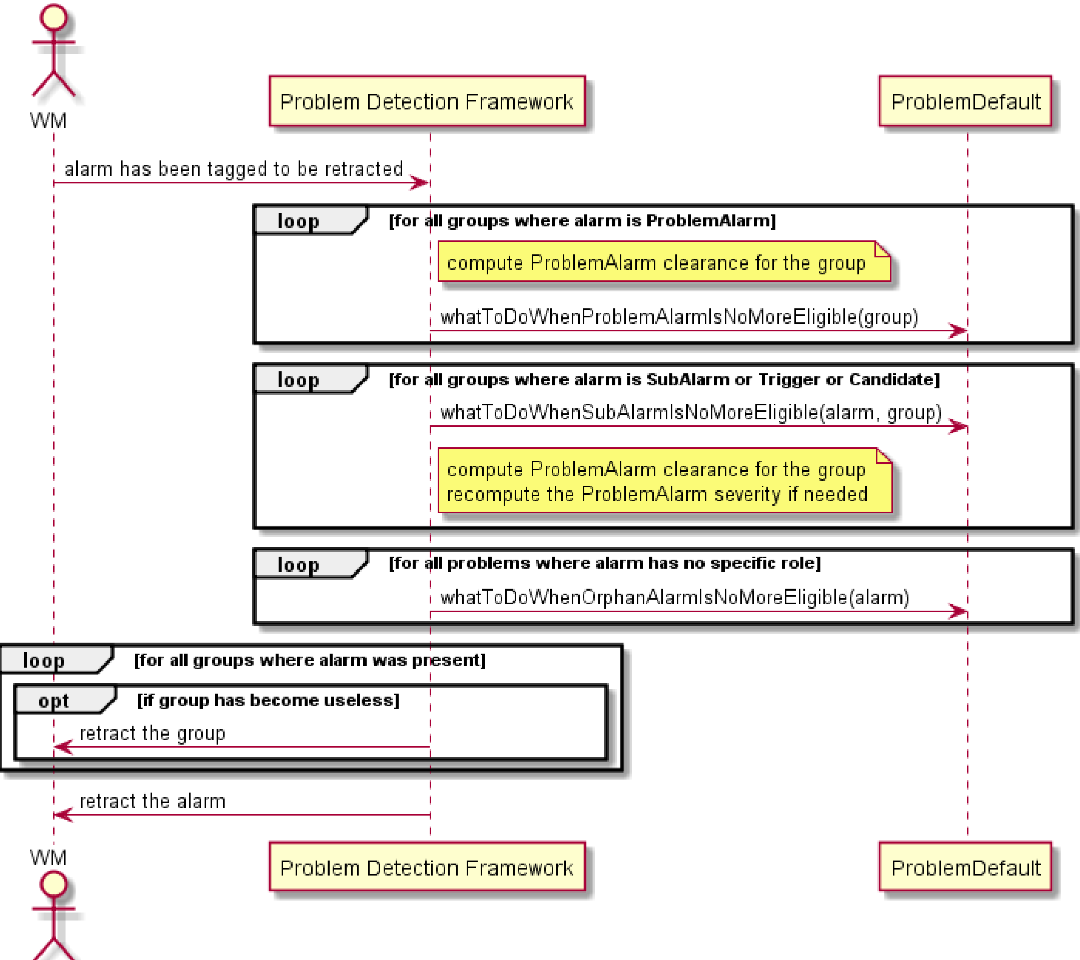
7.1.11 Periodic Check



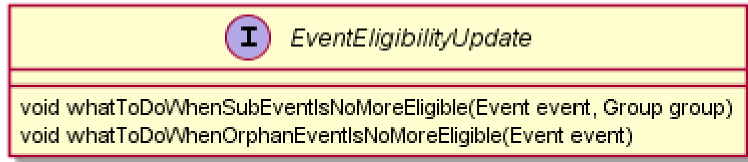


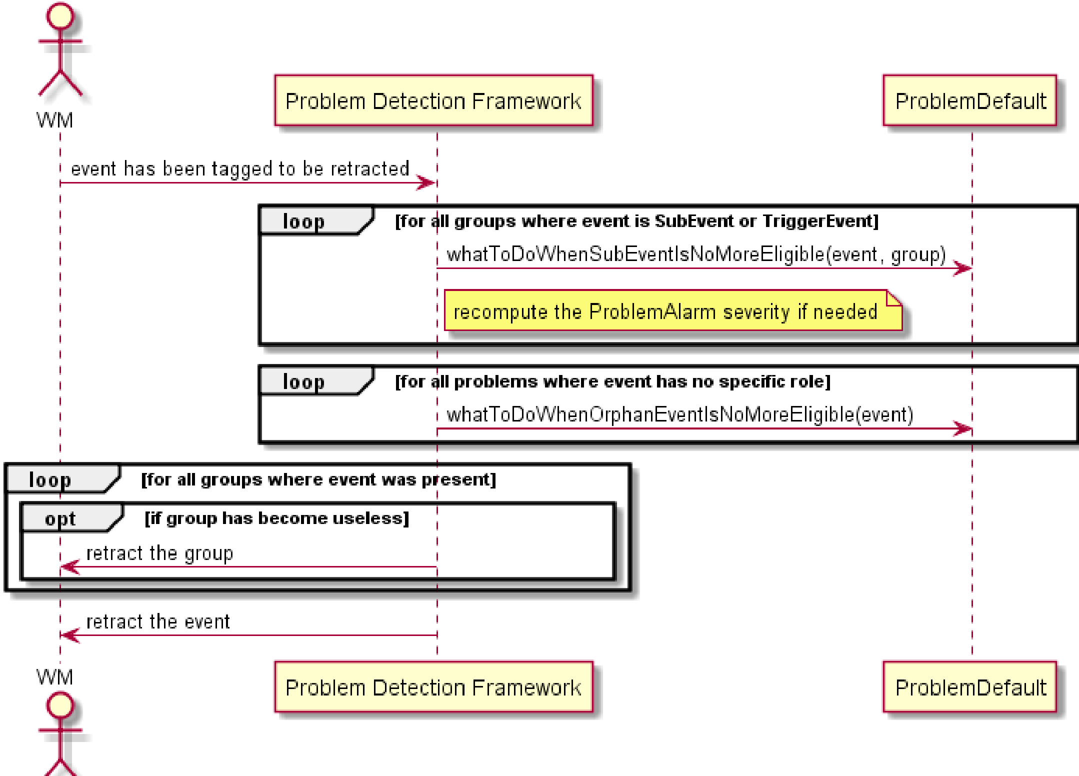
7.1.12 Alarm eligibility update





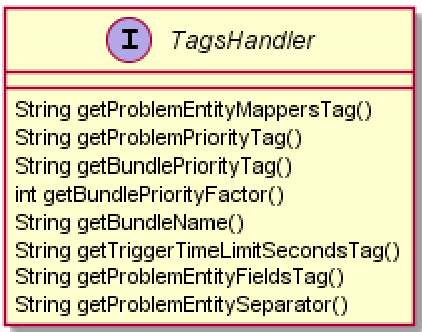
7.1.13 Event eligibility update

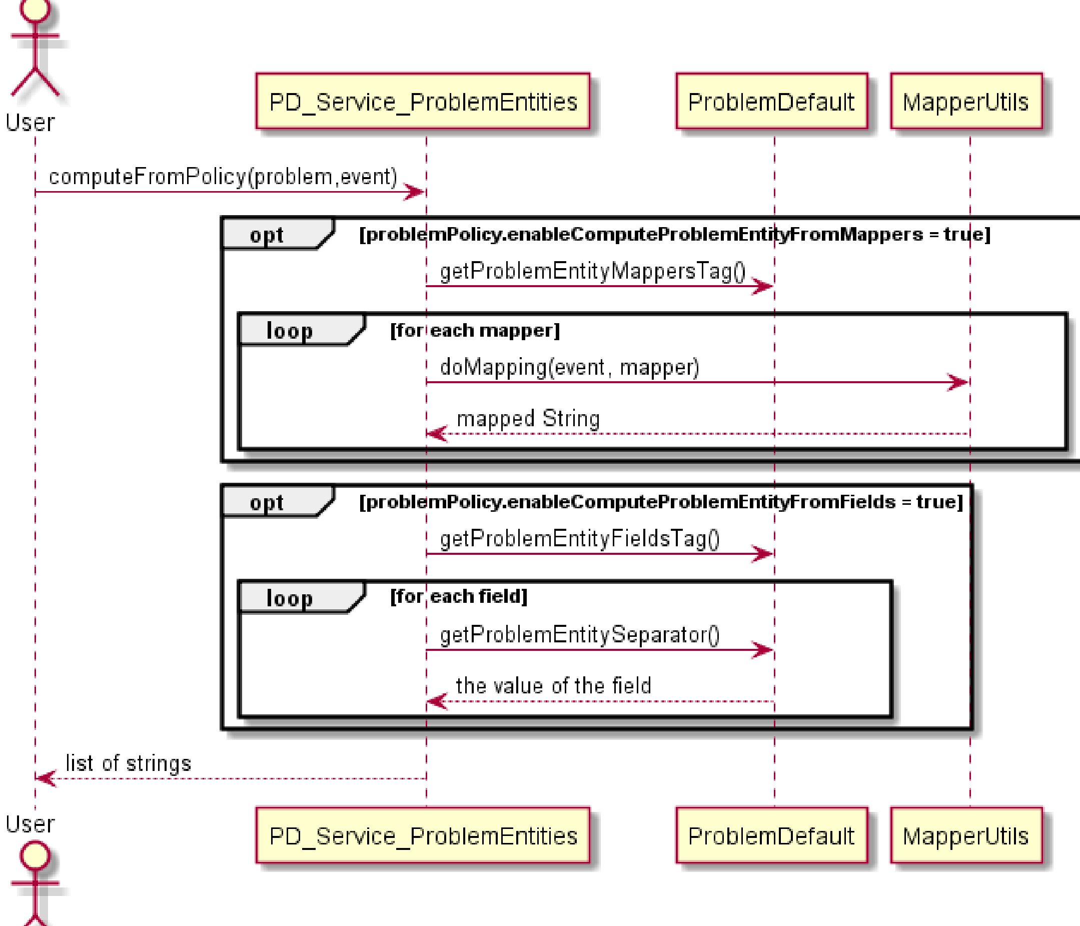


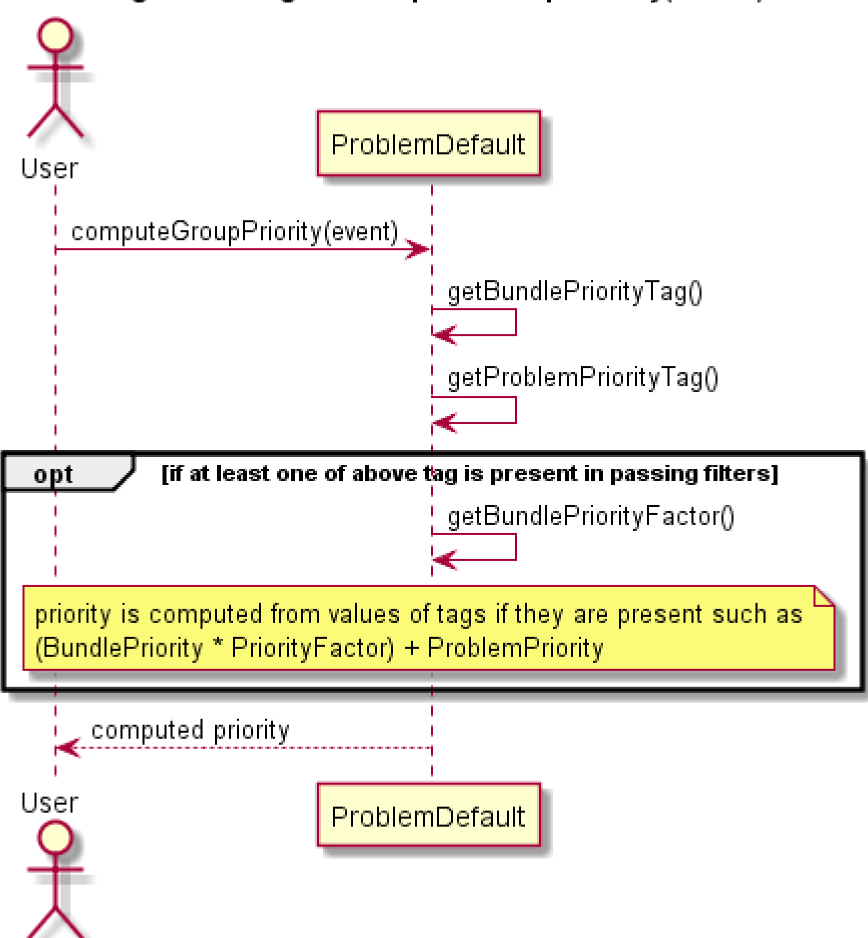


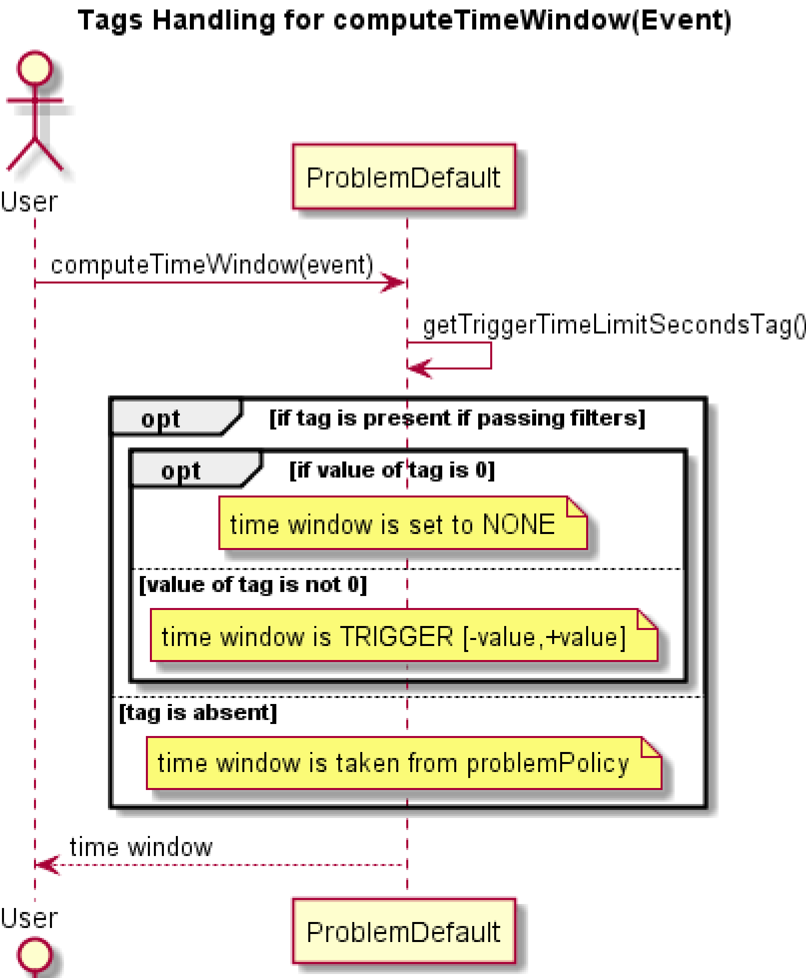
7.1.14 Tags handling

These are features introduced in V3.2. They are used to control the tag names used by the Problem Detection filters tags.






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



7.2 Generic Events (other than Alarm types) are supported

Problem Detection V3.2 is able to correlate generic events and group them. Hence:

- The Trigger of a PD correlation group can be now an Event (type introduced in UCA-EBC V3.1)

Most methods are applicable therefore for the Event type as parameter and not only Alarm. That explains why some methods are now deprecated.

A VP using PD with Events (other than Alarm types) only is delivered as an example with the IM SDK, described in Annex D.

7.3 Computing Problem Information starting V3.2

When new alarm comes in Problem Detection, Problem information is now computed in two ways, described in the following subsections (7.3.1 and 7.3.2).

7.3.1 Case where Problem Detection is topology-aware

In such a case, 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 should provide the name of the Cypher Query to execute

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

Notes:

- *The above default conditions can be changed by overriding the ProblemDefault.isAllowingDbAccess(Event event) method.*
- *In case of successful computation, method ProblemDefault.computeProblemEntity(Event event) is therefore not used.*

7.3.2 Default case (non-topology aware)

If above case does not apply or fails, the new *ProblemDefault.computeProblemEntity(Event event)* is used.

7.3.3 ProblemXmlConfig schema changes

7.3.3.1 Namespace

Some elements defined in the ProblemXmlConfig.xml configuration file are now coming from the common schema defined in the IM common library, having therefore a different namespace. Hence, existing configuration file should be migrated. Refer to “0 How do I migrate my PD VP 3.0/3.1 to 3.2?” migration steps for more information.

7.3.3.2 MainPolicy

New attribute ***enablePrioritySort*** : Boolean flag indicating whether the groups should be sorted on priority order or not. Default is false.

New attribute ***multipleParentSupport*** : Boolean flag indicating whether an alarm grouping will send the parent relationship only for the highest priority parent (false), or for each of the ProblemAlarm where this alarm is grouped (true). Default is true.

New attribute ***enableTopoAccess*** : Boolean flag indicating whether to use topologyAccess when computing information for Problem Alarm (by calling *computeSourceUniqueID(Event event)* and *computeDBRecords()* methods) during the workflow) (true) or not (false). Default is false.
When true, the *computeProblemEntity(Event event)* is not called. Attention, this uses Neo4j database, so requires Topology license.

7.3.3.3 ProblemPolicy

New attribute ***enableComputeProblemEntityFromMappers***: When true, enables the use of calling mappers in *computeProblemEntity()*. Default is true,

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

New element ***computeProblemEntityFromFields***: Configuration of the FieldsChooser element, which is a sequence of fields to use as keys. Called in *computeProblemEntity()* when computation of fields key/value pairs is enabled and when ComputeProblemEntityFields tag is not used.

7.3.4 ProblemDefault.computeProblemEntity(Event event)

This V3.2 method that takes Event as parameter. It is called by the existing ***computeProblemEntity(Alarm alarm)*** method.

The default behavior of the new ***computeProblemEntity(Event)*** method has been completely improved to satisfy most of the end-user needs.
It executes the following procedures (7.3.4.1, 7.3.4.2 and 7.3.4.3) in respective order.

7.3.4.1 Usage of extended mappers

Firstly, it makes use of the UCA-EBC V3.2 feature: the extended mappers.

When an event comes in the Problem Detection Value Pack, it is checked against the presence of the filter tag named "ComputeProblemEntityMappers" which is a parameter tag that should contain the name of the mapper(s) to use for computing the problem entity.

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

The usage of extended mappers is automatically taken into account.

Notes about mappers' usage:

- The mappers usage can be disabled by setting the corresponding *ProblemPolicy.enableComputeProblemEntityFromMappers* attribute to false in ProblemXmlConfig.xml file. By default, it is considered as true.
- Each mapper name in the "*ComputeProblemEntityMappers*" tag should be separated by ".".
- You can change the name of the filter tag used by overriding the *getProblemEntityMappersTag*() method of your problem.

7.3.4.2 Direct mapping of alarm fields as key/value pairs

Secondly, if requested, it can make use of the fields of the alarm computed as key/value pair. This function work as described below, each option being evaluated in following order:

1. Use of a well-known tag

If the filter tag "*ComputeProblemEntityFields*" is present in the incoming alarm filtered tags, that tag should contain the name of the field(s) to use for computing the problem entity. Each field described in this tag is checked against its presence in the alarm and the resulted problemEntity is computed as ***$field.name$separator$field.value***.

Notes about ComputeProblemEntityFields filter tag usage:

- The computation of the key/value pairs can be enabled by setting the corresponding ProblemPolicy.enableComputeProblemEntityFromFields attribute to true in ProblemXmlConfig.xml file. By default, it is considered as false; hence this feature is by default not used.
- Each field name in the "ComputeProblemEntityFields" tag should be separated by ".".

- You can change the name of the filter tag used by overriding the getProblemEntityFieldsTag() method of your problem.
- You can change the value of ***$separator*** used by overriding the getProblemEntitySeparator() method of your problem. By default, it is "=".

2. Use of V3.2 policy

The corresponding *ProblemPolicy.computeProblemEntityFromFields* element can be defined in ProblemXmlConfig.xml file and is used for computing the problem entity. This policy defines a sequence of XML field elements and a keyValueSeparator XML element which is by default "=".

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

When tagName is defined, it corresponds to a tag that should be present if the incoming alarm filtered tags which should 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 should be present in the alarm and is equivalent to $field.key.tagName.value

When tagName is not defined and fieldName is defined, it corresponds directly to the field of the alarm to take into account.

The field name is then checked against its presence in the alarm and the resulted problemEntity is computed as $fieldName$keyValueSeparator$fieldName.value

Notes about computeProblemEntityFromFields policy usage:

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

7.3.4.3 Default mode

When none of above two methods is used, the function returns as previously (up to V3.1) the originating managed entity of the incoming Alarm.

7.3.4.4 Modification of examples

The pd-example value pack contains the updated classes *Problem_Synch* and *Problem_BitError*, which are showing the usage of extended mappers feature to compute their problem entity based on bsc and bts identifiers. The *computeProblemEntity*() function has then been removed from those classes, which are now using the mapper getBscBtsFromUserText instead.

7.3.5 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, *i.e.* when the *MainPolicy.enableTopoAccess* attribute is set to true. In such a case, a special filter should be defined with the *ReservedForGeneralBehavior* as the filter name.

Inside this filter, the *ComputeSourceUniqueIdMapper* tags are used to compute the source unique Id. When mappers are defined in the topFilter having the name *ReservedForGeneralBehavior*, Problem Detection will call the *computeSourceUniqueId(Event)* method.

Example (extracts of filters and mappers files):

```
<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>
```

7.3.6 ProblemDefault.computeDbRecords(String dbUniqueIdReference, Event event)

This method is used to calculate the Neo4j query, which will be executed to retrieve the data base records for having the database id reference for the Event. Called by the Problem Detection Framework when the *MainPolicy.enableTopoAccess* attribute is set to true and when ***CypherQuery*** tag is present.

Example (extracts of filters and mappers files):

```
<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>
```

7.3.7 ProblemDefault.computeGroupPriority(Event event)

A default implementation has been introduced to make use of specific tags that can be set at filters level: "***Bundle.Priority***" which defines the priority of the family of Problems and "***Problem.Priority***" which defines the priority of the 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 left to *null*.

The group priority is automatically taken into account if the attribute **enablePrioritySort** is defined to **true** in MainPolicy of *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 present; hence groups are not sorted by default.

Notes about the priority computation:

Lower priority numbers come first. A *null* priority comes last.

You can change the value of the *$priority.factor* used by overriding the *getBundlePriorityFactor()* method of your problem.

You can change the name of the *Bundle.Priority* tag used by overriding the *getBundlePriorityTag()* method of your problem.

You can change the name of the *Problem.Priority* tag used by overriding the *getProblemPriorityTag()* method of your problem.

7.3.7.1 Example with Alarm

Trigger alarm A1 comes in with *Bundle.Priority*=10, *Problem.Priority*=1 => group G1 priority will be set to 10001.
Trigger alarm A2 comes in with *Problem.Priority*=2 => group G2 priority will be set to 2.
Trigger alarm A3 comes in with no tags => group G3 priority will be set to null.
Now suppose an alarm S is subalarm of all 3 above groups => the *getGroups().getGroupsWhereAlarmSetAs(S, Qualifier.SubAlarm)* will return the groups [G2, G1, G3] in strict order if *MainPolicy.enablePrioritySort* is set.

7.3.7.2 Example with Event (other than Alarm)

Trigger event E1 comes in with *Bundle.Priority*=10, *Problem.Prio*rity=1 => group G1 priority will be set to 10001.
Trigger event E2 comes in with *Problem.Priority*=2 => group G2 priority will be set to 2.
Trigger event E3 comes in with no tags => group G3 priority will be set to null.
Now suppose an event S is subEvent of all 3 above groups => the *getGroups().getGroupsWhereEventSetAs(S, EventQualifier.SubEvent)* will return the groups [G2, G1, G3] in strict order if *MainPolicy.enablePrioritySort* is set.

7.3.8 ProblemDefault.computeTimeWindow(Event event)

The default behavior of the default *computeTimeWindow(Alarm alarm)* method has been changed to make use of specific tag "*Trigger.TimeLimit.Seconds*" that can be set at filters level and can be applied on the Event generic type.

If this tag is present after filtering an alarm, and given that the value is T, the timeWindow returned overrides the one defined at ProblemPolicy level and is computed as:

If T is 0: TimeWindowMode.NONE
If T is not 0: TimeWindowMode.TRIGGER and Window is [abs(T) * 1000 , abs(T) * 1000]
Note: you can change the name of the *Trigger.TimeLimit.Seconds* tag used by overriding the *getTriggerTimeLimitSecondsTag()* method of your problem

7.4 How to customize default behavior

The ways to customize the behavior of a Problem Detection Value Pack are:

- to override some java methods specially defined for this purpose
- or to write some customization XML code.

The list of java methods that can be overridden is presented in section 7.1 Default Behavior. The way to override those 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 presented 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`" of the trigger alarm as "Problem Entity". Since one important objective of creating a Problem Alarm is to show clear and concise information to the operator, it may be useful to redefine the way Problem Detection computes the "Problem Entity" of a problem. This can be done without writing any Java code as shown below. This can also be done by writing Java code (see next section).

Below is an extract of "`ProblemXmlConfig.xml`" file located in the `src/main/resources/valuepack/conf/` folder.

It shows an example of two methods: the `computeProblemEntity()` and `calculateProblemAlarmAdditionalText()` methods, being overwritten:

```
<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>
```

Also available are the three following methods. Note that all other methods listed in 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>
```

In paragraph 6.3.7Define the Filters, Table 12 – Tags for possible roles of an alarm, we saw that the role of an alarm is determined by the tag associated to it in the Filters xml file. However if some of the three methods above are overridden, then what happens?
For instance, does the tag="SubAlarm" takes precedence over the criteria defined in the `isMatchingSubAlarmCriteria(alarm)` method ?

The answer is that for an alarm a to be considered a sub-alarm by the Problem Detection Value Pack, it needs to be tagged as subalarm in the Filters xml file **and** the method `isMatchingSubAlarmCriteria(a)` must return true.

7.4.2 Java customization

The main way to customize the default behavior of Problem Detection Value Packs is to override some of the Java methods listed in section 7.1. There are three levels of customization:

- Per problem (this section)
- For a set or for all problems (section 7.4.3 “My ProblemDefault”)
- For non-problem specific matters (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.

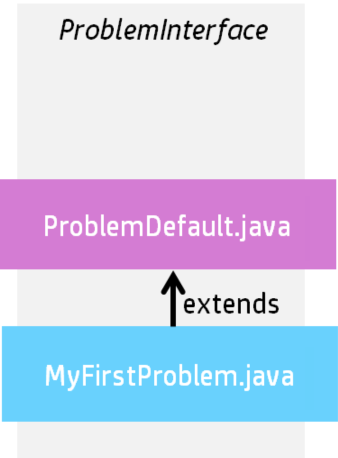


Figure 22 - One problem specific customization

ProblemDefault.java is the class implementing the methods of the ProblemInterface as seen in Figure 22. It defines the default behavior of Problem Detection Value Packs.

The way to override a method of the ProblemInterface is to create a customization class per problem, which extends ProblemDefault.

Below 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.getLogger
(Problem_Skeleton.class));
                                                       }

        }
```

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

The following equation must be true

Name of the customization class for problem X = name of problem X as defined in filters file.

For example, if the extract of `ProblemDetection_filters.xml` is like this:

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

Then the extract of `Problem_LOS.java` must look like this:

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

Below is the same file renamed as MyFirstProblem.java, which overrides both the `computeProblemEntity()` and `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.getAttributeUsedForKeyDuringRecognition
());

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)";

        }
```

Which overridable methods will be called depending on the lifecycle of the alarm and depending on the problem and its context.

The Problem Detection framework will automatically invoke the methods whatToDoWhenXXX(...) listed in section 7.1, at precise times of the lifecycle of every alarm.

For instance, when an alarm 'alm1' is cleared, the Problem Detection framework will invoke the method whatToDoWhenXXXAlarmIsCleared(alm1...)

If 'alm1' belongs to only one problem "Problem A", then the Problem Detection framework will invoke the method whatToDoWhenXXXAlarmIsCleared(alm1 ...) present in the customization class of "Problem A" . If the method whatToDoWhenXXXAlarmIsCleared( ) has not been overridden for "Problem A", the default method is invoked.

But if 'alm1' **also** belongs to "Problem B", the Problem Detection framework will **also** invoke 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 lifecycle at a given time, the Problem Detection framework will decide exactly which exact method(s) whatToDoWhenXXX(..) to invoke.

In the above example, suppose 'alm1' belongs to "Problem A" and "Problem B", and that 'alm1' at the moment it gets cleared, is
- 'subalarm' for "Problem A"
- 'orphan alarm' for "Problem B".

Then the methods

whatToDoWhenSubAlarmIsCleared(alm1) will be called for "Problem A"

whatToDoWhenOrphanAlarmIsCleared(alm1) will be called for "Problem B"

An orphan alarm for a given problem is an alarm that does not belong to any group of the given problem.

A Candidate alarm for a given problem is an alarm that belongs to a group of the given problem, but the problem alarm of this group has not yet come.

A Sub alarm for a given problem is an alarm that belongs to a group of the given problem, and the problem alarm of this group has come.

Figure 23 below shows a graphical representation of the methods that will be invoked based on the lifecycle of the alarm.

In

Figure 23, there are 3 alarms, 'a1', 'a2' and 'a3'

- '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 problem 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 'Orphan' for "Problem B". At this time the consolidated role of 'a1' across all problems will be 'SubAlarm'. This consolidated role will be placed in the "Pb" field of the alarm

Context of Problem A

a1 — alarm having a potential role for Problem A

a1				
Qualifier for Problem A		Unnknown	Candidate	SubAlarm
methods to invoke		whatToDoWhen**Orphan**AlarmXXXX()	whatToDoWhen**Sub**AlarmXXX ( )	whatToDoWhen**Sub**AlarmXXXX ( )

a2 — trigger alarm for Problem A

a2				
Qualifier for Problem A			Trigger	Trigger
methods to invoke			whatToDoWhen**Sub**AlarmXXX ( )	whatToDoWhen**Sub**AlarmXXXX ( )

a3 — problem alarm for Problem A

a3				
Qualifier for Problem A				ProblemAlarm
methods to invoke				whatToDoWhen**Problem**AlarmXXX()

Context of Problem B

a1 — alarm having a potential role for Problem B

a1				
Qualifier for Problem B		Unknown	Unknown	Unknown
methods to invoke		whatToDoWhen**Orphan**Alarm XXX( )	whatToDoWhen**Orphan**Alarm XXX( )	whatToDoWhen**Orphan**Alarm XXX( )

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

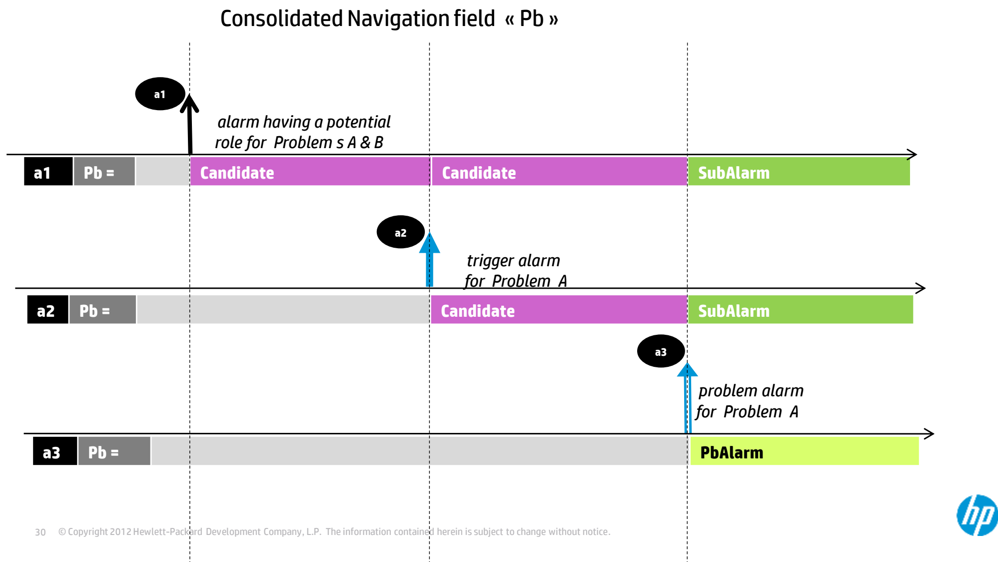


Figure 23 - Consolidation of alarm's qualifiers

7.4.3 My ProblemDefault

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

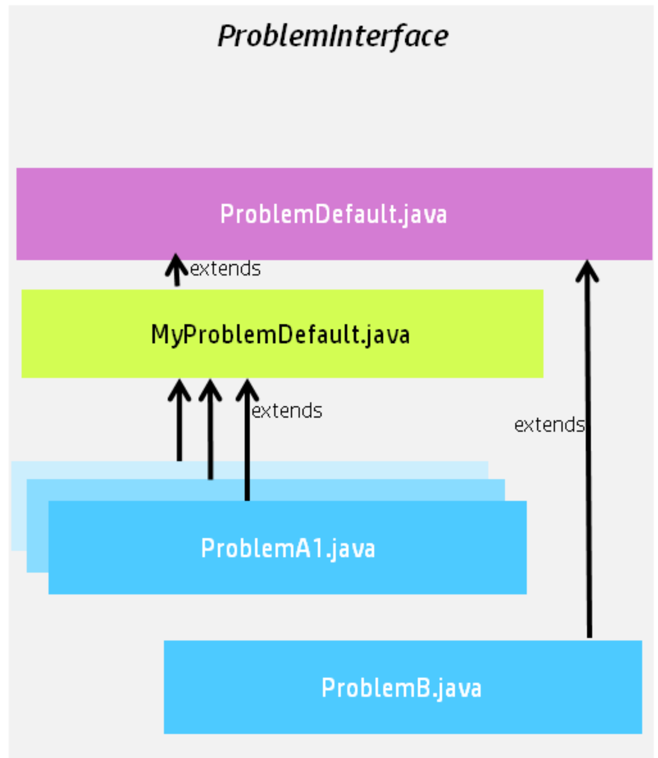


Figure 24 - MyProblemDefault: a customization for a group of problems

In the diagram above MyProblemDefault.java implements some or all of the methods of ***ProblemInterface***. Each problem customization class that extends MyProblemDefault.java will benefit from the implementation of those methods. In the diagram, by default, ProblemA1, ProblemA2 (hidden behind ProblemA1) and ProblemA3 (hidden behind ProblemA1) will use the methods implemented in MyProblemDefault.java. This happens only because the different propagation java classes ProblemA1 to A3 explicitly extended in their java code the MyProblemDefault. ProblemB will use the methods implemented in ProblemDefault.java, unless these methods are overridden in ProblemB.java

For a comprehensive diagram showing the advanced possibilities and subtleties of using extensions of Problemdefault.java, refer to Annex 0.

7.4.4 Problems initialization starting V3.2

Initialization of problems defined inside the `<problemPolicy>` tag in the *ProblemXmlConfig.xml* file has changed starting V3.2.

Until V3.2, for a problem defined in the *ProblemXmlConfig.xml* file with only part of the problem policies as described in section 5.3.2.2 The Problem Specific Policies, the other policies are given default values. ProblemDefault (can be

MyProblemDefault) configuration is used only to initialize a problem whose policy 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 given in the ProblemDefault.java class.

Starting V3.2, all the policies defined in the *ProblemDefault* problem policy (can be MyProblemDefault) are applied to all the other Problems, if not overwritten by their respective custom problem policy. Furthermore, for the policies seen 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 the other Problems, so even if defined in a custom problemPolicy they are added to the ones defined in the sequence, and not overwritten. Therefore, if wanting specific behavior for each of the Problems, it is better to empty the ProblemDefault configuration and defined it in each of the custom problem policies. On the other hand, it is a good tip to identify what is common to all problems and define it only once in the ProblemDefault configuration.

What has not changed comparing to V3.1 is that ProblemDefault (can be MyProblemDefault) configuration is also used to completely initialize a problem whose policy 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 given in the ProblemDefault.java class.

In the following example of configuration in V3.1 the Strings for ProblemDefault, Problem_Synch, Problem_BitError and Problem_Power are the same but have to be written for each of them. Also, the Booleans defined in ProblemDefault siteDown is 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). We observe also that the delayForProblemAlarmCleareance 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 the one 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 we transform this configuration file in V3.2 and considering that these are all the problems who have their top filter defined in the ProblemDetection_filters.xml (or if there are other, then all the characteristics defined in ProblemDefault policy apply on them), then we obtain the following lighter file:

```
…
<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 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.

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

For example, the behavior, in other words the things that are done, when a Problem Detection Value Pack is initialized is a "non-problem-specific" behavior.

The way to customize a "non-problem-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.pd.core]*.
- Ensure that the value of the property *generalBehaviorClassName* in the file *context.xml* in *src/main/resources/valuepack/conf/* folder matches MyGeneralBehavior , *a*s shown in *Figure 25 – PD MyGeneralBehavior name* matching
- Override the methods of the **GeneralBehaviorInterface** for which the behavior has to be customized.

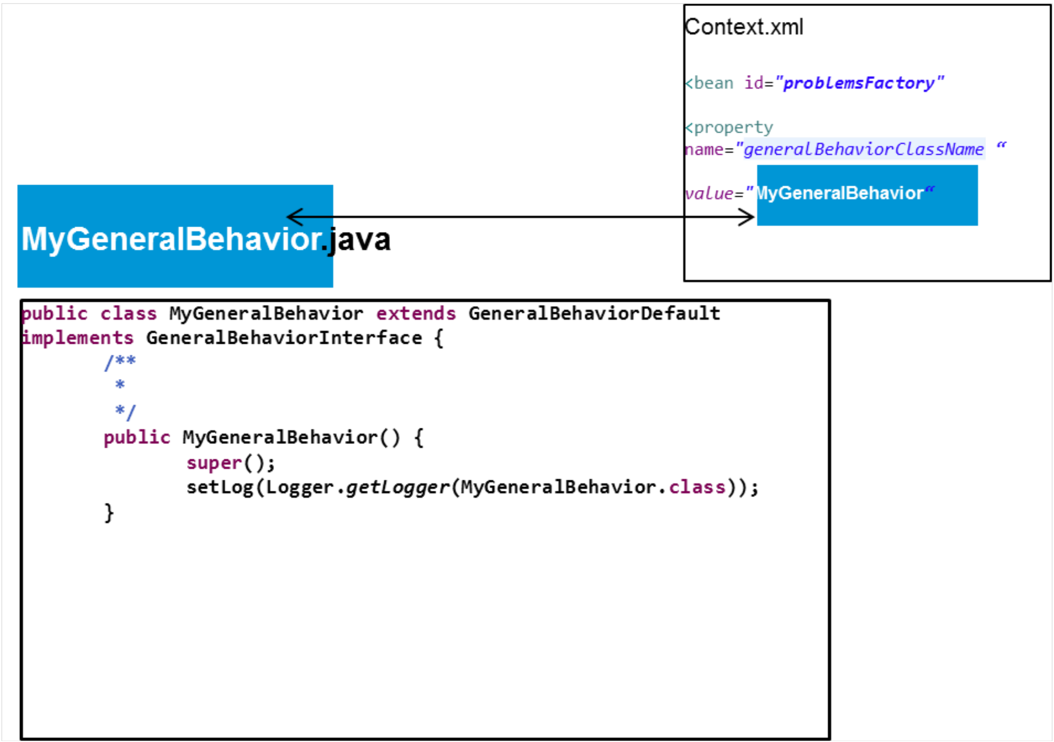


Figure 25 – PD MyGeneralBehavior name matching

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

```java
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

There are three ways to enrich alarms in Problem Detection

Through UCA-EBC lifecycle, synchronous enrichment is possible. Refer to UCA for EBC 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 synchronous, if the method **isInformationNeededAvailable()** is overridden with synchronous code.

This enrichment can be asynchronous, if the method **isInformationNeededAvailable()** is overridden with asynchronous code.

The enrichment is called "**synchronous**" when the Problem Detection value pack waits for the enrichment of the alarm to be completed before to proceed with the alarm processing.

The enrichment is called "**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

Example One time enrichment "independent of all problems"

The example below shows the method whatToDoWhenNewAlarmIsJustInserted () being overridden. The method adds a new custom field in 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 example below 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 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);
  }
 }
}
```

Example of code 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);

}
```

Chapter 8

Advanced features of Topology State Propagator

Once configured (see 5.4), a TSP Value Pack runs with a standard 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, TSP offers the flexibility to change the default behavior.

The default behavior is presented in section 8.1.

The ways to customize the default behavior are described in section 8.1.12.

8.1 The default behavior explained

In this section the default behavior is presented, as well as the many overridable methods available for the value pack developer to customize it.

As Problem Detection, the Topology State Propagator Framework is a set of Java libraries, with some Java classes that can be extended and methods overridden in order to change the default behavior of TSP Value Packs.

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

The default behavior of all these methods is available by consulting the javadoc. The implementation code of these methods is available in the example value pack delivered as part of the TSP Dev Kit. The code of each of these methods is executed for every propagation and can be overridden by the value pack developer.

Firstly, an example is presented in 8.1.1 and secondly the different interfaces available.

8.1.1 Example

An example of how the workflow of the different methods triggered in the case of an alarm Operator State Update is shown in the sequence diagram in Figure 26, where an 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 for any of Propagation3's groups.

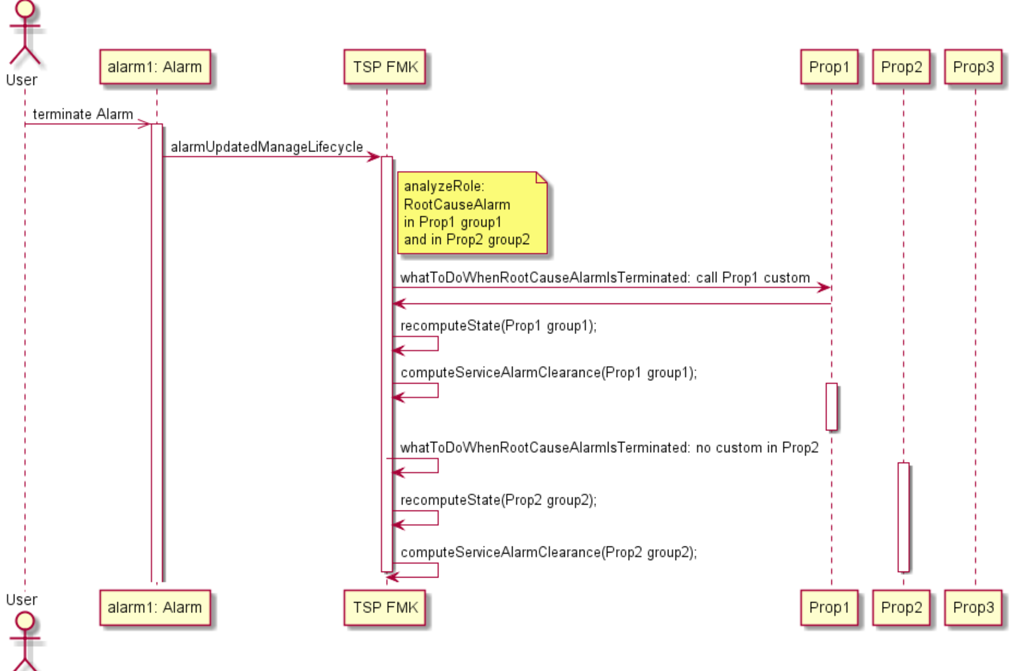


Figure 26 Alarm termination sequence diagram example

Let's consider for our example the simple topology from Figure 27 where Prop1 and Prop2 have no connection between each other and Prop3 can be impacted by Prop2 and vice versa, where Prop3 is a finer grain propagation then Prop2.

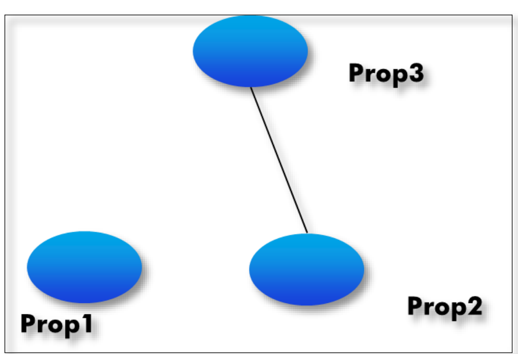


Figure 27 Topology of the example

Let's say that at before the alarm1 termination is received, the propagation groups are as shown in Figure 28 TSP: Alarm termination example: TSP group updates Step1.

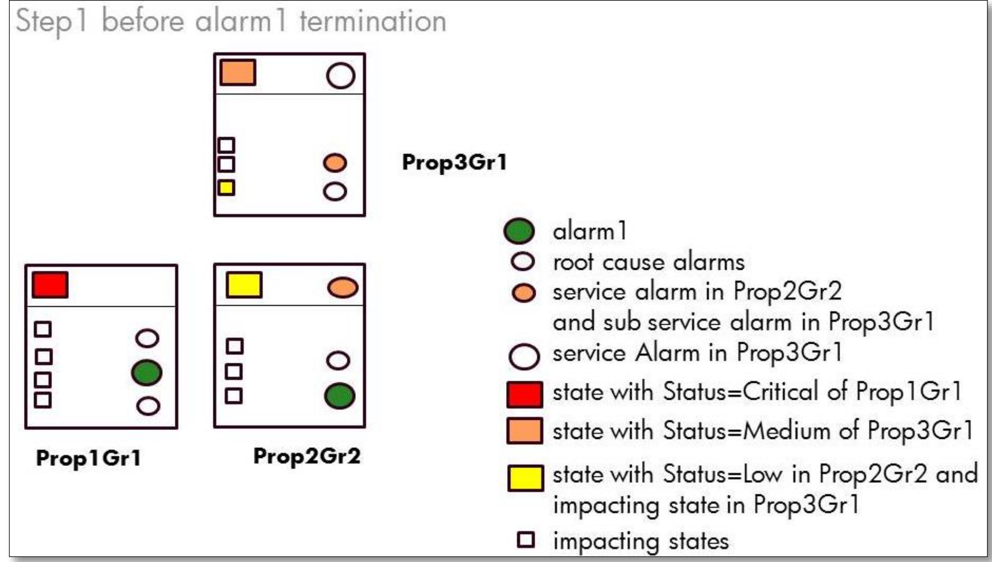


Figure 28 TSP: Alarm termination example: TSP group updates Step1

The alarm1 termination is received and, according to the sequence diagram in Figure 26, a number of methods will be called by the TSP framework. If we assume that alarm1 has a role in the computation of all states in all groups, it will result in their status change. We can also assume that alarm1 no impact in service Alarms computation. Then the propagation groups could look like represented in Figure 29 TSP: Alarm termination example: TSP group updates Step2.

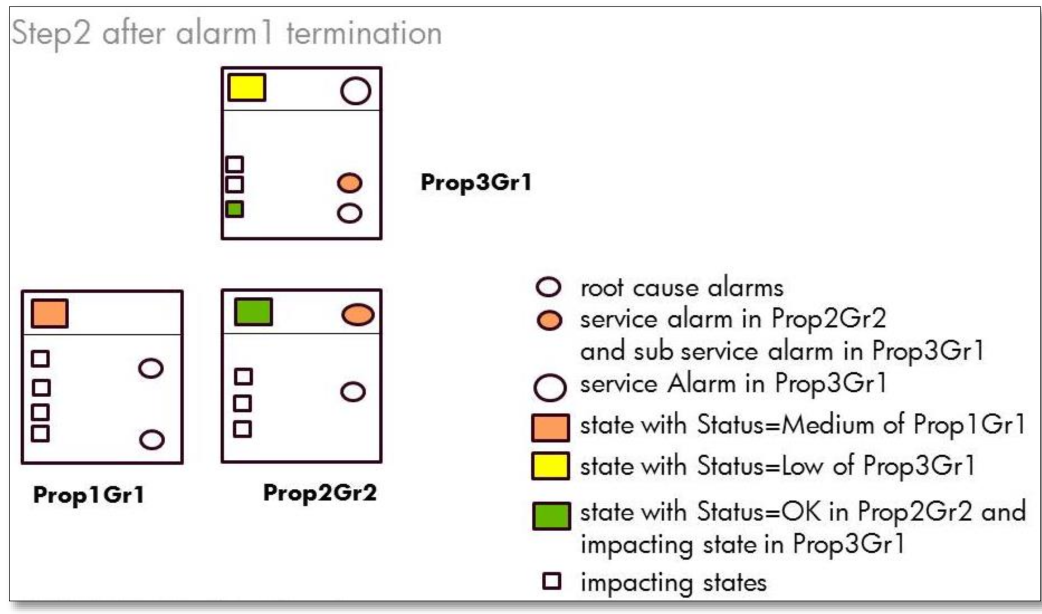
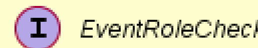


Figure 29 TSP: Alarm termination example: TSP group updates Step2

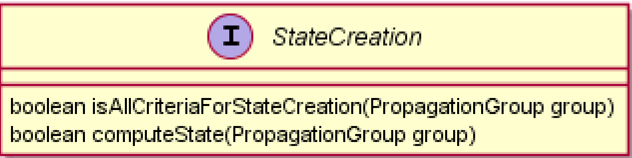
8.1.2 Event Role Check



boolean isMatchingRootCauseAlarmCriteria(Alarm a, PropagationGroup group)
boolean isMatchingSubRootCauseAlarmCriteria(Alarm a, PropagationGroup group)
boolean isMatchingImpactingStateCriteria(State state, PropagationGroup group)
boolean isMatchingServiceAlarmCriteria(Alarm a, PropagationGroup group)
boolean isMatchingSubAlarmCriteria(Alarm a, PropagationGroup group)

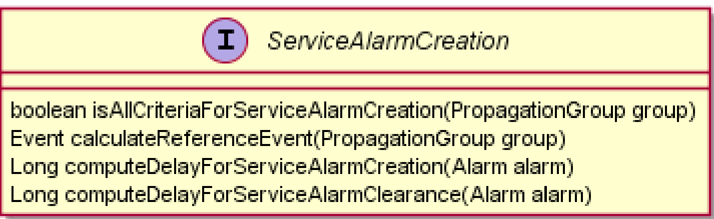
8.1.3 State Creation

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



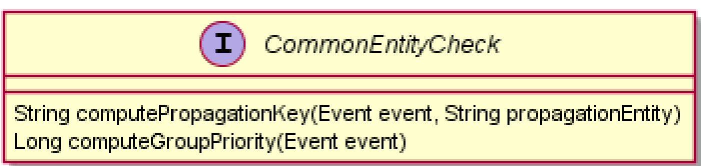
8.1.4 Service Alarm Creation and Clearance

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



8.1.5 Common Entity Check

Methods used to calculate Information for optimizations



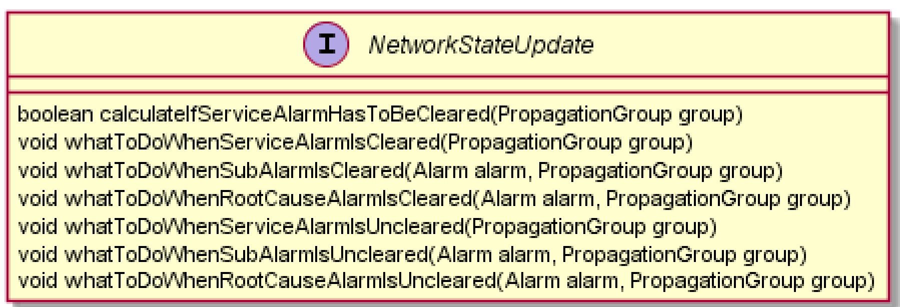
8.1.6 PropagationGroup update

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

PropagationGroupUpdate

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)

8.1.7 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

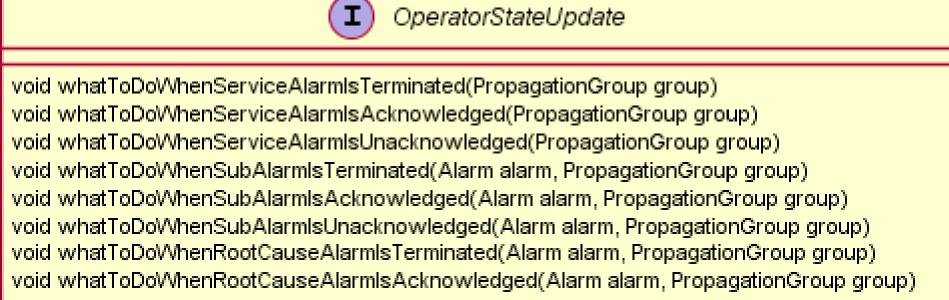
PropagationDefault

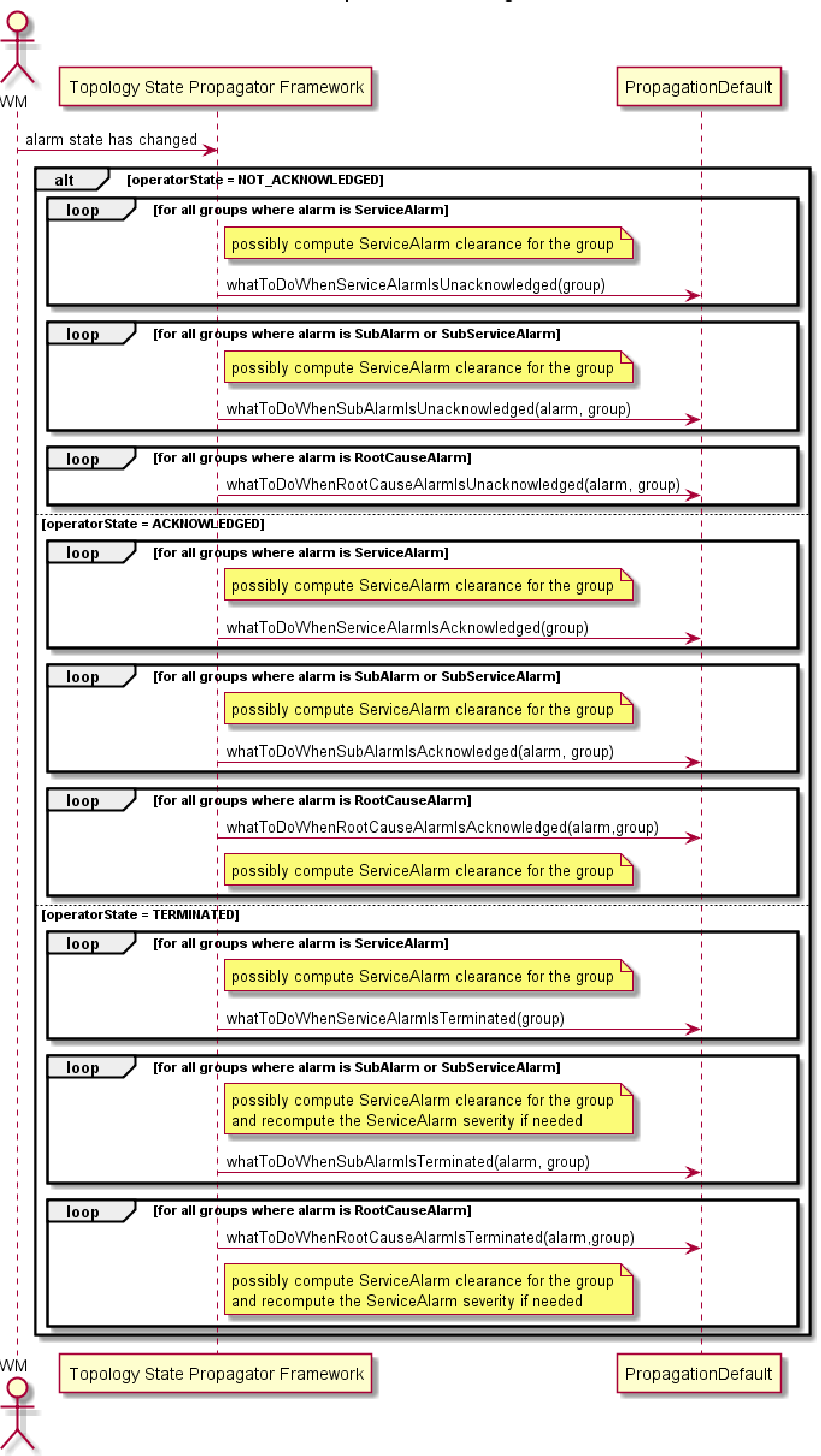
8.1.8 Operator State Update

Methods used to manage the lifecycle of a

- ServiceAlarm
- SubAlarm
- RootCauseAlarm

And its consequence





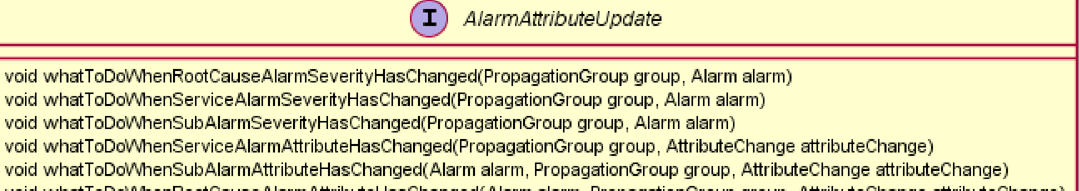
Alarm Operator State Changes
WM
Topology State Propagator Framework
PropagationDefault
alarm state has changed
alt
[operatorState = NOT_ACKNOWLEDGED]
loop
[for all groups where alarm is ServiceAlarm]
possibly compute ServiceAlarm clearance for the group
whatToDoWhenServiceAlarmIsUnacknowledged(group)
loop
[for all groups where alarm is SubAlarm or SubServiceAlarm]
possibly compute ServiceAlarm clearance for the group
whatToDoWhenSubAlarmIsUnacknowledged(alarm, group)
loop
[for all groups where alarm is RootCauseAlarm]
whatToDoWhenRootCauseAlarmIsUnacknowledged(alarm, group)
[operatorState = ACKNOWLEDGED]
loop
[for all groups where alarm is ServiceAlarm]
possibly compute ServiceAlarm clearance for the group
whatToDoWhenServiceAlarmIsAcknowledged(group)
loop
[for all groups where alarm is SubAlarm or SubServiceAlarm]
possibly compute ServiceAlarm clearance for the group
whatToDoWhenSubAlarmIsAcknowledged(alarm, group)
loop
[for all groups where alarm is RootCauseAlarm]
whatToDoWhenRootCauseAlarmIsAcknowledged(alarm,group)
possibly compute ServiceAlarm clearance for the group
[operatorState = TERMINATED]
loop
[for all groups where alarm is ServiceAlarm]
possibly compute ServiceAlarm clearance for the group
whatToDoWhenServiceAlarmIsTerminated(group)
loop
[for all groups where alarm is SubAlarm or SubServiceAlarm]
possibly compute ServiceAlarm clearance for the group
and recompute the ServiceAlarm severity if needed
whatToDoWhenSubAlarmIsTerminated(alarm, group)
loop
[for all groups where alarm is RootCauseAlarm]
whatToDoWhenRootCauseAlarmIsTerminated(alarm,group)
possibly compute ServiceAlarm clearance for the group
and recompute the ServiceAlarm severity if needed
WM
Topology State Propagator Framework
PropagationDefault

8.1.9 Alarm Attribute Update

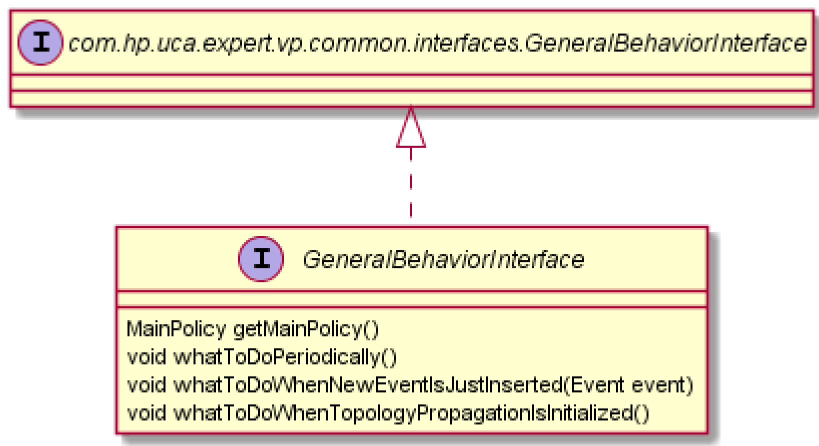
Methods used to manage the Severity or an Attribute Update of

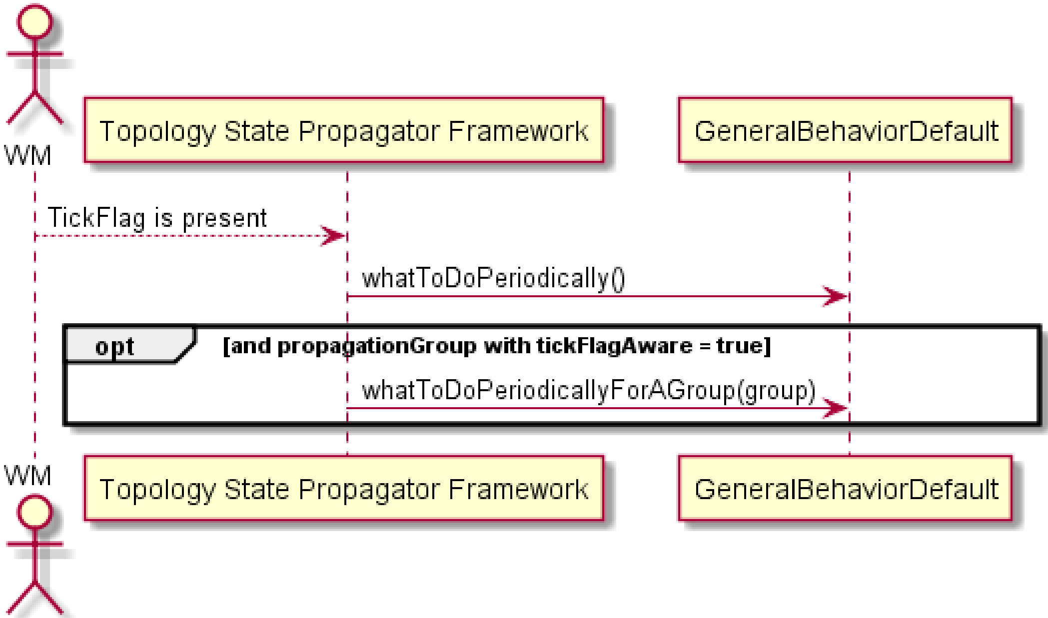
- ServiceAlarm
- RootCause Alarm
- SubAlarm

, and its consequence

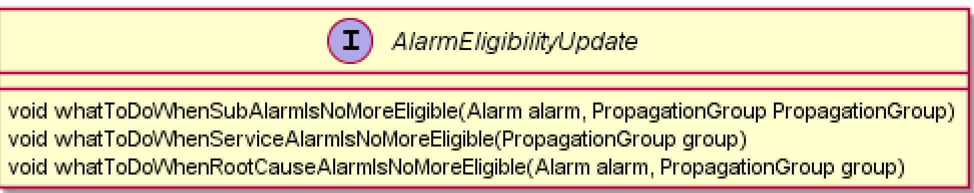


8.1.10 Periodic Check and General Behavior



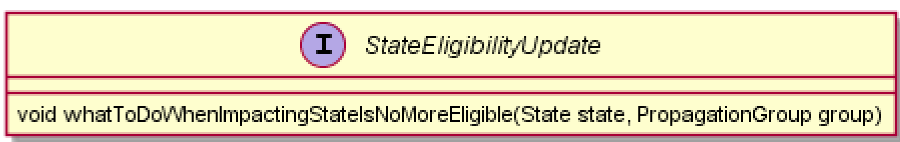


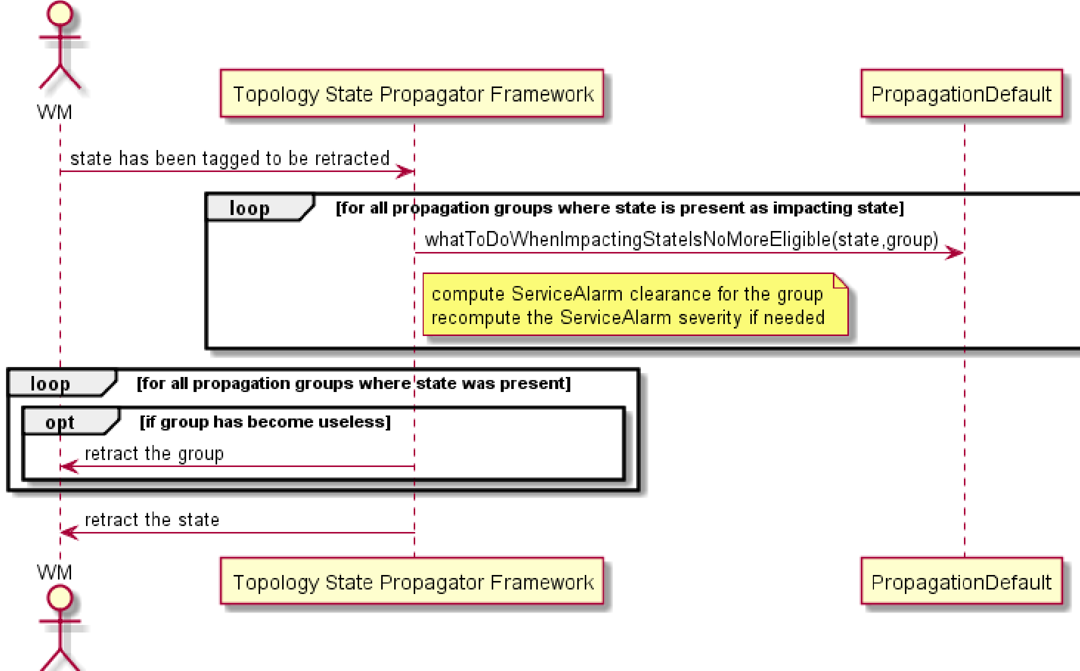
8.1.11 Alarm eligibility update



Alarm Eligibility Update

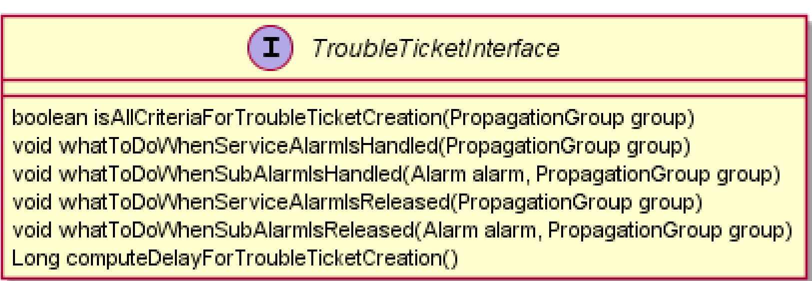
8.1.12 State eligibility update





8.1.13 TroubleTicket update

Methods used to manage the Trouble Ticket lifecycle when related to a ServiceAlarm or a SubAlarm, and its consequence



8.2 How to customize default behavior

A TSP VP delivered as an example with the IM SDK is described in Annex E.

8.2.1 Java customization

The way to customize the default behavior of Topology State Propagator Value Packs is to override some of the Java methods listed in section 8.1. There are three levels of customization:

- Per propagation (this section)
- For a set or for all propagations (section 8.2.2)
- For non-propagation specific matters (section 8.2.3)

The methods that can be overridden to customize the “propagation specific” behavior of a Topology State Propagator Value Pack are all 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 all listed in the `GeneralBehaviorInterface` java interface.

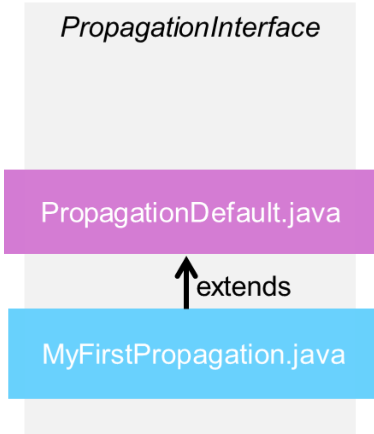


Figure 30 - One propagation specific customization

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

The way to override a method of the PropagationInterface is to create a customization class per propagation, which extends PropagationDefault as seen in Figure 30.

Below is 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, in the above example MyPropagation, must be changed to the name of the propagation for which we want to customize the behavior.

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 like this:

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

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

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

Below is the same file renamed as MyFirstPropagation.java, which 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 MyFistPropagation 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 precise times of the lifecycle of every alarm (and depending on propagation context).

For instance, when an alarm ‘alarm1’ is a root cause in ‘propagationGroup1’ and 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...) present in the customization class of “PropagationA1” . If the method whatToDoWhenRootCauseAlarmIsCleared ( ) has not been overridden for “PropagationA1”, the default method is invoked.

But if ‘alarm1’ **also** belongs 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 present in the customization class of “Propagation B”, or the default method otherwise.

Depending of the position of the alarm in its lifecycle at a given time, the Topology State Propagator framework will decide exactly which exact method(s) whatToDoWhenXXX(..) to invoke.

8.2.2 My PropagationDefault

As in the case of extending ProblemDefault for problems (seen in 7.4.3 My ProblemDefault), PropagationDefault class can also be extended. The benefit of extending PropagationDefault class is to modify the default behavior for all propagations or for a set of propagations.

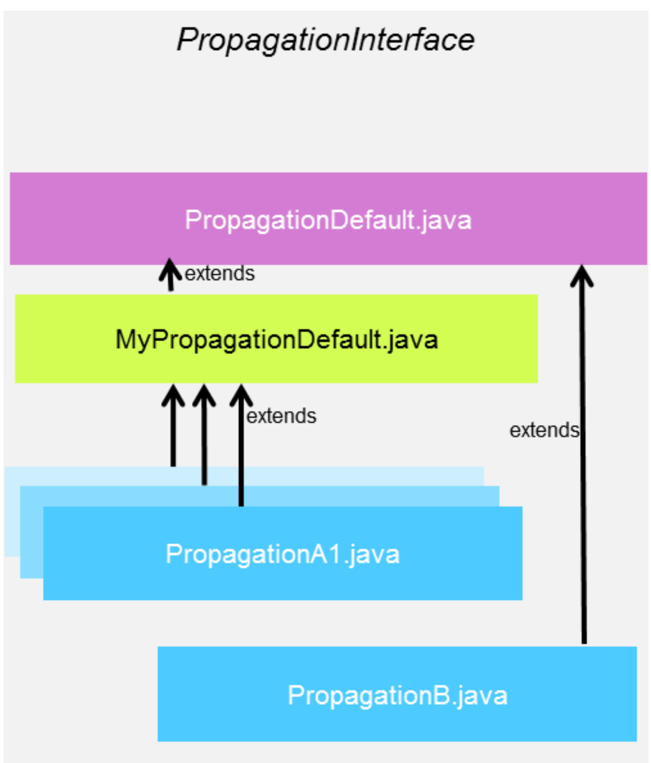


Figure 31 - MyPropagationDefault: a customization for a group of propagations

In Figure 31 MyPropagationDefault.java implements some or all of the methods of ***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 (both hidden behind PropagationA1) will use the methods implemented in MyPropagationDefault.java. This happens only because the different propagation java classes PropagationA1 to A3 explicitly 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 of extending PropagationDefault.java, refer to Annex F

Propagations initialization

Propagations are initialized from the *PropagationXmlConfig.xml* defined inside the `<propagationPolicy>` tag in the following way: all the policies defined in the *PropagationDefault* propagation policy (can be MyPropagationDefault) are applied to all the other Propagations, if not overwritten by their respective custom propagation policy. Furthermore, for the policies seen in Table 24 – TSP customized “per-propagation” configuration: Strings, Longs, Booleans (which contain a sequence of String, Long and Boolean types) defined in the PropagationDefault are valid for all the other Propagations, so even if defined in a custom propagationPolicy they are added to the ones defined in the sequence, and not overwritten. This applies also for topology policies: Nodes, PoiCategories and Threshold values seen in 5.4.2.2. Therefore, as for Problems, if wanting specific behavior for each of the Propagations, it is better to empty the PropagationDefault configuration and defined it in each of the custom propagation policies. On the other hand, it is a good tip to identify what is common to all propagations and define it only once in the PropagationDefault configuration.

PropagationDefault (can be MyPropagationDefault) configuration is used to completely initialize a propagation whose policy is not defined in *the PropagationXmlConfig.xml*, but 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, then the default values are applied as given in the PropagationDefault.java class.

In the following example of configuration the PropagationDefault policies will therefore apply for all the other propagations defined, like 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 of the 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>
…
```

8.2.3 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 32 – TSP MyGeneralBehavior name matching
- Override the methods of the **GeneralBehaviorInterface** for which the behavior has to be customized.

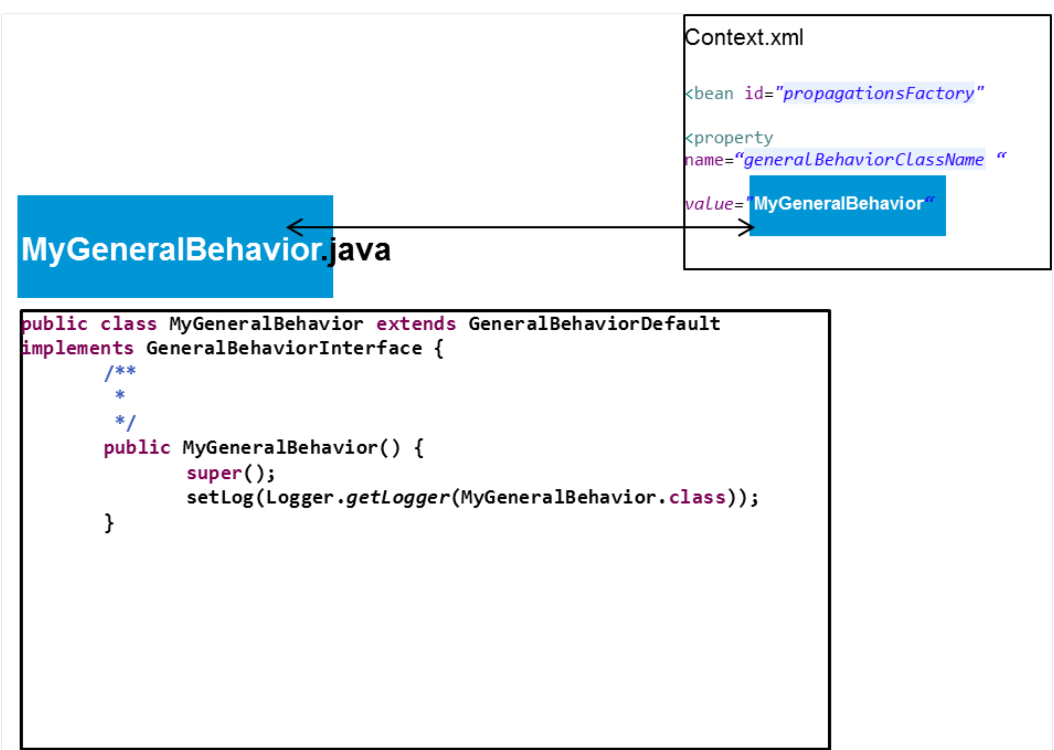


Figure 32 – TSP MyGeneralBehavior name matching

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

Like for any UCA for EBC Value Pack, the logging configuration for a Inference Machine Value Pack has to be done in the file
${UCA_EBC_INSTANCE}/conf/uca-ebc-log4j.xml
on the UCA for EBC server.

The list of specific IM loggers is given below:

Logger	Description
com.hp.uca.expert.vp.common.actions.db	Controls the executions of DB requests
com.hp.uca.expert.vp.common.actions.temip	Controls the executions of actions (and TroubleTicket actions) to TeMIP
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.ProblemProperties	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_AlarmRecognition	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

Logger	Description
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 subalarm to a group, creating a Trouble Ticket, ...)
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 Lifecycle
com.hp.uca.expert.vp.pd.services.PD_Service_ProblemAlarm	Controls Problem Detection Services for ProblemAlarm
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

Logger	Description
com.hp.uca.expert.vp.tp.propagation	Controls the customization of classes
com.hp.uca.expert.vp.tp.im.lifecycle	Controls TSP Internals Lifecycle
com.hp.uca.expert.vp.tp.services.TP_Service_Lifecycle	Controls TSP Services for Lifecycle
com.hp.uca.expert.vp.tp.services.TP_Service_ServiceAlarm	Controls TSP Services for ServiceAlarm
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 can be very useful to log with the following 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 from V3.1 to V3.2

PD 3.2 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, etc.), 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 in counterpart, it brings some improvements:

- the logic of actions is separated from PD and TSP
- as such, it is reusable easily (same ActionsFactory can be used across PD and TSP)
- easier to understand at the end

Deprecated APIs

All methods/classes/packages below are deprecated with this version and will be removed in next major update.

This is mainly due to the fact that most of the methods are now coming within uca-evp-common.jar that is used also by another toolkit (aka Topology State Propagator for Service Impact).

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)

Type	API	Deprecated by
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

How do I migrate my PD VP 3.0/3.1 to 3.2?

Problem Detection 3.2 **does not provide any automatic migration tool** for your Java files. However, the SDK provides an XLST (eXtensible Stylesheet Language Transformation) file that you can use to migrate your PD configuration file.

In your Java code

- **Removed classes**

Following imports will generate compilation errors because the classes do not exist anymore

Class (in V3.1)	Should be replaced in V3.2 by
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

Class (in V3.1)	Should be replaced in V3.2 by
import com.hp.uca.expert.vp.pd.config.TroubleTicketAction	import com.hp.uca.expert.vp.im.config.TroubleTicketAction
import com.hp.uca.expert.vp.pd.config.TroubleTicketActions	import com.hp.uca.expert.vp.im.config.TroubleTicketActions
import com.hp.uca.expert.vp.pd.core.exception.InvalidSupportedActions	import com.hp.uca.expert.vp.common.exceptions.InvalidSupportedActions
import com.hp.uca.expert.vp.pd.core.exception.InvalidSupportedTroubleTicketActions	import com.hp.uca.expert.vp.common.exceptions.InvalidSupportedTroubleTicketActions
import com.hp.uca.expert.vp.pd.interfaces.ActionsFactoriesSelection	import com.hp.uca.expert.vp.common.interfaces.ActionsFactoriesSelection
import com.hp.uca.expert.vp.pd.interfaces.SupportedActions	import com.hp.uca.expert.vp.common.interfaces.SupportedActions
Import com.hp.uca.expert.vp.pd.interfaces.SupportedTroubleTicketActions	import com.hp.uca.expert.vp.common.interfaces.SupportedTroubleTicketActions

- **What needs to be changed in your customized ProblemDefault**

If you are overriding the following methods from ProblemDefault, they need to be changed because they do not exist anymore:

Method (in V3.1)	Should be replaced in V3.2 by
chooseSupportedActions(Alarm alarm, ProblemInterface problem)	chooseSupportedActions(Event event, CommonActionInterface problemOrPropagation)
chooseSupportedTroubleTicketActions(Alarm alarm, ProblemInterface problem)	chooseSupportedTroubleTicketActions(Event event, CommonActionInterface problemOrPropagation)

- **What needs to be changed in your customized ActionsFactory**

If you are overriding the following methods from ActionsFactory, they need to be changed because they do not exist anymore:

Method (in V3.1)	Should be replaced in V3.2 by
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)

Method (in V3.1)	Should be replaced in V3.2 by
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)

- ### What needs to be changed in your customized TroubleTicketActionsFactory

If you are overriding the following methods from TroubleTicketActionsFactory, they need to be changed because they do not exist anymore:

Method (in V3.1)	Should be replaced in V3.2 by
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)

In your XML configuration

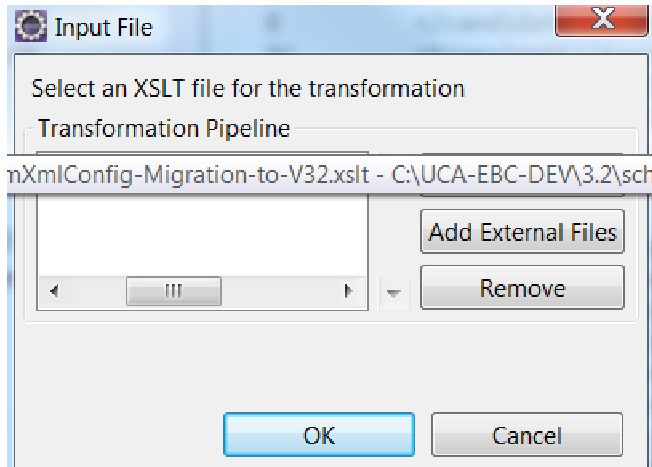
Your ProblemXMLConfig.xml file (or equivalent) needs to be modified to make use of the new namespace "**http://config.im.vp.expert.uca.hp.com/**" for certain elements of the file like:

- actions
- troubleTicketActions
- booleans
- longs
- strings

You can use the ProblemXmlConfig-Migration-to-V32.xslt file part of the Inference Machine SDK to transform your current ProblemXmlConfig.xml version 3.1 to version 3.2.

Within Eclipse, you can proceed as per following steps:

1. Select your ProblemXmlConfig.xml
2. Right-click and choose Run As -> XSL Transformation
3. Input File should be added by clicking Add External Files
4. Select the xslt file provided under ${UCA_EBC_DEV_HOME}/schemas



5. Click OK

If you have errors like '`Namespace for prefix 'p1' has not been declared`', it's probably because you're not using the right processor to transform your XML. In that 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.

PD Value Pack example

As part of the Inference Machine Development Kit, an example Value Pack project, named 'pd-example', is available.

If deployed, the pd-example Value Pack will be able to recognize four problems:

- Problem_BitError
- Problem_Synch
- Problem_Power
- XmlGeneric_Synch

Each of these four 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.

Examples of Alarm enrichment, Action Factory and Trouble Ticket Action Factory are also given. Also, sample tests file that can be run with JUnit are contained. Those 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.

pd-example, content of src/main/java

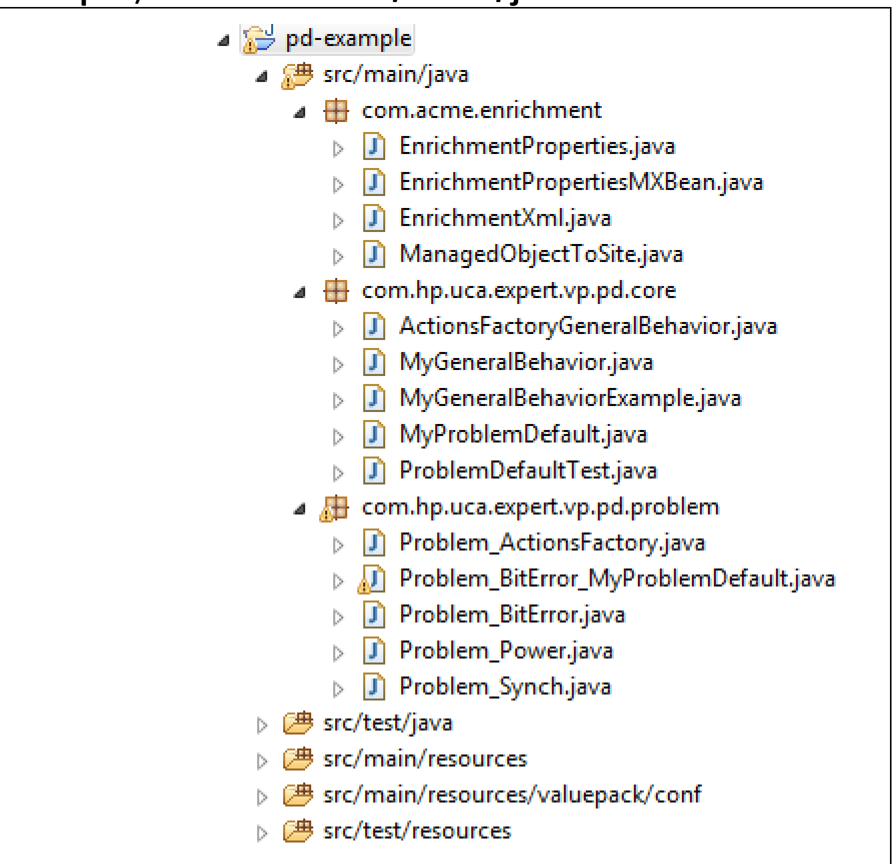


Table 27 - src/main/java: the customization code for the example Value Pack

Package com.acme.enrichment

This package contains classes used to read an XML file called *Enrichment.xml* present in src/main/resources/valuepack/conf. *Enrichment.xml* contains information to enrich alarms. It is a kind of table where if you know the managedObject of an alarm, then you can find the associated site.

Extract of Enrichment.xml

```
      <managedObjectToSite>
    <managedObject>motorola_omcr_system [...] 5 btssitemgr 0 msi 18 mms
0</managedObject>
    <site>bsc khorfakkan_bsc24 bts bridippm_6185</site>
      </managedObjectToSite>
```

The file *MissingInfoAlarmPowerTest.java* present in src/test/java/ft/enrichment is the test file sending alarms belonging to problem 'Problem_Power' and that need to be enriched with site information

EnrichmentProperties.java is the class that contains method to read the *Enrichment.xml* file.

EnrichmentPropertiesMXBean.java is the interface implemented by *EnrichmentProperties.java*

EnrichmentXml.java and *ManagedObjectToSite.java* are data structure to store the enrichment information.

Package com.hp.uca.expert.vp.pd.core

ActionsFactoryGeneralBehavior.java contains an example of method whatToDoWhenAlarmIsJustInserted() being overridden to do enrichment.

MyGeneralBehavior.java & *MyGeneralBehaviorExample.java* also contain examples of methods of the GeneralBehaviorInterface being overridden. See 7.4.5

`MyProblemDefault.java` illustrates methods of the ProblemInterface being overridden for a subset of problems. See 7.4.3

Package com.hp.uca.expert.vp.pd.problem

Problems' customizations

In `src/main/java`, problems' customization classes are available in package `com.hp.uca.expert.vp.pd.problem`.

pd-example has four main problems. Out of these four problems, have been customized by writing Java code: Problem_BitError, Problem_Synch, Problem_Power, and one has been customized by writing XML (in `src/main/resources/valuepack/conf/ProblemXmlConfig.xml`): XmlGeneric_Synch

File	overrides
Problem_BitError.java	calculateProblemAlarmAdditionalText computeProblemEntity isAllCriteriaForProblemAlarmCreation
Problem_Sync.java	Same as Problem_BitError + calculateProblemAlarmEventTime
Problem_Power.java	Same as Problem_BitError + calculateProblemAlarmSeverity isInformationNeededAvailable isMatchingProblemAlarmCriteria
Problem_BitError_MyProblemDefault.java	Same as Problem_BitError + calculateProblemAlarmSeverity
Problem_ActionsFactory.java	Same as Problem_BitError + isMatchingSubAlarmCriteria isMatchingTriggerAlarmCriteria

pd-example, content of src/test/java

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.

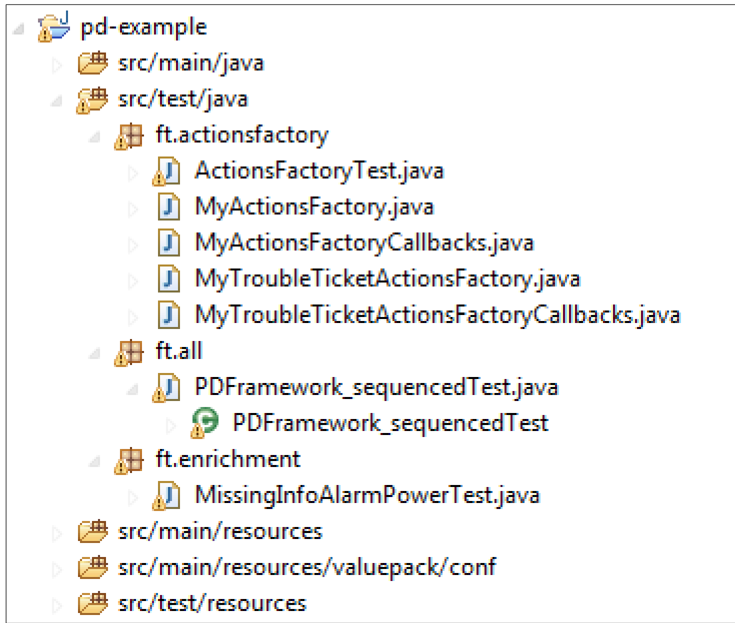


Table 28 - src/test/java: the source code of the tests

Package ft.actionsfactory

A Problem Detection Value Pack receives alarms from a Network Management System (NMS), does some processing, and has to ask the NMS to execute some actions. The list of actions that are supported is present in the SupportedActions java interface. The SupportedActions interface defines methods such as *createProblemAlarm(), terminateAlarm(), clearAlarm(),* …
The ActionsFactory.java class is a nutshell implementation of the SupportedActions interface.

Problem Detection provides TeMIPActionsFactory.java, a real implementation of SupportedActions for the case the NMS is TeMIP.
For cases where the NMS is not TeMIP, it is required to write an implementation of the SupportedActions interface on the model of the *MyActionsFactory.java*.

MyActionsFactoryCallback.java contains the callbacks methods that the NMS must call after executing some of the actions.

A Problem Detection Value Pack may also need to 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 nutshell implementation of the SupportedTroubleTicketActions interface.

Problem Detection provides TeMIPTroubleTicketActionsFactory.java, a real implementation of SupportedTroubleTicketActions for the case the trouble ticketing system is HP Service Manager (accessed through TeMIP)

For cases where the trouble ticketing system is not HP Service Manager, it is required to write an implementation of the SupportedTroubleTicketActions interface on the model of the *MyTroubleTicketActionsFactory.java*

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 the sending of some alarms and then checks that the necessary actions have been emitted.

Package `ft.all`

PDFramework_sequencedTest.java is a test file. It sends alarms corresponding to the four problems Problem_BitError, Problem_Synch, Problem_Power and XmlGeneric_Synch. It checks that problems are detected, that Problem Alarms are created, that sub-alarms are tagged, that number of groups created is correct and that number of actions executed is correct.

Package ft.enrichment

MissingInfoAlarmPowerTest.java is a test file. It sends alarms that need to be enriched. It checks that the enrichment was successful.

pd-example, content of src/main/resources

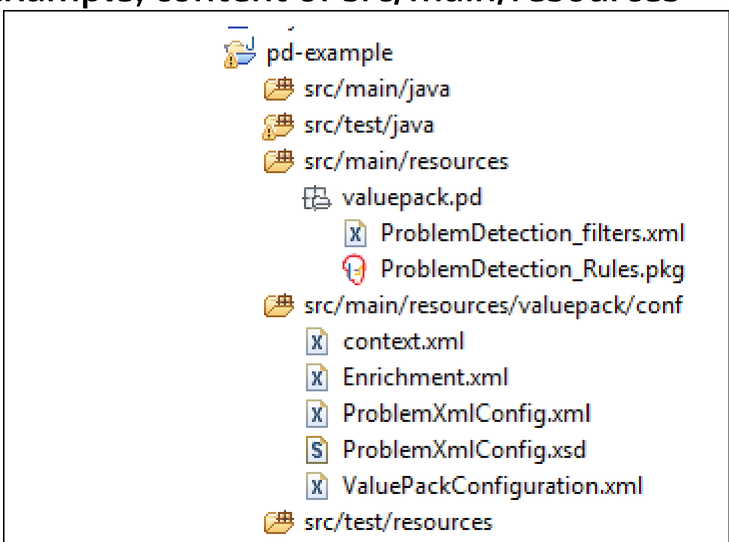


Table 29 - src/main/resources: the configuration files of the example Value Pack

Filters

Available in src/main/resources/valuepack/pd/*ProblemDetection_filters.xml*

There are the topFilters corresponding to the four problems:

- Problem_Synch
- Problem_Power
- Problem_BitError
- XmlGeneric_Synch

```
<topFilter name="XmlGeneric_Synch">
<topFilter name="Problem_Synch">
<topFilter name="Problem_Power">
<topFilter name="Problem_BitError">
```

Rules

Hidden under src/main/resources/valuepack/pd/*ProblemDetection_Rules.pkg*

Configuration

Files located in src/main/resources/valuepack/conf

context.xml → This file can be used to declare that the Problem Detection Value Pack pd-example relies on a customization of the GeneralBehavior

Enrichment.xml → This file contains data to enrich alarms belonging to Problem_Power

ProblemXmlConfig.xml → This file 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 → The XML schema of *ProblemXmlConfig.xml*

ValuePackConfiguration.xml → This file is used to define the configuration of the Value Pack and its Scenarios, the scenario policies, and the mediation flows

pd-example, content of src/test/resources

- pd-example
  - src/main/java
  - src/test/java
  - src/main/resources
  - src/main/resources/valuepack/conf
  - src/test/resources
    - com.hp.uca.expert.vp.pd.core
      - ProblemDefault.java
    - ft.actionsfactory
      - ActionsFactoryTest-context.xml
      - Alarms.xml
      - ProblemDetection_filters_ActionsFactory.xml
      - ProblemXmlConfig_ActionsFactory.xml
      - ValuePackConfiguration_ActionsFactory.xml
    - ft.all
      - Alarms_all_problems.xml
      - Alarms_BitError_T1.xml
      - Alarms_BitError_T2.xml
      - Alarms_Power_T1.xml
      - Alarms_Power_T2.xml
      - Alarms_Power_T3.xml
      - Alarms_Synch_T1.xml
      - Alarms_Synch_T2.xml
      - Alarms_XmlGeneric_Synch_T1.xml
      - PDFramework_sequencedTest-context.xml
    - ft.enrichment
      - Alarms_power_only.xml
      - MissingInfoAlarmPowerTest-context.xml
    - ActionRegistry.xml
    - ActionRegistry.xsd
    - log4j.xml
    - uca-ebc.properties

Table 30 - src/test/resources: the tests configuration files

`com.hp.uca.expert.vp.pd.core`

ProblemDefault implementation

Located in `src/test/resources/com/hp/uca/expert/vp/pd/core/`

`ft.actionsfactory`

Each JUnit test can run with a specific configuration for the Value Pack. For example the JUnit test file named ActionsFactoryTest.java, will use *ActionsFactoryTest-context.xml* (name must be <test file name>-context.xml) as context file.

This context file points at *ProblemXmlConfig_ActionsFactory.xml*, which is the policies configuration file, and at *ValuePackConfiguration_ActionsFactory.xml*,

which is the main Value Pack configuration file which in turns points to *ProblemDetection_filters_ActionsFactory.xml*, which is the filters file

Alarms.xml is the file describing the simulated alarms that will be sent by the test ActionsFactoryTest.java.

`ft.all`

This package contains all the alarms files used by JUnit test file PDFramework_sequencedTest.java. The JUnit test file PDFramework_sequencedTest.java sends alarms from each alarms file one by one, in sequence.

It would be possible to send all alarms simultaneously by using the file *Alarms_all_problems.xml*

- *Alarms_BitError_T1.xml* → alarms belonging to Problem_BitError and grouped in a group different from the group where alarms coming from Alarms_BitError_T2.xml will be gathered
  Alarms_BitError_T2.xml → alarms belonging to Problem_BitError and grouped in a group different from the group where alarms coming from Alarms_BitError_T1.xml will be gathered
- *Alarms_Power_T1.xml* → 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 will be gathered
  Alarms_Power_T2.xml → 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 will be gathered
  Alarms_Power_T3.xml → 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 will be gathered
- *Alarms_Synch_T1.xml* → alarms belonging to Problem_Synch and grouped in a group different from the group where alarms coming from Alarms_Synch_T2.xml will be gathered
  Alarms_Synch_T2.xml → alarms belonging to Problem_Synch and grouped in a group different from the group where alarms coming from Alarms_Synch_T1.xml will be gathered
- *Alarms_XmlGeneric_Synch_T1.xml* → alarms belonging to problem XmlGeneric_Synch

- *PDFramework_sequencedTest-context.xml* →the context file of PDFramework_sequencedTest.java test file

`ft.enrichment`

- *Alarms_power_only.xml* → the alarms file containing alarms sent by MissingInfoAlarmPowerTest.java
- *MissingInfoAlarmPowerTest-context.xml*→the context file of MissingInfoAlarmPowerTest.java test file.

Like any 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, namely: OSS Open Mediation V6.0.

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.

UCA for EBC Value Pack scenarios use web services to communicate with the Action Service web service of a Channel Adapter, typically the 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 please refer to the [R11] UCA for EBC Administration, Configuration and Troubleshooting Guide, and in particular to the 'uca-ebc.properties file configuration' chapter.

`ActionRegistry.xsd`

is 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 UCA -EBC Server. This file generally does not need to be modified. Please refer to the [R11] UCA for EBC Administration, Configuration and Troubleshooting Guide, and in particular to the 'ActionRegistry.xml file configuration' chapter

Annex C.

PD Advanced customization

Problem Detection behavior customization

As seen in chapter 7.4 it is possible to modify the default behavior of Problem Detection Value Packs.

The behavior can be modified

- per problem
- per family of problems
- for all problems
- for non problem specific matters

Per problem

Modifying the behavior of Problem Detection for one given problem, is done through overriding some of the methods of the *ProblemInterface* in the problem's customization class.

Per family of problems

Modifying the default behavior of Problem Detection for a set of problems, is done in two steps:

1st step -- creation of a MyFamilyOfProblems (this name is given as an example) customization class that implements some overriden methods of the *ProblemInterface.*

2nd step – for each problem in the family, creation of the problem's customization class that extends the MyFamilyOfProblems customization class.

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 all problems' customization class must extend one "MyAllProblemsDefault" (this name is given as an example) class

For non problem specific matters

Problem Detection framework offers the possibility to modify some behaviors not linked to any problem, through the creation of a customization class like MyGeneralBehavior (name is given as an example), and overriding methods of the *GeneralBehaviorInterface* interface such as `whatToDoWhenProblemDetectionIsInitialized( ),` `whatToDoWhenNewAlarmIsJustInserted( )`

It is also required to modify the `context.xml` file in the `src/main/resources/valuepack/conf/` folder to tell Problem Detection that

the customized implementation of the methods of the GeneralBehaviorInterface have to be found in **and only in** MyGeneralBehavior class. It is therefore pointless to override any GeneralBehaviorInterface method anywhere else other than in the class specified in the `context.xml` file.

GeneralBehaviorInterface defines methods such as "`whatToDoWhenProblemDetectionIsInitialized()`" that are not specific to any problem, and are not invoked by the Problem Detection framework on a problem object. It is therefore useless to provide an implementation of those methods in the class of customization of the problems.

The figure below shows an example of

- a "*per problem*" customization => Problem1.java
- a "*per family of problems*" customization => MyFamilyOfProblems.java for Problem2 & Problem3
- a "*non problem specific*" customization => MyGeneralBehavior.java

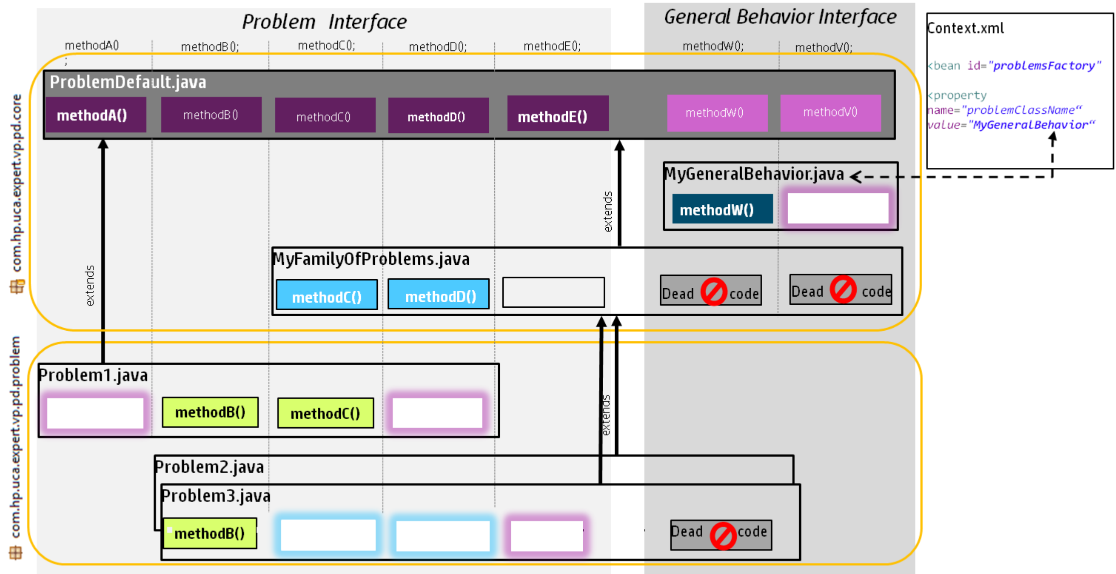
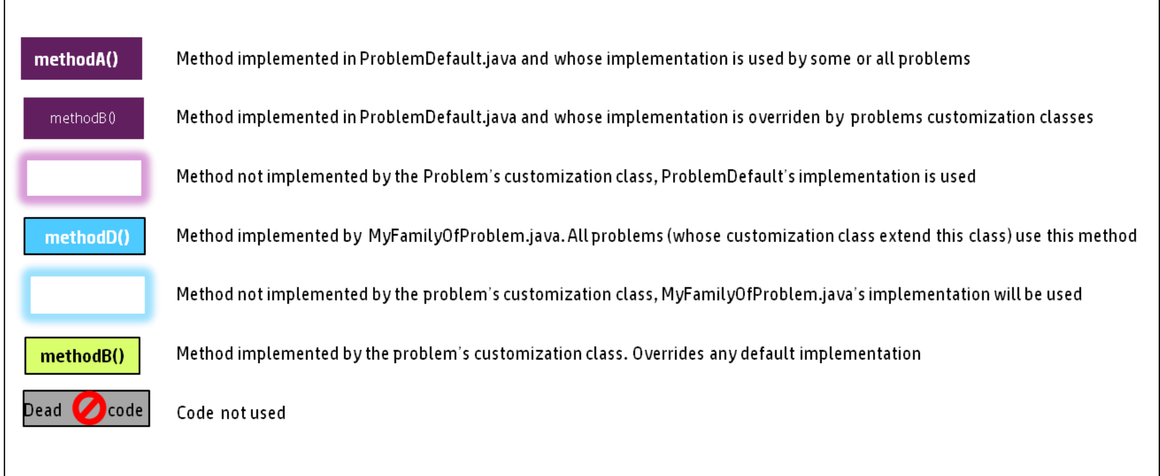


Figure 33 - schema of implementation of the main Problem Detection interfaces



Problem Entity, Multiple Problem Entities, Problem key

Problem Entity / Problem Entities definition

For each alarm passing the filters, Problem Detection will calculate a single or multiple problem entities. This or these problem entities represent the "module(s), element(s), service(s), …" affected.

For example

1) Alarm reporting the crash of a processor
=> possible problem entity : the processor ID

2) Alarm reporting the fact that a server is unavailable
=> possible problem entity: the server name

3) Alarm reporting a pipe cut between two machines
=> possible problem entities: machine A, machine B

Problem Key definition

As mentioned in the previous paragraph, each alarm passing the filters will have one or several problem entities. To this problem entity, or to each of these problem entities will be associated one problem key.

What is this problem key used for? It defines a perimeter equal or larger than the problem entity. All alarms who passed the same filters, and who share a same problem key, will be considered for potential grouping.

For example

1) Alarm reporting the crash of a processor
=> possible problem entity : the processor ID
=> possible problem key: the server in which the processor is

2) Alarm reporting the fact that a server is unavailable
=> possible problem entity: the server name
=> possible problem key: the server name (same as problem entity)

3) Alarm reporting a pipe cut between two machines
=> possible problem entities: machine A, machine B
=> possible problem key: the site containing machine A, the site containing machine B

Role of Problem Entity / Problem Entities / Problem Key in grouping

When grouping alarms of a type of problem, the problem entit(y)ies of those alarms will be considered.

Case 1 – All the alarms have the same {problem entity} and same [problem key]
For instance, if the following alarms have been received

Alarm1: Destination Host Unreachable {lotus.gre.hp.com} [lotus.gre.hp.com]

Alarm2: server down {lotus.gre.hp.com} [lotus.gre.hp.com]

Alarm3: fans stopped working {lotus.gre.hp.com} [lotus.gre.hp.com]

In this simplest case, all alarms have the same problem key, so they will be considered for grouping. They also have the same problem entity so they will be grouped.
The group will also be given this same problem entity.

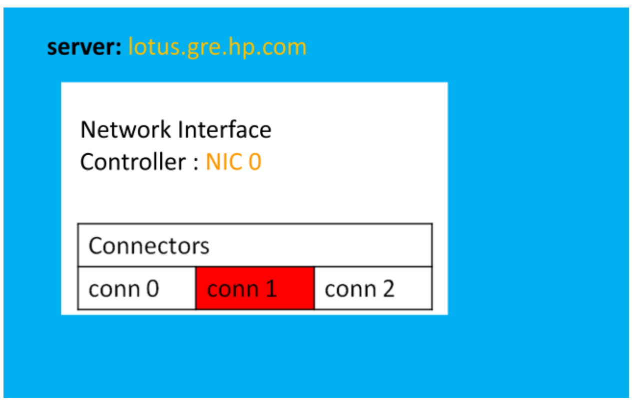
Case 2 – All the alarms have the same [problem key] and a similar {problem entity}

For instance, if the following alarms have been received

Alarm1: Destination Host Unreachable {lotus.gre.hp.com} [lotus.gre.hp.com]

Alarm2 (**Trigger alarm**) : Network Interface Controller down {lotus.gre.hp.com__NIC_0} [lotus.gre.hp.com]

Alarm3: 8P8C 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 superstring or substring of the problem entity of the trigger alarm. The overridable method `compareProblemEntities` decides whether each alarm should be part of the group or not.
The group will be given the problem entity of the trigger alarm : lotus.gre.hp.com__NIC_0

Case 3 – Some alarms have multiple {problem entities}

For instance, if the following alarms have been received

Alarm1: remote site not accessible {site GRE} [lotus.gre.hp.com]

Alarm2 (**Trigger alarm**) : Broken pipe {site GRE, site VBE} [lotus.gre.hp.com, nenufar.vbe.hp.com]

Alarm3: 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.

If the property “sameGroupForAllProblemEntities” is set to false (default value), two groups will be created:

Group 1 (groupname = <p> *problem name* </p> <e> lotus.gre.hp.com </e>
group keys = <p> *problem name* </p> <k> site GRE </k>
containing alarm 1 and alarm 2

Group 2 (groupname = <p> *problem name* </p> <e> nenufar.vbe.hp.com </e>
group keys = <p> *problem name* </p> <k> site VBE </k>
containing alarm 2 and alarm 3

If the property "sameGroupForAllProblemEntities" is set to true, only one group will be created:

Group 1 (groupname = <p> *problem name* </p> <e> lotus.gre.hp.com </e> **OR** <p> *problem name* </p> <e> nenufar.vbe.hp.com </e> (random choice)
group keys = <p> *problem name* </p> <k> site GRE </k>
<p> *problem name* </p> <k> site VBE </k>
containing alarm 1, alarm 2, alarm 3

ActionsFactory implementation

A Problem Detection Value Pack needs to send some actions to the various NMS it takes alarms from. For example, a Problem Detection Value Pack needs to tell a particular NMS to clear an alarm, or to create a Problem Alarm.

The set of actions Problem Detection framework is susceptible to invoke is defined in the SupportedActions interface. See [R6] *UCA for EBC Inference Machine – JavaDoc (C:\%UCA_EBC_DEV_HOME%\apidoc\inference-machine\index.html)*

A Problem Detection Value Pack needs to implement the SupportedActions interface for each of the NMS it is connected to.

For example if a Problem Detection Value Pack receives alarms from TeMIP, SCOM and SMARTS, it will have to provide three implementation of the SupportedActions interface.

The implementations of the SupportedActions interface must be done by extending the abstract class `com.hp.uca.expert.vp.pd.actions.ActionsFactory` which provides some common code.

Example of the TeMIP Actions Factory

UCA-EBC Problem Detection provides the implementation of the SupportedActions interface for TeMIP in the uca-evp-pd-fwk.jar. Below is an extract of the TeMIPActionsFactory class showing 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`

Below is an extract of the TeMIPActionsFactoryCallbacks class showing how the clearAlarmCallback method set 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()");
   }
 }
```

B3.2 Example of a non-TeMIP Actions Factory

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 the three following methods is less obvious, so here are some explanations.

associateAlarmsForHistoryNavigation(Action action, Scenario scenario, Group group, Collection<Alarm> children, ProblemInterface problem)

is the method used to tell the NMS that all children alarms have to be grouped together under a problem alarm

In case the NMS is TeMIP, associateAlarmsForHistoryNavigation will invoke the TeMIP directive GROUPALARMS

In the case of a non-TeMIP NNMS, there maybe one dedicated method to group children alarms with a problem alarm, or maybe it is done through setting some field of the alarms to be grouped.

In any case associateAlarmsForHistoryNavigation is the place where to invoke the one or several NMS methods to achieve grouping

dissociateAlarmsForHistoryNavigation is the reverse method of associateAlarmsForHistoryNavigation.

Is the method to use when the children alarms should not be grouped any longer under the problem alarm of a given group.

setHistoryNavigation(Action action, Scenario scenario, Alarm alarm, Qualifier qualifier)

is the method to set the field of the alarm indicating the alarm is a subalarm, or a problem alarm, or a candidate alarm, or an orphan alarm

Even if you don’t need to modify the alarms in your NMS with this information, you at least need to update the alarm in the Working Memory of Problem Detection

Below we have taken the example of an Actions Factory for a NMS called MyCOolNMS

```
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

The javadoc of the ActionRequest class is given at
[R7] Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Actions (C:\%UCA_EBC_DEV_HOME%\apidoc\uca-mediation-action-client\index.html)

Basically, you need to add the right commands in the form of key/value pairs to the action object that is passed

What to put in the action, what commands... depends on what your MyCOolNMS Channel Adapter expects.

How Actions Factory are referenced and invoked

Suppose your UCA-EBC Problem Detection Value Pack is connected to two NMS : Smarts and SCOM.

You have implemented one Actions Factory for each of these NMS.

Now when it needs to send an action, for example when it needs to create a Problem Alarm, Problem Detection framework will need to know which actions factory to use, and which NMS to target.

The ProblemXmlConfig.xml of your Value Pack (that could look like the one given in the example below) will associate an action name and an action class

```
<ProblemPolicies xmlns="http://config.pd.vp.expert.uca.hp.com/">
  <mainPolicy>
       [ . . . ]
    <actions>
     <defaultActionScriptReference>Exec_localhost</defaultActionScriptReference>
      <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 given action to do on a given alarm, the Actions Factory to invoke will be found thanks to the method below available in the ProblemDefault.java and in your Problem customization classes if you have defined it.

```
public SupportedActions chooseSupportedActions(Alarm alarm,
ProblemInterface problem)
[...]
            SupportedActions supportedActions =
getSupportedActions().get(alarm.getSourceIdentifier());
[...]
```

In the code snipped above, the action name is taken from the "alarm.getSourceIdentier()"

So if in the alarm, the field sourceIdentifier == SMARTS, then the actions Factory chosen will be the one having `<action name="SMARTS">` in ProblemXmlConfig.xml, i.e. `com.acme.af.SmartsActionsFactory`

And the Action Reference will be `Smarts_Notif_localhost`

And to know which NMS to target, Problem Detection will look at the ActionRegistry.xml located at:
${UCA_EBC_INSTANCE}/conf/ActionRegistry.xml

that could look like this example below

```
<?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

If you want your UCA-EBC Problem Detection Value Pack to be sending actions to a Trouble Ticketing System, then you need

- To configure `ProblemXmlConfig.xml` located in the `src/main/resources/valuepack/conf/` in your development environment.
- To configure ${UCA_EBC_INSTANCE}/conf/ActionRegistry.xml
- To implement a Trouble Ticket Actions Factory for your Trouble Ticketing System (if it is not TeMIP)
- To develop a Channel Adapter for your Trouble Ticketing System (not covered in this guide)

configuring the ProblemXmlConfig.xml

The ProblemXmlConfig.xml associates a TroubleTicketAction name with

- an actionReference that will be used to know which Trouble Ticketing system to address
- an actionClass that will be used to know which implementation of the TroubleTicketActionsFactory to use

```
<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.TeMIPTroubleTicketActionsFactory</acti
onClass>
[ . . . ]
</troubleTicketAction>
</troubleTicketActions>
</mainPolicy>
```

By default, the name of the TroubleTicketAction to use for a given alarm, is to be found in the filters of that alarm.

Below is an extract of the ProblemDefault.java

```
@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);
                                                    }
    }
  }
```

Note that this behavior is overridable.

configuring the ActionRegistry.xml

The action registry will associate an actionReference with a Trouble Ticketing System name, here called as NmsName.

ActionRegistry.xml

```
<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 your Trouble Ticketing System is not TeMIP, then you need to implement a Trouble Ticket Actions Factory

A Trouble Ticket Actions Factory is the place where you will implement the methods of the **SupportedTroubleTicketActions** interface.
See [R6] *UCA for EBC Inference Machine – JavaDoc (C:\%UCA_EBC_DEV_HOME%\apidoc\inference-machine\index.html)*

Some of the methods of this interface are `createTroubleTicket`, `closeTroubleTicket`, …

The Trouble Ticket Actions Factory corresponding to the Trouble Ticketing System you use, must implement SupportedTroubleTicketActions interface and extend the TroubleTicketActionsFactory abstract class that contains some common code

The example below shows an extract of the implementation 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 above example) must fill the action to be sent to the Trouble Ticketing System.

The javadoc of the ActionRequest class is given at
[R7] Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Actions (C:\%UCA_EBC_DEV_HOME%\apidoc\uca-mediation-action-client\index.html)

You need to add the right commands, in the format of key/value pairs, to the action object that is passed

The content of the commands depends on what your Trouble Ticketing System Channel Adapter expects and supports.

PD Value Pack example with Events Only

Annex D.

Not yet available in IM SDK.

Annex E. TSP Value Pack example

Not yet available in IM SDK.

Annex F.

TSP Advanced customization

As seen in chapter 8.2 it is possible to modify the default behavior of Topology State Propagator Value Packs.

The behavior can be modified, as for PD Value Packs in the following way:

- per propagation
- per family of propagations
- for all propagations
- for non propagation specific matters

It is very similar to Problem Detection customization as described in Annex C

Annex G.

IM 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 will be able to recognize 2 problems with Problem Detection scenario:

- Problem_SwitchDown
- Problem_PhoneUnavailable

And several propagations based on above problems with 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 TSP scenario.

Propagations are hierarchized and only top-level ones are creating Service Alarms. The Problem and Service Alarms are stored on disk using the DBActionsFactory.

Also, sample tests file that can be run with JUnit are contained. Those 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.