hp Unified Correlation Analyzer

Unified Correlation Analyzer
for EBC
Problem Detection

Version 3.1

Installation, Administration and Development Guide

Edition: 1.0

For Windows and Linux (RHEL 5.8 & 6.3) Operating Systems

April 2014

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

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.

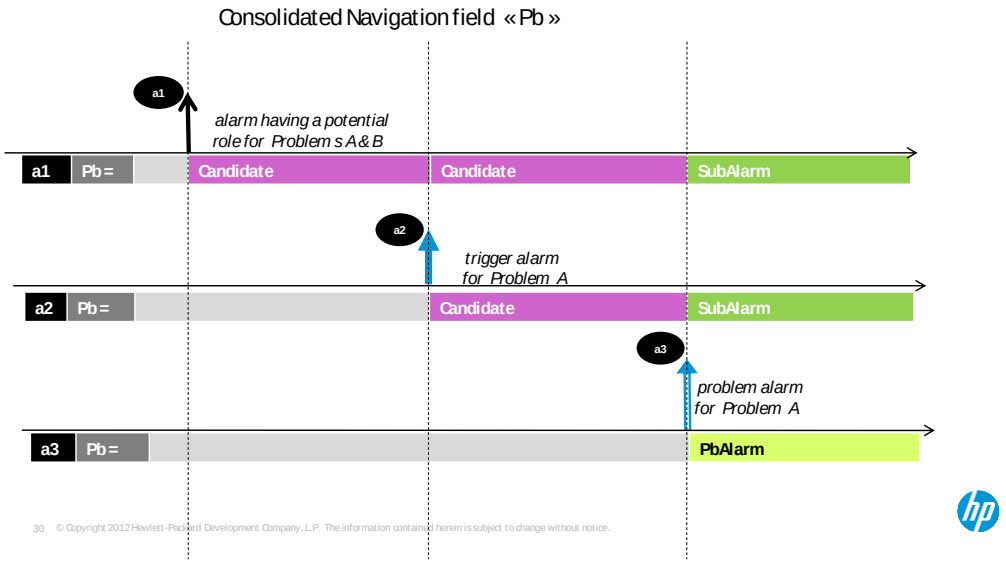
Red Hat® is a registered trademark of the Red Hat Company.

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

Contents

Tables

Figures



Preface

The intention of this document is to gather all the information about HP UCA for EBC Problem Detection.

Product Name: Unified Correlation Analyzer for Event Based Correlation Problem Detection

Product Version: 3.1

Kit Version: V3.1

Intended Audience

The intended audience of this guide is primarily developers (customers or HP consultants) wanting to create a Problem Detection Value Pack in UCA for EBC.
This document will also be interesting for anyone wanting to know more about Problem Detection 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”

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 3.1	• HP-UX 11.31 for Itanium • Red Hat Enterprise Linux Server release 5.8 & 6.3
UCA for Event Based Correlation Channel Adapter Version 3.1	• HP-UX 11.31 for Itanium • Red Hat Enterprise Linux Server release 5.8 & 6.3
UCA for Event Based Correlation Software Development Kit Version 3.1	• Windows XP / Vista • Windows Server 2007 • Windows 7 • Red Hat Enterprise Linux Server release 5.8 & 6.3
UCA for Event Based Correlation Problem Detection Kit Version 3.1	• Windows XP / Vista • Windows Server 2007 • Windows 7 • Red Hat Enterprise Linux Server release 5.8 & 6.3

Table 1 - Software versions

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 Problem Detection – JavaDoc Problem Detection framework (C:\%UCA_EBC_DEV_HOME%\apidoc\uca-evp-pd-fwk\index.html)*

[R7] *Unified Correlation Analyzer for Event Based Correlation – JavaDoc UCA Actions (C:\%UCA_EBC_DEV_HOME%\apidoc\uca-mediation-action-client\index.html)*

Support

Please visit our HP Software Support Online Web site at www.hp.com/go/hpsoftwaresupport 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

UCA-EBC Problem Detection: a quick tour

Chapter 2 lists the features and capabilities of the Problem Detection framework

Chapter 3 is the Problem Detection Development Kit installation guide

Chapter 4 to Chapter 7 are the Problem Detection value pack development guide

Chapter 7 and Chapter 8 are the Problem Detection administration guide

Annex A provides a comprehensive Problem Detection Value Pack example

Annex B deals with advanced customization possibilities of Problem Detection

1.1 Problem Detection naming disambiguation

The name "*Problem Detection*" has different meanings in different contexts.

Problem Detection can be a short name for

- Problem Detection Development Kit
- Problem Detection Framework
- Problem Detection Value Pack

Problem Detection Development Kit: the Eclipse environment (including plug-ins) to develop a Problem Detection Value Pack. The Problem Detection Development Kit is composed of both the UCA EBC Development Kit and the UCA EBC Development Kit Problem Detection Extension (which contains templates to create Problem Detection Value Packs using the UCA EBC Eclipse Plug-in).

Problem Detection Value Pack: A UCA EBC Value Pack built on top of the Problem Detection Framework (using the Problem Detection Development Kit).

Problem Detection 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 Dev Kit Problem Detection Extension and packaged into any Problem Detection Value Pack.

1.2 Licensing

Problem Detection is a licensed product.

1.3 What does Problem Detection do?

When a type of failure (problem) occurs in the network on some resource at some time T_{pb} (T_{pb} denotes time when the problem occurred), equipments in the neighborhood of that resource, usually generate several alarms in a time window around T_{pb}.

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 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

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.

Please see in the figure below a graphical representation of the process of creating a Problem Alarm and grouping sub-alarms under it using a UCA EBC Problem Detection Value Pack.

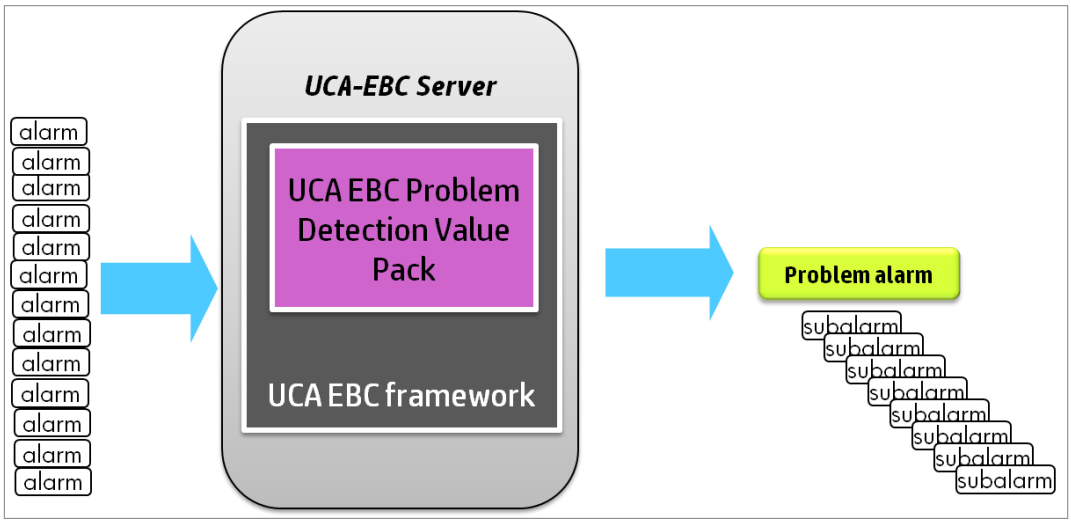


Figure 1 - Alarms grouping

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.

1.4 Architecture overview

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 Problem Detection Value Pack receives its alarms through UCA EBC Alarm Collection flow coming from one or several of the Network Management Systems connected to OSS Open Mediation.

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

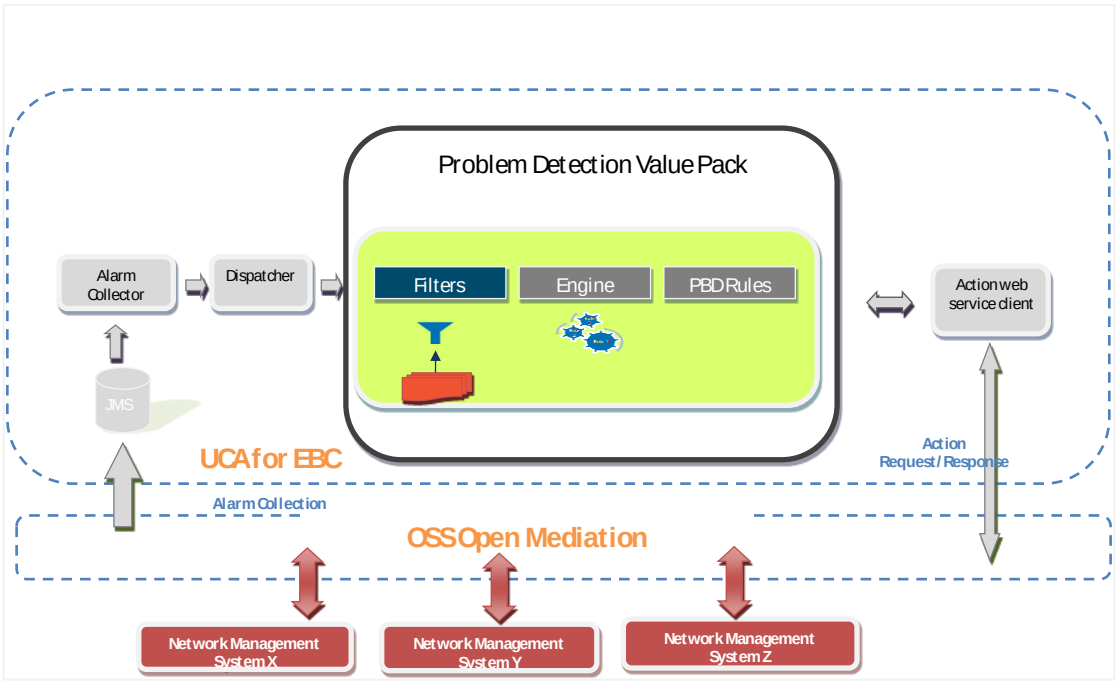


Figure 2 – Problem Detection solution architecture

Contrary to other UCA for EBC Value Packs, a Problem Detection Value Pack does not allow its developer to modify the set of rules.

Chapter 2

Problem Detection Features

2.1 Main features

2.1.1 Problem identification

The primary role of a Problem Detection Value Pack is to identify that a failure (problem) has occurred based on the appearance of a certain set of alarms, and on the presence of certain conditions;

And then to generate an operator readable Problem Alarm that summarizes the problem.

2.1.2 Alarms grouping

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.

2.2 Automatic actions

Besides noticing and reporting the appearance of a failure (problem), besides grouping alarms, Problem Detection can execute other automatic actions with respect to the lifecycle of alarms, and with respect to Trouble Tickets.

2.2.1 Alarm state propagation

Problem Detection offers the following default behaviors

Table 2 - Alarm state propagation from Problem Alarm to sub-alarms

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)

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

Table 3 - Alarm state propagation from sub-alarms to Problem Alarm

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

2.2.2 Trouble Ticket creation

Problem Detection can be configured to automate the creation of Trouble Tickets.

2.2.3 Trouble Ticket propagation

When a Trouble Ticket is created (automatically or manually) for a Problem Alarm it is possible to associate all sub-alarms of the Problem Alarm to the Trouble Ticket.

When a Trouble Ticket is manually created for a sub-alarm, it is possible to associate the Problem Alarm to the Trouble Ticket.

2.3 Cross domain correlation

Problem Detection Value Packs, 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.

Without developers having to write any Java code, the Problem Detection framework is able to send actions to TeMIP, and is 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 Problem Detection framework too since it is layered on top of the UCA-EBC framework.

Please see section 5.2.3 Actions and section 5.2.4 Trouble Ticket Actions

Of course an open API is 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 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

2.4 Enrichment

If some of the alarms received from the NMS (Network Management System) do not contain enough information to be correlated, the Problem Detection framework offers 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 (Problem Detection framework defines standard actions that can be customized)

It is also possible to write Java code doing any imaginable synchronous or asynchronous request (database access, file access, HTTP request …).

2.5 Performance

Compared to a standard UCA for EBC Value Pack that would have been developed to perform correlation, a Problem Detection Value Pack is very likely to perform significantly better. The reason is that the Problem Detection 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.

2.6 Robustness

One of the greatest advantages of the Problem Detection Framework is its robustness.

All Problem Detection Value Packs use the fixed set of rules provided by the Problem Detection framework.
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 a Problem Detection Value Pack will neither have to worry about the rules nor the performance of the Value Pack.

2.7 Ease of use / Simulation

The steps (listed in Chapter 4) required to create a Problem Detection Value Pack are relatively simple and short.

If you're satisfied with the default behavior of Problem Detection Value Packs, the creation of a Value Pack will not require any java coding or rule writing. It will only require modifying a few XML files as explained in Chapter 5.

Another advantage of Problem Detection 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

Problem Detection Development Kit Installation Guide

This chapter explains how to install the Problem Detection Development Kit. The Problem Detection Development Kit contains the Eclipse environment (including plug-ins) to develop a Problem Detection Value Pack. The Problem Detection Development Kit is composed of both the UCA EBC Development Kit and the UCA EBC Development Kit Problem Detection Extension. Please note that the installation and deployment of a Problem Detection Value Pack are covered in sections 8.1 “Installing a Value Pack” and 8.2 “Deploying a Value Pack”.

The UCA for EBC Problem Detection Development Kit runs and is supported on Windows and Linux (RHEL 5.8 & 6.3). It is delivered as follow:

On Windows XP/Vista, Windows 7, Windows Server 2007:

uca-evp-pd-packaging-3.1-msi.zip

On Linux:

uca-evp-pd-packaging-3.1-linux.tar

This chapter describes the software prerequisites, the installation steps, and gives a brief content description of the UCA for EBC Problem Detection Development kit.

3.1 Licensing

The UCA for EBC Problem Detection product is a licensed product.

The following table shows the link between UCA for EBC Problem Detection part numbers and features:

Product Part Number	Description	Enabled UCA for EBC features
JJ128FAE	UCA for EBC Problem Detection Value Pack	UCA for EBC Problem Detection Framework

Table 4 - UCA for EBC Problem Detection product part numbers and features

For any questions related to licensing, please get in touch with the UCA for EBC product management.

3.2 Disk requirements

Here are the disk requirements for the UCA for EBC Problem Detection Development Kit:

On Windows:

Type	Disk requirements
Temporary disk space	4 MB minimum: 2 MB minimum for the uca-evp-pd-packaging-3.1-msi.zip file 2 MB minimum for the UCA-EBC-DEVPD-V3.1-00B.msi file (expanded from the uca-evp-pd-packaging-3.1-msi.zip file)
Permanent disk space	6 MB minimum for UCA for EBC Problem Detection Development Kit V3.1 installed on the system

Table 5 – Disk Requirements for UCA for EBC Problem Detection Development Kit on Windows

On Linux:

Type	Disk requirements
Temporary disk space	4 MB minimum: 2 MB minimum for the uca-evp-pd-packaging-3.1-linux.tar file 2 MB minimum for the UCA-EBC-DEVPD-V3.1-00B.noarch.rpm and install-uca-ebc-pd.sh files (expanded from the uca-evp-pd-packaging-3.1-linux.tar file)
Permanent disk space	6 MB minimum for UCA for EBC Development Kit V3.1 installed on the system

Table 6 – Disk Requirements for UCA for EBC Problem Detection Development Kit on Linux

3.3 Software prerequisites

The UCA for EBC Problem Detection Development Kit is installed on top of the UCA for EBC Development Kit. It brings the ability to create UCA for EBC Problem Detection Value Packs using the UCA for EBC Development Kit.

Product	Version	Operating System
UCA for EBC Development Kit	3.1	Windows or Linux
Java	Java 1.6 or Java 1.7	Windows or Linux

Table 7 – Software Prerequisites for UCA for EBC Problem Detection Development Kit

3.3.1 Java

UCA for EBC V3.1 Server, UCA for EBC V3.1 Topology Extension, UCA for EBC V3.1 Development Toolkit, and UCA for EBC V3.1 Value Packs support both Java 1.6 and Java 1.7. The same applies to UCA for EBC V3.1 Problem Detection Development Toolkit.

Any of the following versions of Java is needed:

Software	Version
Java JRE/JDK 6	1.6.0.08 (or later)
Java JRE/JDK 7	1.7.0.00 (or later)

Table 8 - Java Prerequisites for UCA for EBC Problem Detection Development Kit

Note

Please note that if your Value Packs are compiled with one version of Java, it is strongly recommended that UCA for EBC Server is also running the same version of Java to avoid running into compatibility issues between Java 6 and Java 7.

For more information on these compatibility issues, you can go to:

http://www.oracle.com/technetwork/java/javase/compatibility-417013.html

Java JRE (Java Runtime Environment) is enough for using the UCA for EBC Development Kit. However the Java JDK (Java Development Kit) comes with some useful debugging tools (jconsole, jvisualvm, etc…) that may help. It is therefore recommended to install the JDK, instead of just installing the JRE.

The JAVA_HOME environment variable must be set before using UCA for EBC Problem Detection Development Kit:

On Windows:

In the *Control Panel*, Open *System Properties,* open the *Advanced* tab and click *Environment Variables*, then set the JAVA_HOME environment variable according to the location of your JDK:

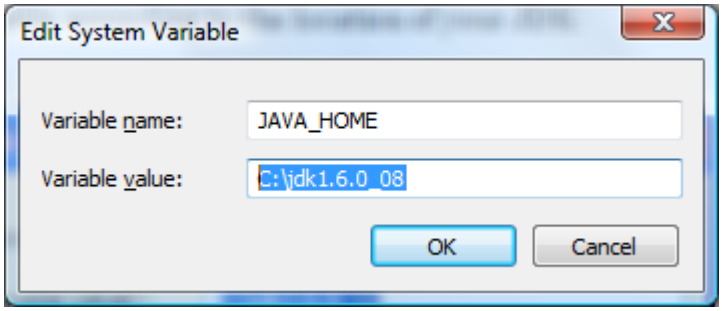


Figure 3 - Setting the JAVA_HOME environment variable on Windows systems

In case Java is not yet installed on your system, the latest JRE/JDK package for Microsoft Windows operating systems can be downloaded (for free) from http://java.com/en/download/manual.jsp.

On Linux:

Depending on your shell, and the location of the Java JRE/JDK software, please use one of the following commands to set the JAVA_HOME environment variable:

Example for **csh**-like shell:

```
$ setenv JAVA_HOME /usr/java/jdk1.6.0_37
```

Example for **sh**-like shell:

```
$ export JAVA_HOME=/usr/java/jdk1.6.0_37
```

To check if you already have Java installed:

```
$ rpm -qa | grep jdk
```

Red Hat Enterprise Linux Server comes with OpenJDK Java VM. You should get an output similar to the following (here 1.6.0 and 1.7.0 are installed):

```
java-1.6.0-openjdk-1.6.0.0-1.50.1.11.5.el6.x86_64
java-1.6.0-openjdk-devel-1.6.0.0-1.50.1.11.5.el6.x86_64
java-1.7.0-openjdk-1.7.0.9-2.3.4.1.el6_3.x86_64
java-1.7.0-openjdk-devel-1.7.0.9-2.3.4.1.el6_3.x86_64
```

You can also download (for free) the latest Java packages (HotSpot Java VM) from Oracle from http://java.com/en/download/manual.jsp. If this is installed (usually under /usr/java), you should get an output similar to the following:

```
jdk-1.6.0_37-fcs.x86_64
```

3.3.2 UCA for EBC Development Kit

The UCA for EBC Problem Detection Development Kit is installed on top of the UCA for EBC Development Kit. So you need to install the UCA for EBC

Development Kit prior to installing the UCA for EBC Problem Detection Development Kit.

☞ Please refer to the [R3] *Unified Correlation Analyzer for Event Based Correlation Installation Guide* for information on how to install the UCA for EBC Development Kit.

3.4 Installation of the UCA for EBC Problem Detection Development Kit

3.4.1 Product installation

The UCA for EBC Problem Detection Development Kit runs and is supported on Windows and Linux (RHEL 5.8 & 6.3). It is delivered as follow:

On Windows XP/Vista, Windows 7, Windows Server 2007:

`uca-evp-pd-packaging-3.1-msi.zip`

On Linux:

`uca-evp-pd-packaging-3.1-linux.tar`

The following paragraphs detail how to install the UCA for EBC Problem Detection Development Kit on either Windows or Linux systems:

On Windows:

Install the UCA for EBC Development Kit by executing the `UCA-EBC-DEVPD-V3.1-00B.msi` file.

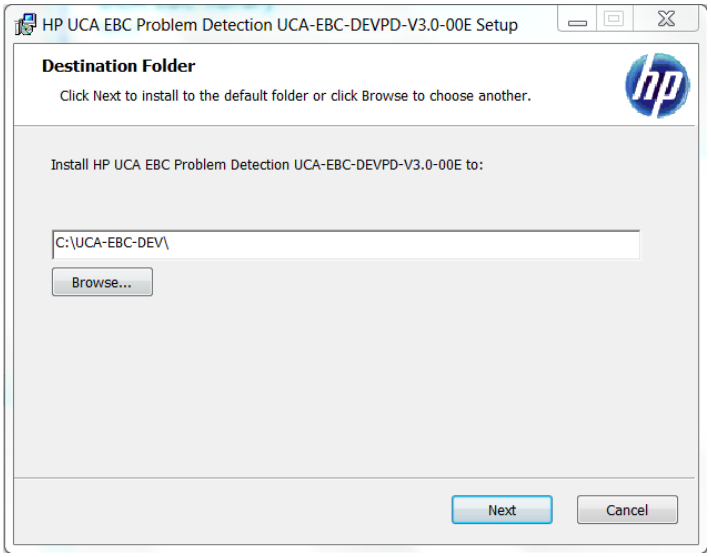


Figure 4 - Installing the UCA for EBC Problem Detection Development Kit

By default, the UCA for EBC Problem Detection Development Kit is installed in the `C:\UCA-EBC-DEV` directory because it is the default location of the UCA for EBC Development Kit. If you installed the UCA for EBC Development Kit at an alternate location, please use the same location for installing the UCA for EBC Problem Detection Development Kit.

On Linux:

Untar the `uca-evp-pd-packaging-3.1-linux.tar` archive in a temporary directory:

As **root** user, untar the archive in a temporary local directory (For example: `/tmp`):

```
$ cd /tmp
$ tar -xvf <kit location>/uca-evp-pd-packaging-3.1-linux.tar
```

Run the installation script

Depending on whether you wish to install the UCA for EBC Problem Detection Development Kit at the default location, i.e. `/opt/UCA-EBC-DEV`, or an alternate location, run either of the following commands to execute the installation script. In any case, as the UCA for EBC Problem Detection Development Kit is installed on top of the UCA for EBC Development Kit, the location you select (either the default location or an alternate location) must be a valid location where the UCA for EBC Development Kit is installed.

To install UCA for EBC Problem Detection Development Kit at the default location (in the `/opt/UCA-EBC-DEV` directory, assuming UCA for EBC Development Kit is installed at the same location), please execute the following command as **root** user:

```
$ install-uca-ebc-pd.sh
```

To install UCA for EBC Problem Detection Development Kit at an alternate location of your choosing (assuming UCA for EBC Development Kit is installed at the same location), please execute the following command as **root** user:

```
$ install-uca-ebc-pd.sh –r <Alternate root directory>
```

Note

Installing UCA for EBC Development kit as non-root user (Linux only):

For testing purpose (or for some very specific needs) the UCA for EBC Problem Detection package can be installed by a non-root user. This feature is available for Linux only.

When installing UCA for EBC Problem Detection as non-root user, the following limitations must be understood and acknowledged:

The system RPM database is not accessible by a non-root user. As a consequence, when installation is performed by a non-root user, a specific RPM database must be specified. The default RPM repository for non-root installation is set to ~/.rpmdb (where ~ is the user home directory).
This directory can be overridden by specifying the –-rpmdbpath option as installation script argument.

The UCA for EBC Problem Detection root directory must be read/write accessible by the non-root user. Usually the default /opt/UCA-EBC-DEV directory cannot be used (unless some specific rights have been set by the administrator). As a Consequence, when installation is performed by a non-root user, the –r option **must be** specified.

When installed by the non-root users the UCA for EBC Problem Detection files are owned by the user who performed the installation. This user should be the same than the one who performed installation of the UCA for EBC Development toolkit package.

3.4.2 File organization

The UCA for EBC Problem Detection Development Kit is installed in the root directory specified at installation (by default `C:\UCA-EBC-DEV` on Windows systems or `/opt/UCA-EBC-DEV` on Linux systems).

The following table describes the different subdirectories contained in the delivery:

Directory	Description
apidoc	Contains the javadoc for the UCA for EBC Problem Detection Development Kit
bin	Contains the uninstallation script for the UCA for EBC Problem Detection Development Kit (on Linux systems only): `uninstall-uca-ebc-pd`
Eclipseplugin/templates	Contains templates used by the UCA for EBC Eclipse IDE plug-in for creating new UCA for EBC Problem Detection Value Packs
vp-examples/pd-example	Contains the sample pd-example UCA for EBC Problem Detection value pack.

Table 9 - Sub-directories of UCA for EBC Problem Detection Development Kit installation directory

3.4.3 Javadoc

The jar file of the UCA for EBC Problem Detection Javadoc is available in the directory where you installed the UCA for EBC Problem Detection Development Kit, by default the `C:\UCA-EBC-DEV` on Windows systems or the `/opt/UCA-EBC-DEV` directory on Linux systems directory, under the *`apidoc`* directory.

When your UCA for EBC Problem Detection project is created thanks to the UCA for EBC Eclipse IDE project builder plug-in, the Javadoc will be

automatically associated to the UCA for EBC Problem Detection Framework libraries.

☞ Please refer to [R2] *Unified Correlation Analyzer for Event Based Correlation Value Pack Development Guide* for full details on how to install the UCA for EBC Eclipse IDE plug-in.

3.4.4 Post-installation setup

MSL update: This post-installation step is optional. It only applies if, and only if, the target for your Problem Detection Value Packs is TeMIP:

For the UCA for EBC Problem Detection Value Packs to function, new user-defined TeMIP Alarm Object attributes need to be added to the TeMIP Dictionary on the system(s) hosting your TeMIP director(s):

- **PB** (Latin1String: id=10100): This attribute defines the category of the alarm: ProblemAlarm (parent), SubAlarm (child), ProblemSubAlarm (parent and child), Candidate (not yet a child), Alarm (no more a child or a parent)
- **Grouping Keys** (Latin1String: id=10101): This attribute is used by TPD to support real-time parent<->children navigation in the TeMIP Client.
- **Number of Cleared Alarms** (Unsigned32: id=10102)
- **Number of Total Alarms** (Unsigned32: id=10103)
- **Number of Acknowledged Alarms** (Unsigned32: id=10005)
- **Number of Outstanding Alarms** (Unsigned32: id=10006)

Those attributes are available on

Linux

TFR (TeMIP framework) V61L Maintenance Release

HP-UX

PHSS_ 43236 E-Patch on HP-UX IA platform (TFR V6.1)

Solaris

TEMIPTFRSOL_00349 E-Patch on SUN Solaris platform

These user-defined fields are easily added through the dedicated tool (on the machine where the TeMIP server runs)

`temip_ah_user_defined_attr` (located in `/usr/opt/temip/bin`) and the project TPD is configured by running the following command:

```
# temip_ah_user_defined_attr -project TPD
```

Confirm that the attributes listed above are correctly added in the Dictionary running the following command:

```
# temip_ah_user_defined_attr
```

Output should be showing

```
------------------------ User Defined Attributes ------------------------
--------------------------------------
[##]           Pres. Name = MSL ID :           Data Type  -    Symbol
Settable
[ 1] PB =  10100 :         Latin1String ->
AO_PB
[ 2] Grouping Keys =  10101 :  Latin1String ->
AO_GROUPING_KEYS
[ 3] Number of Cleared Alarms =  10102 : Unsigned32 ->
AO NUMBER OF CLEARED ALARMS
[ 4] Number of Total Alarms =  10103 : Unsigned32 ->
AO_NUMBER_OF_TOTAL_ALARMS
[ 5] Number of Acknowledged Alarms = 10005: Unsigned32->
AO_NUMBER_OF_ACKNOWLEDGED_ALARMS [ 6] Number of Outstanding Alarms =
10006: Unsigned32 -> AO_NUMBER_OF_OUTSTANDING_ALARMS
```

You can alternatively check the dictionary

```
# mcc_dap_browser&
```

Then

operation_context->alarm_object->partition->user_defined

3.5 Uninstallation of the UCA for EBC Problem Detection Development Kit

In order to uninstall the UCA for EBC Problem Detection Development Kit, please follow the instructions below:

On Windows:

Go to the Control Panel

Select “Program and Features”

Right-click on “**HP UCA EBC Problem Detection UCA-EBC-DEVPD-V3.1-00B**”

Select “Uninstall”

Click “**Yes”** to confirm the uninstallation.

On Linux:

Go to the `bin/` subfolder of the root directory of UCA for EBC Development Toolkit (by default `/opt/UCA-EBC-DEV/bin`) and execute the uninstallation script:

```
$ cd ${UCA EBD DEV HOME}/bin/
$ uninstall-uca-ebc-pd
```

You should get an output similar to the following text:

```
Here is the list of installed UCA-EBC Problem Detection
packages:


        [0]     UCA-EBC-DEVPD-V3.1-00B.noarch

Enter the index number of UCA-EBC Problem Detection version to
un-install ('Enter' to Cancel):
```

By entering '0' (as in the example above), UCA for EBC Development Toolkit version V3.1-00B will be removed.

3.6 Code signing

The below mentioned procedure* allows you to assess the integrity of the delivered Product before installing it, by verifying the signature of the software packages.

1) Install the **GnuPG** tool
   - Get the gpg software for Windows from The GnuPG website You will easily find it in the Binaries subsection
   - Verify the downloaded SW via its SHA1 checksum if it is a first installation or via its associated signature if a previous version were already installed.
   - Install the downloaded Software the usual way.
   - Start a cmd.exe to have a windows shell

2) Download hpPublicKey
   - Open command prompt
   - Browse to the bin directory in the **GnuPG** installed folder
   - Get the hpPublicKey from following location: https://h20392.www2.hp.com/portal/swdepot/displayProductInfo.do?productNumber=HPLinuxCodeSigning
   - Follow the instruction found at web page
   - Save it as **hpPublicKey.pub**

3) Import gpg-hpPublicKey.pub

Type:

gpp --import <location of HPSignClient installed directory>\gpg-hpPublicKey.pub

4) Verify the signed binary

Type:

gpg --verify <Problem Detection.sig > <Problem Detection .zip >*.

The output should be as shown similar to one given bellow.

```
gpg: Signature made Wed Nov 17 12:32:46 2010 IST using DSA key ID 2689B887
gpg: Good signature from "Hewlett-Packard Company (HP Codesigning Service)"
gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the
owner.
Primary key fingerprint: FB41 0E68 CEDF 95D0 6681  1E95 527B C53A 2689 B887
```

NOTE: message “Good signature from “Hewlett-Packard Company (HP Codesigning Service)” “indicates the code sign verification is successful.

**HP strongly recommends using signature verification on its products, but there is no obligation. Customers will have the choice of running this verification or not as per their IT Policies.*

Chapter 4

Overview of the steps required to create a Problem Detection Value Pack

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**".

4.1 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
- 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)

4.2 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 Chapter 6)

Once the “trigger” alarms and “sub-“alarms have been identified,

Once we have the list of interesting problems (resulting from above step 4.1), 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 5.1 Filters. See also Annex A.

4.3 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 **5.3**

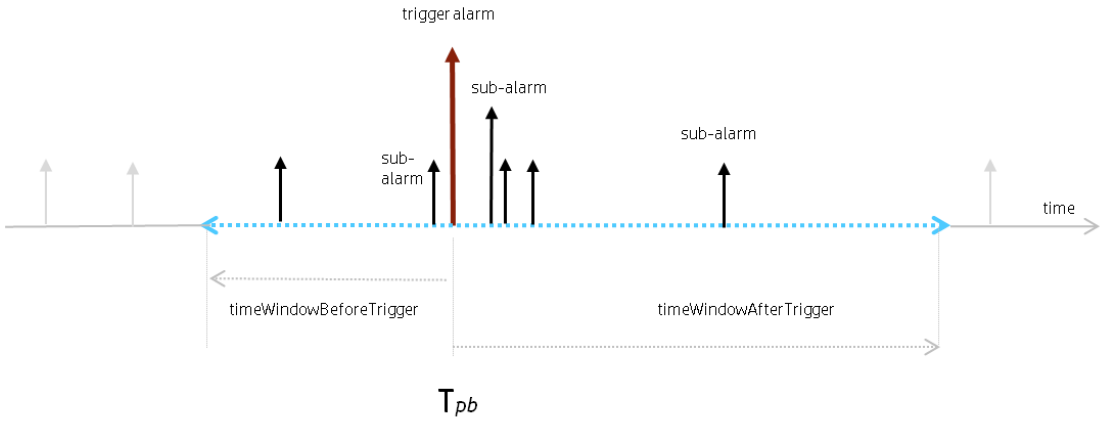


Figure 5 - Time window illustration

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.

4.4 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 5.1

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 5.3

4.5 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 5.3 Problem specific policies

4.6 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 4.1 to 4.5

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 Chapter 6.

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

Chapter 5

How to configure Problem Detection

This chapter describes in detail the steps required to configure a Problem Detection Value Pack. By applying the information presented in this chapter, a developer should be able to create a Problem Detection Value Pack that performs the main steps involved in problem detection: detecting failures (problems), creating Problem Alarms, grouping sub-alarms and managing the life cycle of alarms and trouble tickets.

5.1 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 under *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 A.

```
<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 itself 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 5.2 “Main Policies” 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 10 - 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"`.

5.2 Main Policies

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

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

Please note that the XML schema of this file is available in the directory *src/main/resources/valuepack/conf/*.

Below is an extract of a `ProblemXmlConfig.xml` file, which shows the contents of the `<mainPolicy>` XML tag:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ProblemPolicies xmlns="http://config.pd.vp.expert.uca.hp.com/">
  <mainPolicy>
```

```
        <candidateVisibility>
                <candidateVisibilityTimeMode>Max</candidateVisibilityTimeMode>
                <candidateVisibilityTimeValue>30000</candidateVisibilityTimeValue>
                <markCandidate>false</markCandidate>
        </candidateVisibility>
```

```
        <transientFiltering>
                <transientFilteringEnabled>false</transientFilteringEnabled>
                <transientFilteringDelay>5000</transientFilteringDelay>
        </transientFiltering>
```

```
        <actions>
          <defaultActionScriptReference>Exec_localhost</defaultActionScriptReference>

          <action name="TeMIP EMS">

              <actionReference>TeMIP_AO_Directives_localhost</actionReference>
              <actionClass>com.hp.uca.expert.vp.pd.actions.TeMIPActionsFactory</actionClass>

<attributeUsedForKeyDuringRecognition>userText</attributeUsedForKeyDuringRecognition>

        <attributeUsedForKeyPbAlarmCreation>User_Text</attributeUsedForKeyPbAlarmCreation>
             <longs>
                <long key="maxChildrenLength"><value>15000</value></long>
              </longs>

              <booleans>
                <boolean key="useOnlyGroupingKeys"><value>false</value></boolean>
              </booleans>

              <!-- add the following lines only if you want to configure  -->
              <!-- the Operating Context where you want the problem alarm -->
              <!-- to be created. This Operating Context will be used for -->
              <!-- the whole of this action factory by default, unless    -->
              <!-- specified differently in Problem specific policies     -->
              <strings>
                  <string key="ocName"><value>MY_OC</value></string>
              </strings>
          </action>

        </actions>
```

```
        <troubleTicketActions>

            <troubleTicketAction name="TeMIP TT">
                <actionReference>TeMIP_TT_Directives_localhost</actionReference>
                <actionClass>com.hp.uca.expert.vp.pd.actions.TeMIPTroubleTicketActionsFactory<
        /actionClass>
                <strings>
                  <string key="TT_SERVER entity"><value>TT_SERVER .SM</value></string>
                  <string key="CreateTemplateFile">
                        <value>createTroubleTicketByValueRequest.xml</value></string>
                  <string key="AssociateTemplateFile">
                        <value>associateTroubleTicketByValueRequest.xml</value></string>
                  <string key="CloseTemplateFile">
                        <value>closeTroubleTicketByValueRequest.xml</value></string>
                  <string key="DissociateTemplateFile">
                        <value>dissociateTroubleTicketByValueRequest.xml</value></string>
                  <string key="User"><value>temip</value></string>
                  <string key="Input"><value>input</value></string>
                  <string key="Type"><value>SYNCHRONOUS</value></string>
                  </strings>
              </troubleTicketAction>
            </troubleTicketActions>

        </troubleTicketActions>
```

```
  </mainPolicy>
```

5.2.1 Candidate visibility

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 *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).

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 4.3) 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, then 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.

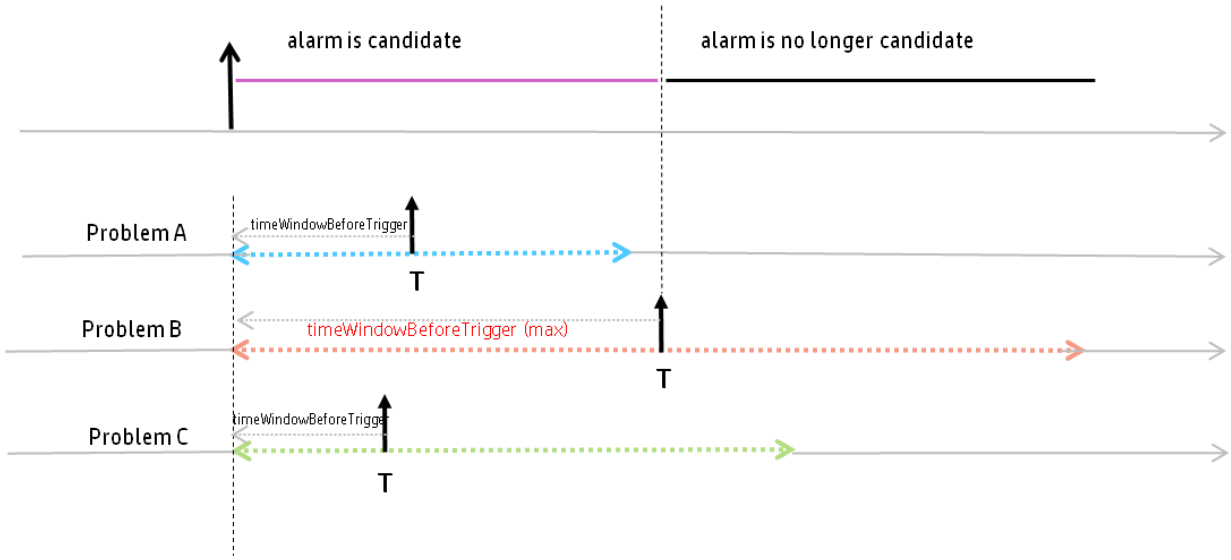


Figure 6 - explanation of the candidateVisibilityTimeMode=Max

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.

5.2.2 Transient Filtering

```
<transientFiltering>
        <transientFilteringEnabled>false</transientFilteringEnabled>
        <transientFilteringDelay>5000</transientFilteringDelay>
</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 Problem Detection 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>`

5.2.3 Actions

The Problem Detection Framework is able to configure multiple actions factories in order to support multiple NMS.

The Problem Detection Framework comes by default with the support of TeMIP Alarm directives.

If your NMS is TeMIP, then you can simply copy/paste the below example.

```
<actions>
  <defaultActionScriptReference>Exec_localhost</defaultActionScriptReference>

  <action name="TeMIP EMS">
        <actionReference>TeMIP_AO_Directives_localhost</actionReference>
        <actionClass>com.hp.uca.expert.vp.pd.actions.TeMIPActionsFactory</actionClass>
        <attributeUsedForKeyDuringRecognition>userText</attributeUsedForKeyDuringRecognit
ion>
        <attributeUsedForKeyPbAlarmCreation>User_Text</attributeUsedForKeyPbAlarmCreation
        >
        <longs>
          <long key="maxChildrenLength"><value>15000</value></long>
        </longs>

        <booleans>
          <boolean key="useOnlyGroupingKeys"><value>false</value></boolean>
        </booleans>

              <!-- add the following lines only if you want to configure  -->
```

```
            <!-- the Operating Context where you want the problem alarm -->
            <!-- to be created. This Operating Context will be used for -->
            <!-- the whole of this action factory by default, unless    -->
            <!-- specified differently in Problem specific policies     -->
        <strings>
          <string key="ocName"><value>MY_OC</value></string>
        </strings>


  </action>

</actions>
```

If your NMS is not TeMIP, an open API to define Actions Factory can be used; please refer to **B.2** ActionsFactory implementation for explanation.

name	type	value
defaultActionScriptReference	property	Unique reference that will be used in the rule to define the routing information of an Action
action	property	Container for attributes defining the actions for a set of alarms
name	attribute	“sourceIdentifier” field of alarms is matched to this name to know which actionsFactory to use for a given alarm
actionReference	attribute	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	attribute	The class implementing the *SupportedAction* interface which describes the methods needed to support any Action on alarms. Methods such as `createProblemAlarm, terminateAlarm, clearAlarm, …`

name	type	value
attributeUsedForKeyDuringRecognition	attribute	The Custom Field Name of the Alarm that will contain the information to identify that a ProblemAlarm is generated by the Problem Detection Framework. In other words this attribute defines the name of the field (in UCA-EBC format) of the problem alarm, that Problem Detection has to look at (when the alarms come back from the NMS) to find the useful info to attach this problem alarm to the right group
attributeUsedForKeyPbAlarmCreation	attribute	The custom Field of the problem alarm that will contain information about the problem. This attribute defines the name of the field in the NMS format) of the problem alarm, in which Problem Detection puts the useful info (at the time of creation of the problem alarm) that it will read when the problem alarm comes back from the NMS. The useful info it contains are things like: name of the trigger alarm, name of the problem, name of the problem entity.
maxChildrenLength	Long attribute	*OPTIONAL* *[TeMIP specific]* 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 attribute	*OPTIONAL* *[TeMIP specific]* 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

name	type	value
copyReferenceAlarmOnPbAlarmCreation	Boolean attribute	*OPTIONAL* *[TeMIP specific]* If set to true (default), the Reference_Alarm directive is always used at problem alarm creation. If set to false, the Reference_Alarm directive might not be used at problem alarm creation, depending on the value of copyReferenceAlarmWhenNotPbAlarm (see below).
copyReferenceAlarmWhenNotPbAlarm	Boolean attribute	*OPTIONAL* *[TeMIP specific]* 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.

Table 11 - actions configuration

5.2.4 Trouble Ticket Actions

```
<troubleTicketActions>
  <troubleTicketAction name="TeMIP TT">
        <actionReference>TeMIP_TT_Directives_localhost</actionReference>
        <actionClass>com.hp.uca.expert.vp.pd.actions.TeMIPTroubleTicketActionsFactory</actionClass>
                <strings>
                  <string key="TT_SERVER entity"><value>TT_SERVER .SM</value></string>
                  <string key="CreateTemplateFile">
                        <value>createTroubleTicketByValueRequest.xml</value></string>
                  <string key="AssociateTemplateFile">
                        <value>associateTroubleTicketByValueRequest.xml</value></string>
                  <string key="CloseTemplateFile">
                        <value>closeTroubleTicketByValueRequest.xml</value></string>
                  <string key="DissociateTemplateFile">
                        <value>dissociateTroubleTicketByValueRequest.xml</value></string>
                  <string key="User"><value>temip</value></string>
                  <string key="Input"><value>input</value></string>
                  <string key="Type"><value>SYNCHRONOUS</value></string>
                </strings>
  </troubleTicketAction>
</troubleTicketActions>
```

The Problem Detection Framework supports the HP Service Manager through TeMIP.

If your Trouble Ticket system is HP Service Manager, then you can simply copy/paste the above example.

If your Trouble Ticket system is not HP Service Manager, an open API to define Trouble Ticket Actions Factory can be used. Please refer to **B.5** TroubleTicketActionsFactory implementation for explanation.

name	type	value
troubleTicketAction	property	Container for attributes defining the trouble ticket actions for a set of alarms
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	attribute	Unique reference that will be to define the routing information of a trouble ticket action
actionClass	attribute	The class implementing the *SupportedTroubleTicketActions* interface which describes the methods needed to support any Action on alarms. Methods such as createTroubleTicket, closeTroubleTicket...
strings	attribute	Container for a set of <string> key / value <string> specifying parameters for the interaction with the trouble ticketing system

Table 12 - Trouble ticket actions configuration

To know which Trouble Ticket System to use for an alarm the value of the tag is matched to the name of the troubleTicketAction.

Example:

tag="TeMIP TT"

<troubleTicketAction name= "TeMIP TT" >

For detailed explanation see Annex B.

5.3 Problem specific policies

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.

These configuration settings are different from the main configuration settings explained in the previous chapter: 5.2 “Main Policies” which apply to all problems defined in a Problem Detection Value Pack.

Problem Policies are configured in the same file where Main Policies are configured. This file is named “*ProblemXmlConfig.xml*” and it is located in the *src/main/resources/valuepack/conf/* folder.

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

Below is an example of such problem specific configuration settings, for a problem name “Problem_Power”:

```
<problemPolicy name="Problem_Power">
```

```
        <problemAlarm>
                <delayForProblemAlarmCreation>2000</delayForProblemAlarmCreation>
                <delayForProblemAlarmClearance>0</delayForProblemAlarmClearance>
        </problemAlarm>
```

```
        <troubleTicket>
                <automaticTroubleTicketCreation>false</automaticTroubleTicketCreation>
                <propagateTroubleTicketToSubAlarms>true</propagateTroubleTicketToSubAlarms>
                <propagateTroubleTicketToProblemAlarm>false</propagateTroubleTicketToProblemAlarm>
                <delayForTroubleTicketCreation>1000</delayForTroubleTicketCreation>
        </troubleTicket>
```

```
        <groupTickFlagAware>false</groupTickFlagAware>
```

```
        <!-- optional property, use carefully -->
        <sameGroupForAllProblemEntities>false</sameGroupForAllProblemEntities>
```

```
        <!-- optional property, use carefully -->
        <problemAlarmAbleToCreateGroup>true</problemAlarmAbleToCreateGroup>
```

```
        <!-- optional property, use carefully -->
        <enableTriggerConsistencyAfterResync>true</enableTriggerConsistencyAfterResync>
```

```
        <timeWindow>
                <timeWindowMode>None</timeWindowMode>
                <timeWindowBeforeTrigger>0</timeWindowBeforeTrigger>
                <timeWindowAfterTrigger>0</timeWindowAfterTrigger>
        </timeWindow>
```

```
        <!-- add the following lines only if you want to configure  -->
        <!-- the Operating Context where you want the problem alarm -->
        <!-- to be created.                                         -->
        <!-- This Operating Context will be used for                -->
        <!-- this specific problem only                             -->
        <strings>
              <string key="ocName"><value>MY_PROBLEM_OC</value></string>
```

```
        </strings>
```

```
    </problemPolicy>
```

For a complete example showing several <problemPolicy> tags, please consult the "*ProblemXmlConfig.xml*" folder of the Value Pack pd-example in Annex A.
It is located in the *src/main/resources/valuepack/conf/* folder.

5.3.1 Problem Alarm

In some cases, it may be useful to

- delay the creation of Problem Alarms.
- delay the clearance of the Problem Alarm.
- support the concept of nested Problems.

name	type	value
delayForProblemAlarmCreation	**long**	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.
delayForProblemAlarmClearance	**long**	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.
problemAlarmCanTriggerAnotherGroupForSameProblem	**boolean**	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 13 – Problem Alarm "per-problem" configuration

5.3.2 Trouble Ticket

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

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

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

Table 14 - Trouble Ticket "per-problem" configuration

5.3.3 Tick Flag awareness

The ***groupTickFlagAware*** parameter of type Boolean, when set to true, indicates that at regular tick intervals, the Problem Detection Value Pack, if customized for that, will execute some user code. The way to customize a Problem Detection Value Pack is explained in Chapter 6.

5.3.4 Multiple problem entities grouping policy

The ***sameGroupForAllProblemEntities*** property of type Boolean is optional. It only has a meaning if a trigger alarm has multiple problem entities (see B.2). 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.

5.3.5 Capacity for problem alarms to create groups

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.

The ***problemAlarmAbleToCreateGroup*** property of type Boolean is optional. 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 subalarms, will be discarded.

5.3.6 Capacity for Problem Detection to supersede a trigger alarm

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, the **enableTriggerConsistencyAfterResync** optional property of type Boolean should be set to 'false'.

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

5.3.7 Time window

Please see chapter 4.3 "Configure the Time Window" for more information on problem specific time windows.

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. **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. **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**	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**	Delay, in milliseconds, after the Trigger's alarmRaisedTime field to consider an Alarm as part of the Trigger's problem. Value is. Default is 30000.

Table 15 – Time Window "per-problem" configuration

5.3.8 Customization (refer to paragraph 6.1.1 first)

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

As explained in paragraph 6.1.1, it is possible to put assign basic customization directives for a specific problem (XmlGeneric_Synch in above extract).

5.4 Value Pack configuration

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 6.1.4

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>
```

Chapter 6

How to extend Problem Detection default behavior

Once configured (see Chapter 5), 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 uses cases where modification or extension is required, Problem Detection offers the flexibility to change the default behavior.

The ways to customize the default behavior are described in section 6.1. The default behavior is presented in section 6.2.

6.1 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 6.2 Default Behavior. The way to override those java methods is presented in section 6.1.2.

The way to modify the Problem Detection Value Pack default behavior by writing XML code is presented in section 6.1.1 below

6.1.1 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 6.2 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 5.1Filters, Table 10 - possible roles for 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.

6.1.2 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 6.2. There are three levels of customization:

Per problem (this section)

For a set or for all problems (section 6.1.3 “My ProblemDefault”)

For non-problem specific matters (section 6.1.4 “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.

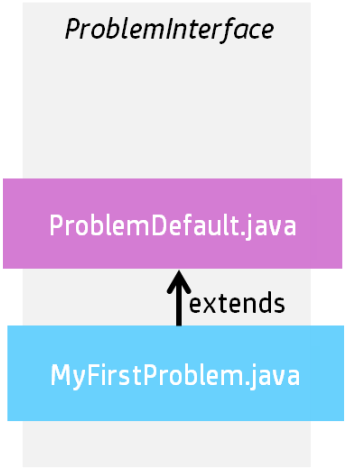


Figure 7 - One problem specific customization

Problemdefault.java is the class implementing the methods of the ProblemInterface. 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 under *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.

The Problem Detection framework will automatically invoke the methods whatToDoWhenXXX(…) listed in section 6.2, 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 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.

The Figure 8 below shows a graphical representation of the methods that will be invoked based on the lifecycle of the alarm.

In Figure 8, there are 3 alarms, ‘a1’, ‘a2’ and ‘a3’

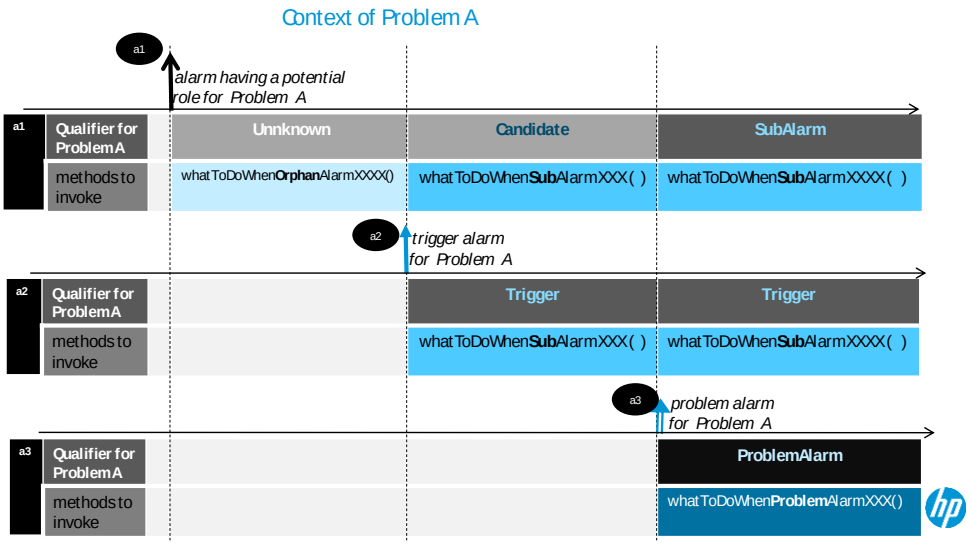
‘a1’ belongs to “Problem A” and “Problem B”

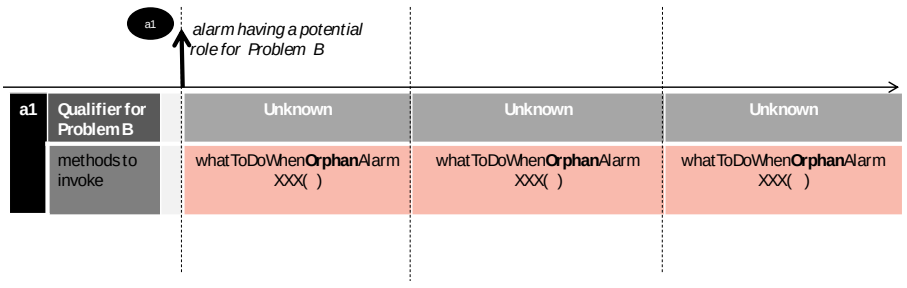
‘a2’ is a trigger alarm and belongs to “Problem A” only

‘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





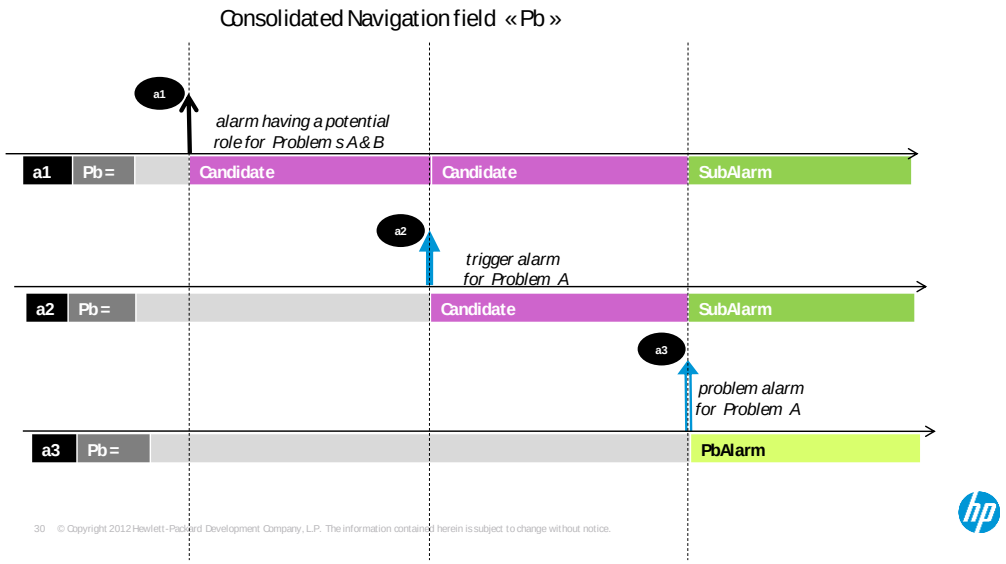


Figure 8 - Consolidation of alarm's qualifiers

6.1.3 My ProblemDefault

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

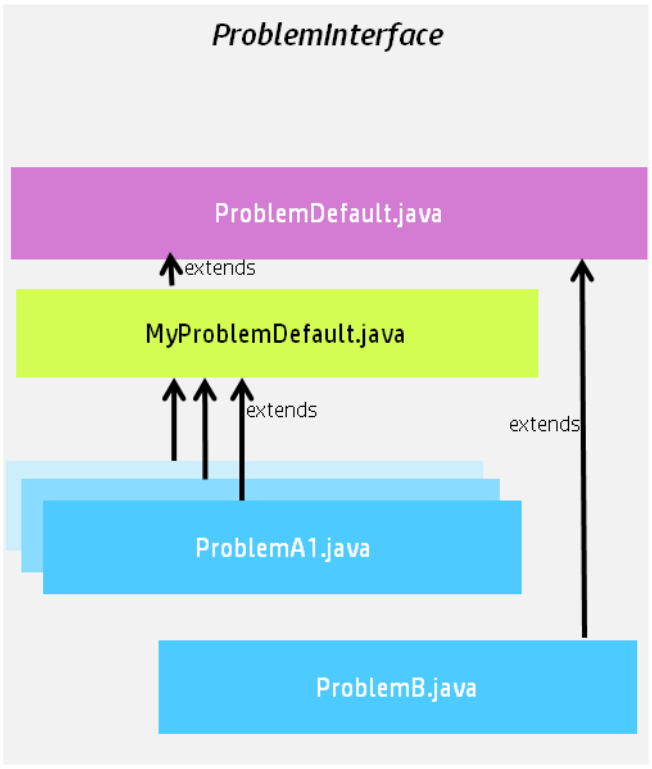


Figure 9 - MyProblemDefault: a customization for a group of problems

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. 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 B.1.

6.1.4 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 to

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 *problemClassName* in the file *context.xml* in *src/main/resources/valuepack/conf/* folder matches MyGeneralBehavior , as shown in *Figure 10 - MyGeneralBehavior name matching*

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

Figure 10 - MyGeneralBehavior name matching

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

```
public class MyGeneralBehavior extends ProblemDefault implements
                GeneralBehaviorInterface {

        /**
         *
         */
        public MyGeneralBehavior() {
                super();
                setLog(Logger.getLogger(MyGeneralBehavior.class));
        }

        /*
         * @see
         *
com.hp.uca.expert.vp.pd.core.ProblemDefault#whatToDoWhenNewAlarmIsJustInserted
         * (com.hp.uca.expert.alarm.Alarm)
         */
        @Override
        public void whatToDoWhenNewAlarmIsJustInserted(Alarm alarm) {

                if (getLog().isDebugEnabled()) {

                        getLog().debug(
                "whatToDoWhenNewAlarmIsJustInserted(): new alarm inserted : "
                                                + alarm.getIdentifier());
                }
```

```
            Flag flag = new Flag("JustInserted: "+alarm.getIdentifier(),
                              "Flag checking
whatToDoWhenNewAlarmIsJustInserted()", true);
            getScenario().getSession().insert(flag);

      }
```

6.1.5 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 II 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 III.a 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 III.b 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);

        }
```

6.2 The default behavior explained

As seen in previous paragraph 6.1How *to* customize default behavior, 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 A.4 pd-example, content of src/test/resources) The code of each of these methods is executed for every problem for which that method has not been overridden

6.2.1 Alarm Role Check

isMatchingCandidateAlarmCriteria(Alarm)

isMatchingProblemAlarmCriteria(Alarm, Group)

isMatchingSubAlarmCriteria(Alarm, Group)

isMatchingTriggerAlarmCriteria(Alarm)

6.2.2 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)

6.2.3 Common Entity Check

Methods used to calculate Information for optimizations

compareProblemEntity(Alarm, Group)

computeProblemEntity(Alarm)

computeProblemKey(ProblemContext, Alarm)

isInformationNeededAvailable(Alarm)

6.2.4 Group update

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

whatToDoWhenProblemAlarmIsAttachedToGroup(Group)

whatToDoWhenSubAlarmIsAttachedToGroup(Alarm, Group)

whatToDoPeriodicallyForAGroup(Group)

6.2.5 NetworkState Update

Methods used to manage the ProblemAlarm lifecycle, and its consequence

calculateIfProblemAlarmhasToBeCleared(Group)

whatToDoWhenProblemAlarmIsCleared(Group)

Methods used to manage the Sub Alarm lifecycle, and its consequence
whatToDoWhenSubAlarmIsCleared(Alarm, Group)

Methods used to manage the Orphan Alarm lifecycle, and its consequence
whatToDoWhenOrphanAlarmIsCleared(Alarm)

6.2.6 OperatorState Update

Methods used to manage the ProblemAlarm lifecycle, and its consequence

whatToDoWhenProblemAlarmIsAcknowledged(Group)

whatToDoWhenProblemAlarmIsUnacknowledged(Group)

whatToDoWhenProblemAlarmIsTerminated(Group)

Methods used to manage the SubAlarm lifecycle, and its consequence

whatToDoWhenSubAlarmIsAcknowledged(Alarm, Group)

whatToDoWhenSubAlarmIsUnacknowledged(Alarm, Group)

whatToDoWhenSubAlarmIsTerminated(Alarm, Group)

Methods used to manage the Orphan Alarm lifecycle, and its consequence

whatToDoWhenOrphanAlarmIsAcknowledged(Alarm)

whatToDoWhenOrphanAlarmIsUnacknowledged(Alarm)

whatToDoWhenOrphanAlarmIsTerminated(Alarm)

6.2.7 ProblemState Update

Methods used to manage the Trouble Ticket lifecycle when related to a Problem Alarm , and its consequence

whatToDoWhenProblemAlarmIsHandled(Group)

whatToDoWhenProblemAlarmIsReleased(Group)

whatToDoWhenProblemAlarmIsClosed(Group)

isAllCriteriaForTroubleTicketCreation(Group)

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

whatToDoWhenSubAlarmIsHandled(Alarm, Group)

whatToDoWhenSubAlarmIsReleased(Alarm, Group)

whatToDoWhenSubAlarmIsClosed(Alarm, Group)

Methods used to manage the Trouble Ticket lifecycle when related to an Orphan Alarm, and its consequence

whatToDoWhenOrphanAlarmIsHandled(Alarm)

whatToDoWhenOrphanAlarmIsReleased(Alarm)

whatToDoWhenOrphanAlarmIsClosed(Alarm)

6.2.8 Attribute Update

Method used to manage the ProblemAlarm Severity Update, and its consequence

whatToDoWhenProblemAlarmSeverityHasChanged(Group)

calculateProblemAlarmSeverity(Group)

Method used to manage the SubAlarm Severity Update, and its consequence

whatToDoWhenSubAlarmSeverityHasChanged(Alarm, Group)

Method used to manage the Orphan Alarm Severity Update, and its consequence.
whatToDoWhenOrphanAlarmSeverityHasChanged(Alarm)

Methods used to manage attribute update

whatToDoWhenProblemAlarmAttributeHasChanged(Group, AttributeChange)

whatToDoWhenSubAlarmAttributeHasChanged(Alarm, Group, AttributeChange)

whatToDoWhenOrphanAlarmAttributeHasChanged(Alarm, AttributeChange)

6.2.9 Periodic Check

whatToDoPeriodically()

whatToDoPeriodicallyForAnAlarm(Alarm)

6.2.10 Alarm eligibility update

whatToDoWhenProblemAlarmIsNoMoreEligible(Group)

whatToDoWhenSubAlarmIsNoMoreEligible(Alarm, Group)

whatToDoWhenOrphanAlarmIsNoMoreEligible(Alarm)

Chapter 7

Value Pack creation

This chapter prepares you to quickly build a Problem Detection Value Pack.

The pre-requisite is the installation of the UCA for EBC Problem Detection Development Kit which is comprised of

- UCA for EBC Development Kit (see UCA for EBC Value Pack Development Guide)
- UCA for EBC Development Kit Problem Detection Extension

7.1 Eclipse plug-in / new Problem Detection Value Pack

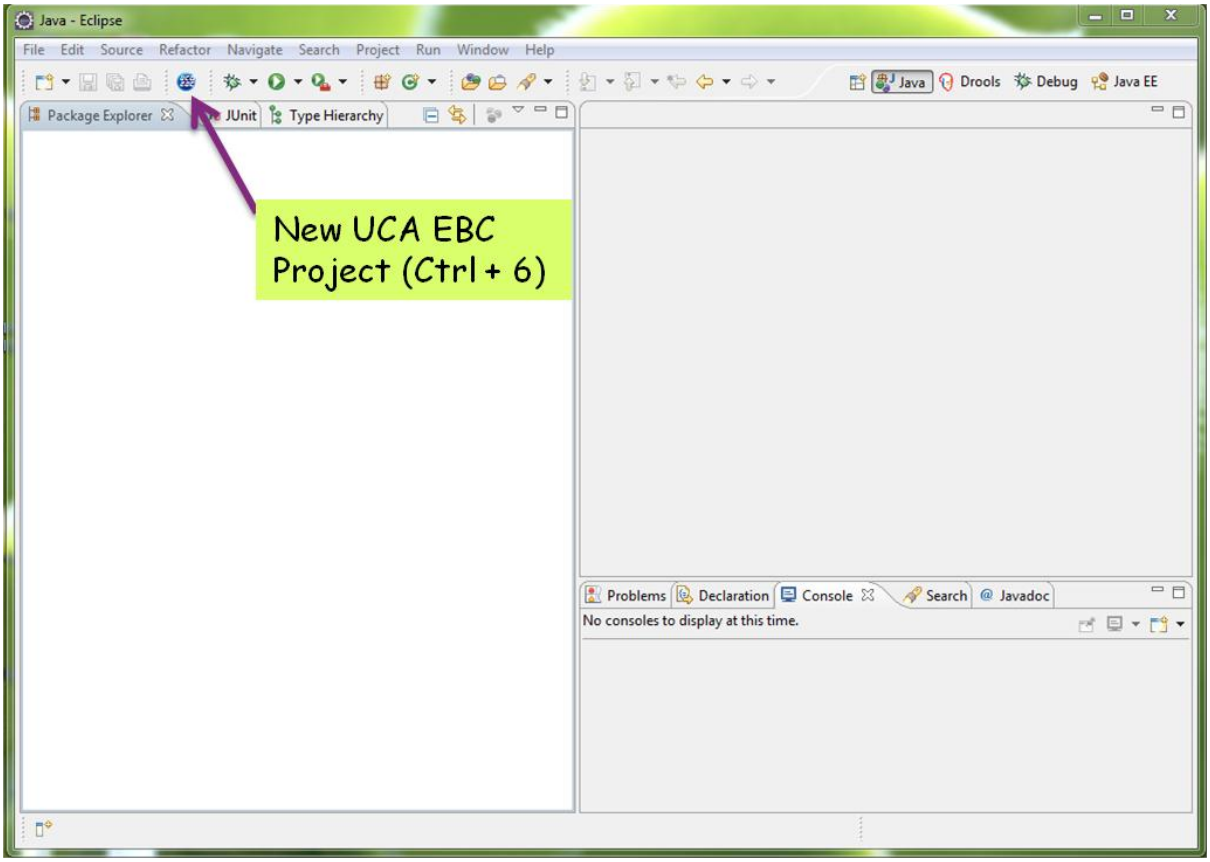


Figure 11 - How to create a UCA EBC project in Eclipse

Step 1: Once the Problem Detection Development Kit is installed, in Eclipse, click on the “New UCA EBC Project” button.

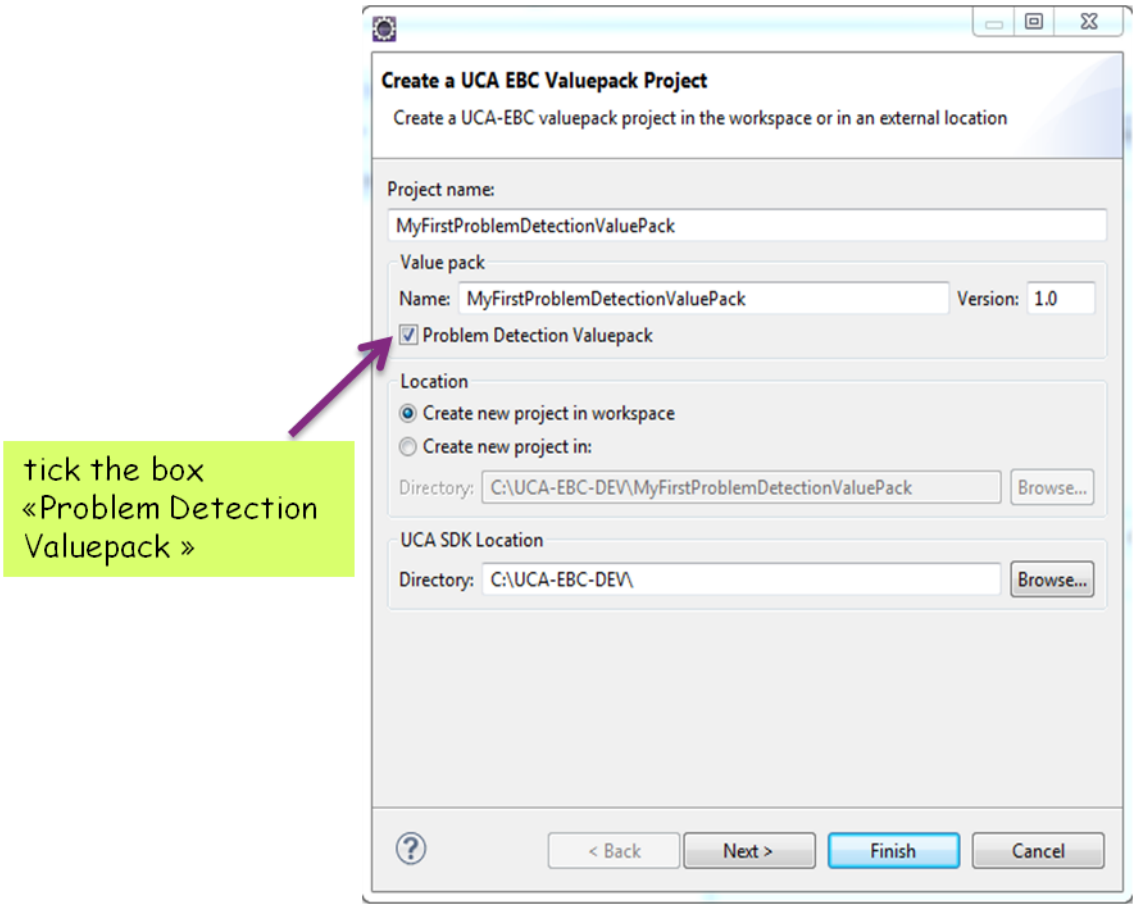


Figure 12 - How to create a UCA EBC Problem Detection Value Pack project in Eclipse

Step 2: Choose a name for the project and a name for the Problem Detection Value Pack. Problem Detection Valuepack tick box must be ticked.

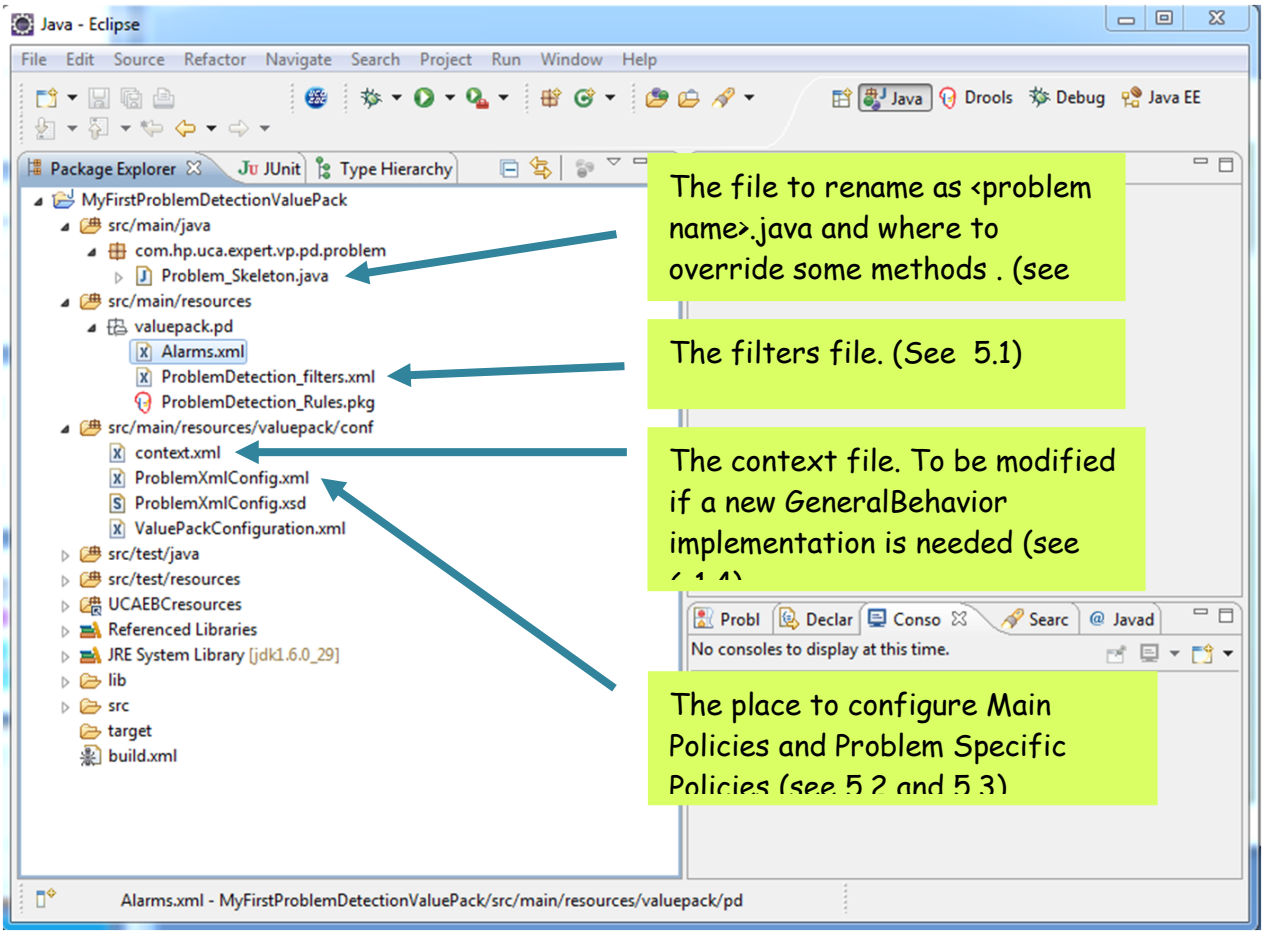


Figure 13 - Files to edit to configure MyFirstProblemDetectionValuePack

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.

7.2 Simulation

It is possible and even quite easy to check the correctness of a Problem Detection Value Pack before actually building and deploying it.

Developing a Problem Detection 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.

For detailed explanation on how to unit test your Value Pack, please see Chapter 3.10 of HP Unified Correlation Analyzer for Event Based Correlation **Value Pack Development Guide**

The screen shot below points to a test skeleton named *MyProblemTest.java*.

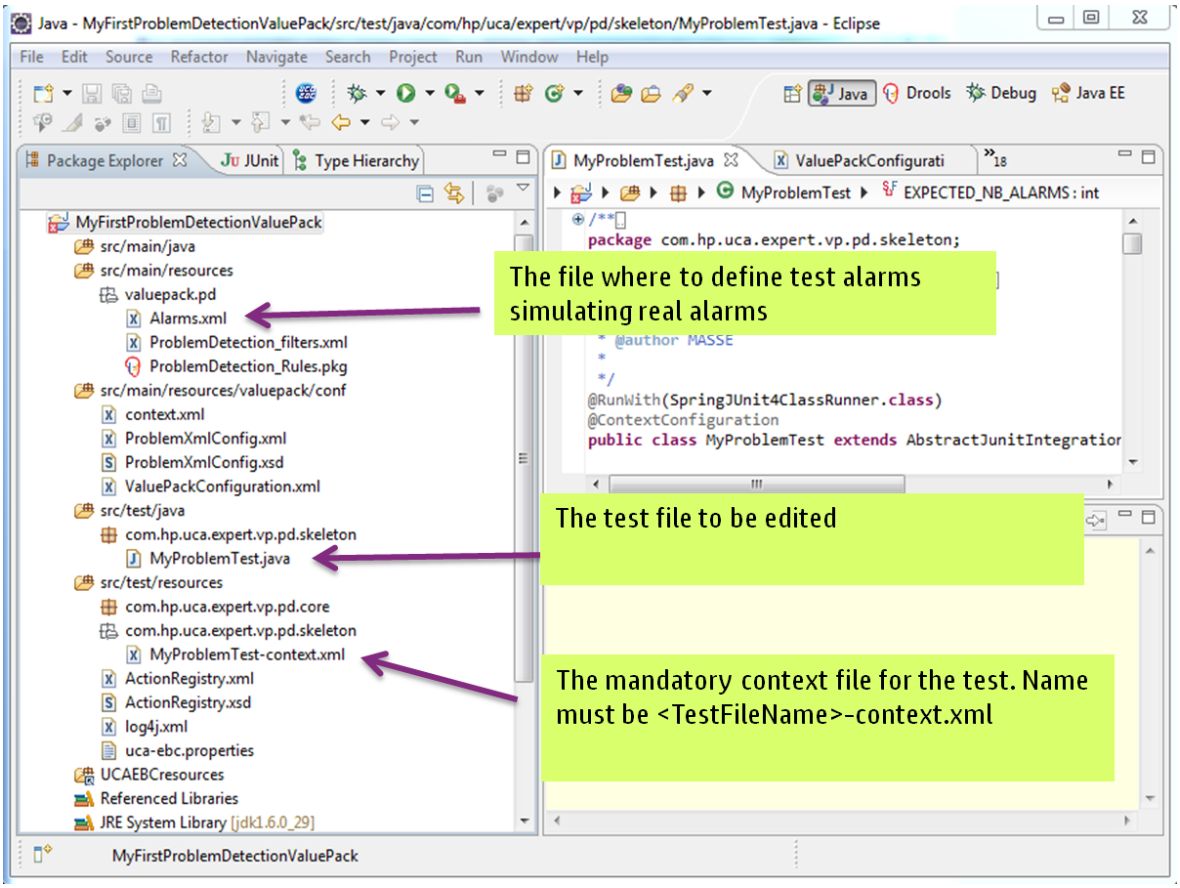


Figure 14 - Files to modify to create a JUnit test

Note that log4j.xml visible in "*Figure 14 - Files to modify to create a JUnit test*" is the place where to configure the level of logging for the JUnit tests such as MyProblemTest.java

7.3 Dynamic configuration update

It is possible to reload the filters and the configuration of a Problem Detection Value Pack by reloading a Problem Detection Value Pack scenario using the UCA for EBC GUI.

☞ Please refer to the [R4] *Unified Correlation Analyzer for Event Based Correlation User Interface Guide* for information on how to reload a UCA for EBC scenario using the UCA for EBC GUI.

7.4 Logging

Like for any UCA for EBC Value Pack, the logging configuration for a Problem Detection 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 Problem Detection loggers is given below:

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.PD_AlarmRecognition	Controls the decoding and setting of the roles of alarms
com.hp.uca.expert.vp.pd.core.PD_Lifecycle	Controls the states propagation methods
com.hp.uca.expert.vp.pd.core.PD_TroubleTicket	Controls the emission of Trouble Ticket requests
com.hp.uca.expert.vp.pd.core.PD_Navigation	Controls the requests for updates on alarms
com.hp.uca.expert.vp.pd.core.PD_Process	Controls the execution of operations of Problem Detection at a high level,(attaching a subalarm to a group, creating a Trouble Ticket, …)
com.hp.uca.expert.vp.pd.core.ProblemDetection	Controls the execution of operations of Problem Detection 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.services.PD_Service_Lifecycle	Controls Problem Detection Internals
com.hp.uca.expert.vp.pd.services.PD_Service_ProblemAlarm	Controls Problem Detection Internals
com.hp.uca.expert.vp.pd.services.PD_Service_Util	Controls Problem Detection Internals
com.hp.uca.expert.vp.pd.services.PD_Service_Navigation	Controls Problem Detection Internals
com.hp.uca.expert.vp.pd.services.PD_Service_Action	Controls Problem Detection Internals
com.hp.uca.expert.vp.pd.services.PD_Service_TroubleTicket	Controls Problem Detection Internals
com.hp.uca.expert.vp.pd.actions.TeMIPActionsFactory	Controls the actions that will be sent to TeMIP
com.hp.uca.expert.vp.pd.actions.TeMIPActionsFactoryCallbacks	Controls the constructions of callbacks that will be invoked by the UCA EBC framework (triggered by TeMIP responses)
com.hp.uca.expert.vp.pd.actions.TeMIPTroubleTicketActionsFactory	Controls the actions that will be sent to the Trouble Ticket System through TeMIP

Logger	Description
com.hp.uca.expert.vp.pd.actions.TeMIPTroubleTicketActionsFactoryCallbacks	Controls the constructions of Trouble Ticket callbacks that will be invoked by the UCA EBC framework

In addition to these Problem Detection 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

7.5 Monitoring

Please refer to the Unified Correlation Analyzer for Event Based Correlation – User Interface Guide and to the Unified Correlation Analyzer for Event Based Correlation – Reference Guide for more information on monitoring.

Chapter 8

Value Pack deployment

As a Problem Detection Value Pack is installed, deployed, started like any UCA for EBC Value Pack, this Chapter is very similar to chapter 3.5 of the UCA for EBC Reference Guide

8.1 Installing a Value Pack

Like any UCA-EBC Value Pack, a Problem Detection Value pack is a packaged as a zip file generated using the UCA for EBC Development toolkit.

To install a Problem Detection Value Pack, you need to copy the zip file to the *${UCA_EBC_HOME}/valuepacks* directory (*). No other action is needed to install a value pack. UCA for EBC server will automatically detect the newly installed Value pack. This value pack will then be visible from the UCA for EBC GUI Dashboard UCA for EBC > Application > Monitoring.

Please refer to the Unified Correlation Analyzer for Event Based Correlation – User Interface Guide for all GUI Administration features.

Note

(*) Since UCA-EBC V3.1 it is possible to directly upload your value pack from your development station to the UCA-EBC server through the Web GUI.

8.2 Deploying a Value Pack

Deploying a value pack can be done either from the command-line or the GUI of UCA for EBC:

From the command line, please execute the following commands (you need to be logged as the “uca” administration user):

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

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

From the Web GUI.

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

8.3 Starting a Value Pack

Starting a value pack can be done either from the command-line or the GUI of UCA for EBC:

From the command line, please execute the following commands (you need to be logged as the “uca” administration user):

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

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

From the Web GUI

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

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

8.4 Stopping a Value Pack

Stopping a Value Pack can be done either from the command-line or the GUI of UCA for EBC:

From the command line, please execute the following commands (you need to be logged as the “uca” administration user)

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

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

From the Web GUI

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

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

8.5 Undeploying a Value Pack

Un-deploying a Value Pack can be done either from the command-line or the GUI of UCA for EBC:

From the command line, please execute the following commands (you need to be logged as the “uca” administration user)

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

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

From the Web GUI

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

Undeploying a Value Pack performs the following actions:

- It removes the Value Pack from the *${UCA_EBC_HOME}/deploy* directory

It makes an archive ZIP file of the Value Pack and stores it in the *${UCA_EBC_HOME}/archive* directory

Annex A.

Value Pack example

As part of the Problem Detection 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

The filters for those four problems are present.

There is customization Java code for Problem_BitError, Problem_Synch, and Problem_Power.

There is customization XML for XmlGeneric_Synch.

There are examples of alarm enrichment, action factory and trouble ticket action factory definition.

There are examples of tests file that can be run with JUnit. 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.

A.1. pd-example, content of src/main/java

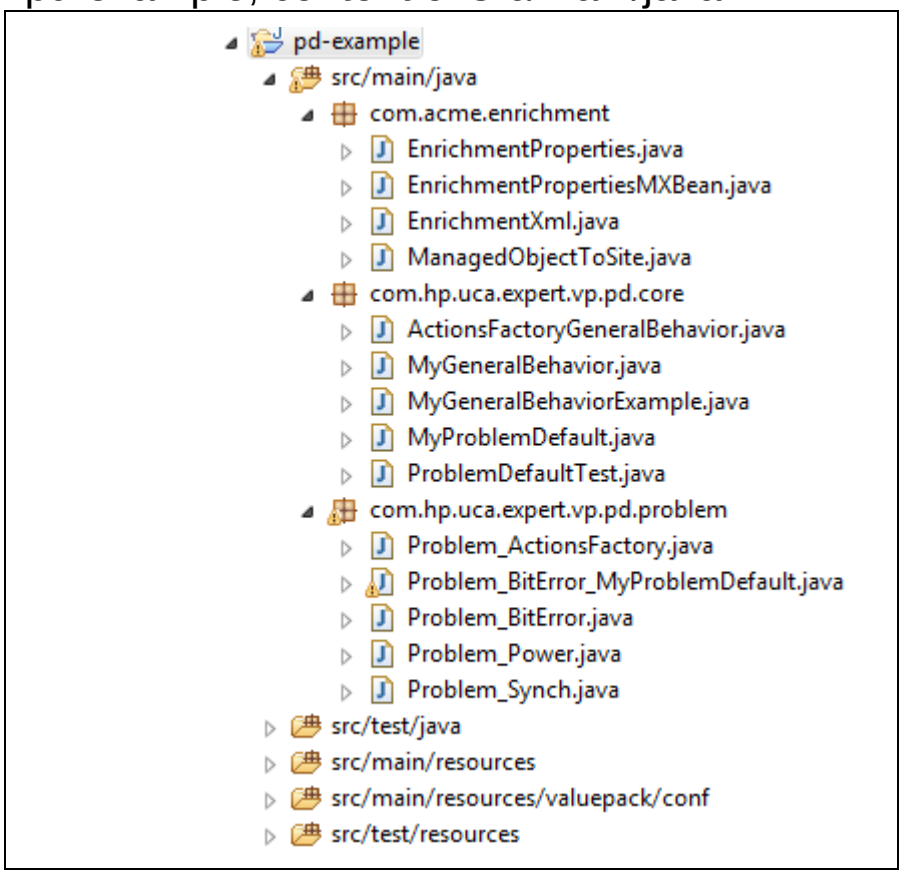


Table 16 - src/main/java: the customization code for the example Value Pack

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 6.1.4

MyProblemDefault.java illustrates methods of the ProblemInterface being overridden for a subset of problems. See 6.1.3

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

A.2. 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.

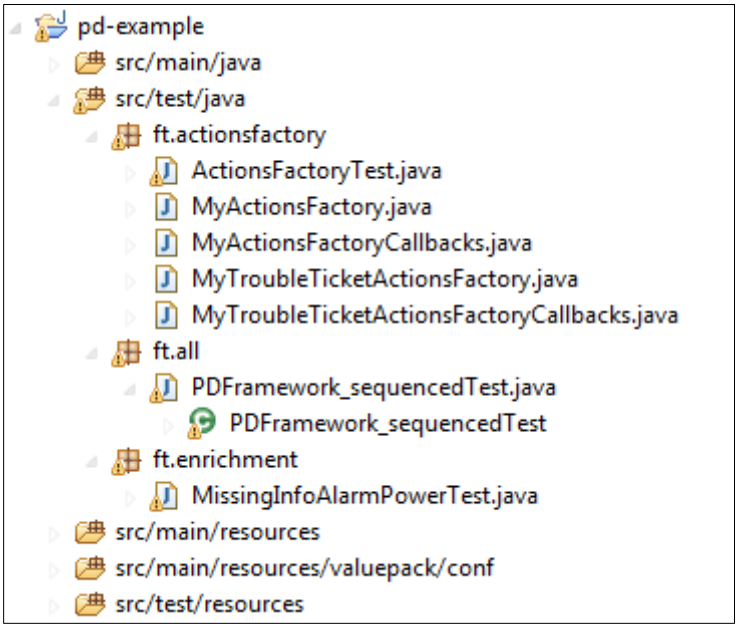


Table 17 - src/test/java: the source code of the tests

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.

A.3. pd-example, content of src/main/resources

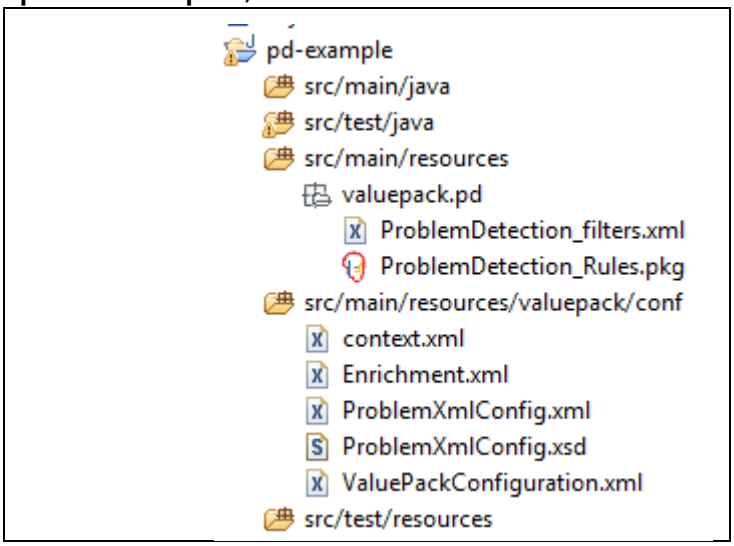


Table 18 - src/main/resources: the configuration files of the example Value Pack

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

A.4. pd-example, content of src/test/resources

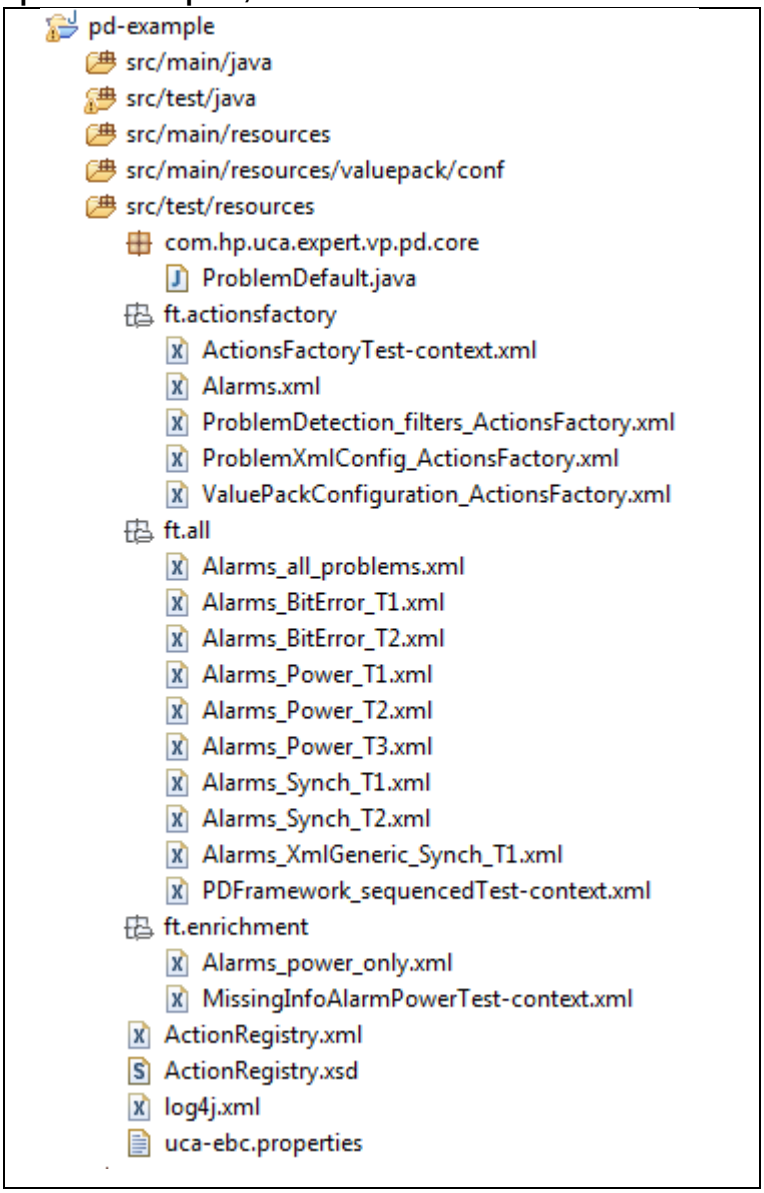


Table 19 - src/test/resources: the tests configuration files

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

ProblemDefault implementation

Located under `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 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 UCA for EBC Administration, Configuration and Troubleshooting Guide, and in particular to the 'ActionRegistry.xml file configuration' chapter

Annex B.

Advanced customization

B.1. Problem Detection behavior customization

As seen in chapter 6.1.3 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

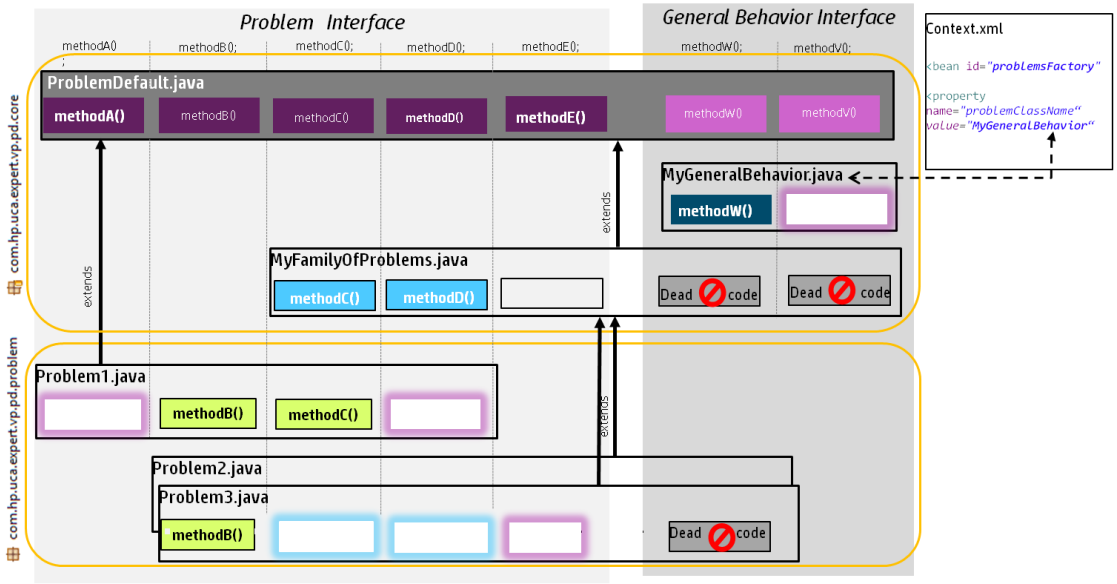
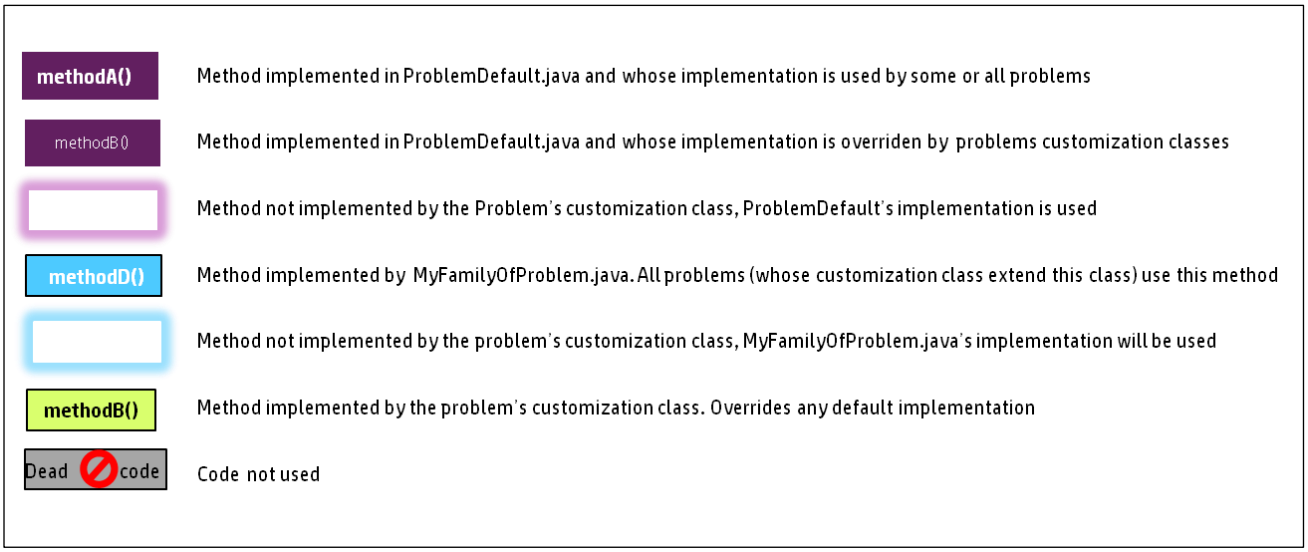


Figure 15 - schema of implementation of the main Problem Detection interfaces



B.2. 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.

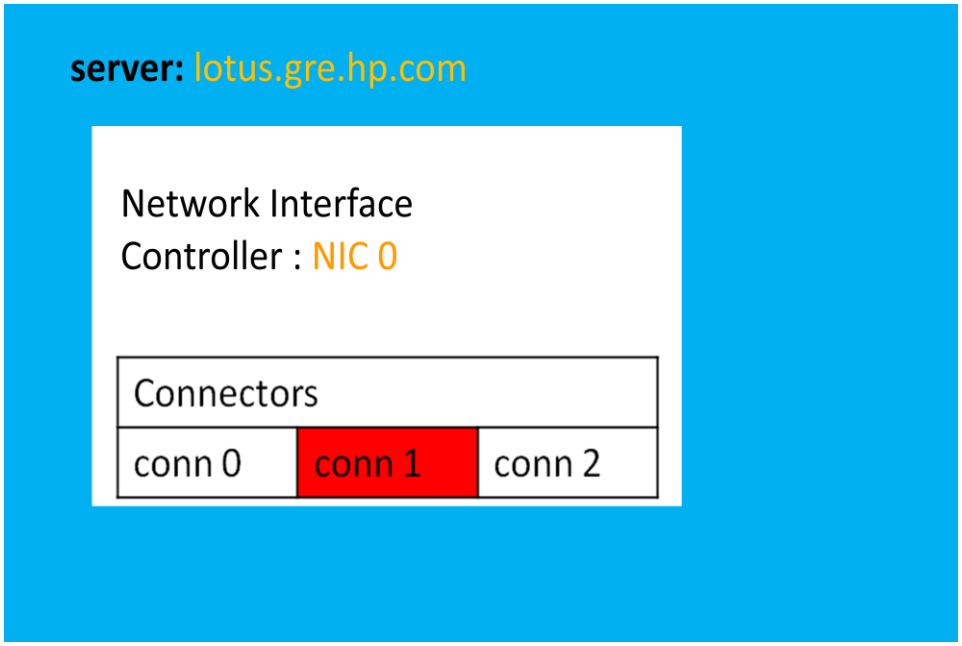
Case 2 – All the alarms have the same [problem key] and a similar {problem entity}

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

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

B.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 Problem Detection – JavaDoc Problem Detection framework (C:\%UCA_EBC_DEV_HOME%\apidoc\uca-evp-pd-fwk\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.

B.3.1 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 commandsin 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.

B.4. 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

who 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>
```

B.5. 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)

B5.1 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.

B5.2 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>
```

B5.3 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 Problem Detection – JavaDoc Problem Detection framework*
(C:\%UCA_EBC_DEV_HOME%\apidoc\uca-evp-pd-fwk\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.