HP Software

Unified Correlation Analyzer

User Guide

Edition: 1.3

For the HP-UX Itanium Operating System

January 2010

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 U.S. trademark of Sun Microsystems, Inc.

Microsoft® , Windows® and Windows NT® are U.S. registered trademarks of Microsoft Corporation.

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.

Contents

Figures

Preface

This User Guide covers the following topics:

- An introduction to the concepts used in correlation for problem detection, service impact and root cause analysis
- A ‘quick start’ guide to starting up, configuring and shutting down the system.
- A description of the UCA architecture and the fundamental concepts at the heart of the system.
- Use of the UCA System Manager GUI.
- Defining the UCA metamodel.
- Creating the UCA model database.
- A detailed description of the UCA Scenario Manager and Mesh Viewer GUIs.
- A description of how to use the Scenario Manager GUI to create and deploy scenarios, filters, mappings and rules.
- An in-depth description of how to configure UCA rules and actions.
- A description of the UCA alarm interfaces.
- Reference information on object types and their attributes.

This guide forms part of the set of UCA documentation, the other guides are listed as part of the associated documents further in this guide.

Intended Audience

This document is aimed at the following personnel:

- Network Management Customers
- Solution Architects
- System Integrators
- Solution Developers
- Software Development Engineers

Supported Software

The supported software referred to in this document is as follows:

Product Version	Operating Systems
Unified Correlation Analyzer 1.0	HP-UX 11.31 for Itanium

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.

italicised red text:

- Important or particularly noteworthy information

 Hints and Tips e.g.

> it's a good idea to create a shortcut to this URL on the web browser's toolbar

- Hints displayed as a boxed text wth a 'thumbs up' graphic

Acronyms and definitions

The following acronyms are used in this documentation:

Acronym	Definition
ER	Early release (Beta version of the product)
MO	Managed Object
MR	Manufacturing Release
MSL	Management Specification Language
OC	Operation Context
OS	Operating System
TeMIP	Telecommunications Management Information Platform
UCA	Unified Correlation Analyzer

Associated Documents

- *HP UCA Installation and Configuration Guide*

- *HP UCA Advanced Configuration and Troubleshooting Guide*
- *HP UCA TeMIP Integration*
- *HP UCA TeMIP Client*

For a full list of TeMIP user documentation, refer to Appendix A of the TeMIP Product Family Introduction.

- *HP TeMIP Client Installation and Configuration Guide*
- *HP TeMIP Web Services Installation and Configuration Guide*
- *HP TeMIP Software Customization Guid*e.
- *TeMIP-Service Manager OSSJ Trouble Ticket Liaison – Installation & Configuration Guide*
- *TeMIP-Service Manager OSSJ Trouble Ticket Liaison - TeMIP Liaison Adapter System Integration Guide*
- *HP Service Manager – Installation Guide*

Support

Please visit our HP Software 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

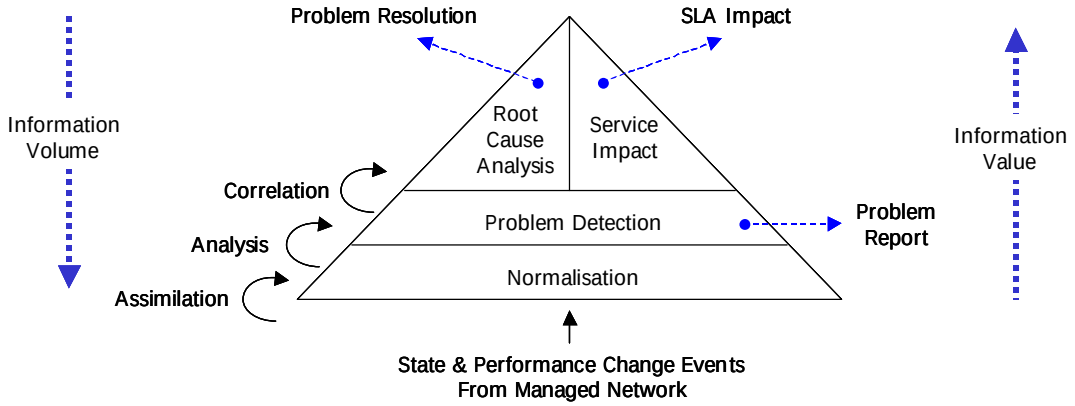
Chapter 1 Introduction

Managed networks exist everywhere – obvious examples include telecommunications networks, utilities providing water, gas and electricity and TV and radio broadcast networks.
Recognising that such networks are built from equipment that can fail, manufacturers of network components usually build in self-monitoring systems of various levels of complexity, or at least provide a capability for an external system to monitor their current status. Depending on the level of sophistication and redundancy built in to the network component, low level failures and errors may be handled automatically with only a cursory event report to the outside world that something has happened. On the other hand, less resilient equipment may deliver a constant stream of event reports as its status changes.
Regardless of the level of sophistication of the individual network components, a managed network will usually employ centralised or regionalised management capabilities to allow network operators to monitor the status and performance and to re-configure the network in response to changing operational needs or failures.
This arrangement works well if the managed network can be monitored and maintained by a reasonable number of experienced network operations personnel. Under these circumstances, human operators are responsible for correlating the streams of state change events and performance information received from individual network components and, based on their experience of operating that network under a range of operational and fault conditions, adjusting the operational parameters to provide the required level of service to their customers.
A major problem arises however when the size and complexity of the network exceeds the capability of the operators to correlate the streams of information received from it. In this situation, network operators often turn to event correlation systems in an attempt to automate some of the analysis workload and speed up fault resolution times.
Event correlation systems typically break down into two types:

- Out-of-the-box solutions, providing a range of standardised network and equipment models and problem analyses for commonly available technologies e.g. IP Communications Networks.
- Rule-based, low-level correlation toolkits, based on Inference Engine technology, suitable for constructing localised stream-based correlations.

Each of these types of system has their own advantages and disadvantages. The former are characterised by rapid deployment but at significant cost, targeted at specific technologies where the investment in developing the correlation solution is justified by the number of similar installations that may benefit from the technology. The major problem however is that the manufacturer determines the range of correlations available and developing user-defined correlations is often technically beyond the ability of the user. Users are also reliant on the supplier providing a continual stream of equipment models as new versions or types are introduced into the market place.
Users of low-level toolkit based solutions benefit from the ability to develop and deploy stream-based correlations from point sources in the network e.g. for event de-duplication or counting over time. Unfortunately, more complex correlations such as those requiring knowledge of the implicit relationships between network components and how their states change over a period of time, result in an explosion in complexity. Typically, the size of the rule base quickly becomes unmanageable and often requires additional, expensive software development to achieve the desired result.
UCA combines the best of both of these approaches, making use of data-driven network models and simple yet powerful high-level rules to achieve complex correlations. A user with problem domain knowledge can quickly and easily construct correlations for any type of network using the visual tools provided. This is achieved without having to invest in understanding proprietary technologies or recourse to complex rules and expensive and time-consuming software development.
The design of UCA takes as its starting point the mental process followed by an experienced network operator when trying to solve a particular problem. Typically, this process involves assimilating state and performance change events provided by the network management system into a conceptual model of the managed network and analysing the resulting mental picture to work out what the problem with the network is. Once this has been done, the underlying root cause of the problem can be investigated and resolved and the impact on managed services (and associated Service Level Agreements) determined through correlation. The operator will often have to take into account the diversity of

network equipment and variation in reported detail when assimilating event data - effectively applying a ‘normalisation’ process to the information received from the network management systems.
The following diagram summarises this process:



In essence, the network operator is acting as an information normaliser, analyser and correlator, condensing large volumes of low value information and generating small amounts of high value information e.g. what the problem is, what the root cause is (and how to fix it) and finally what the impact is on managed services.
UCA achieves the same result as the human operator - quickly, reliably, efficiently and automatically, vastly improving fault resolution times and reducing service impact.

Chapter 2 Quick Start Guide

This chapter provides a high-level guide to starting up and using the system. A detailed explanation for each of the features introduced in this section is provided in subsequent chapters.

2.1 Start-up

1. On the server, start the UCA server as follows:

```
# cd $UCA_HOME/bin
# uca_start
```

2. Using a web browser (such as Internet Explorer 6 or 7 or Firefox 2) on a client machine, navigate to the URL http://***hostname***:18080/**uca** where ***hostname*** is the DNS name or IP address of the server machine.

it's a good idea to create a shortcut to this URL on the web browser's toolbar

The UCA home page will be displayed (see below).
The two main buttons on this page are:

- **UCA Applications** – this is used to access to all authorised applications, according to role. e.g. the Scenario Manager and / or the Mesh Viewer (see later chapters for details).
- **UCA Manager** – this invokes the System Manager GUI (see below). A user must have manager role privilege to invoke this GUI.

In addition, the two links at the bottom left of the page are:

- **Manage Tomcat** – this is used to access the standard Tomcat Manager web page
- **Run ArgoUML** – this runs the ArgoUML design tool.

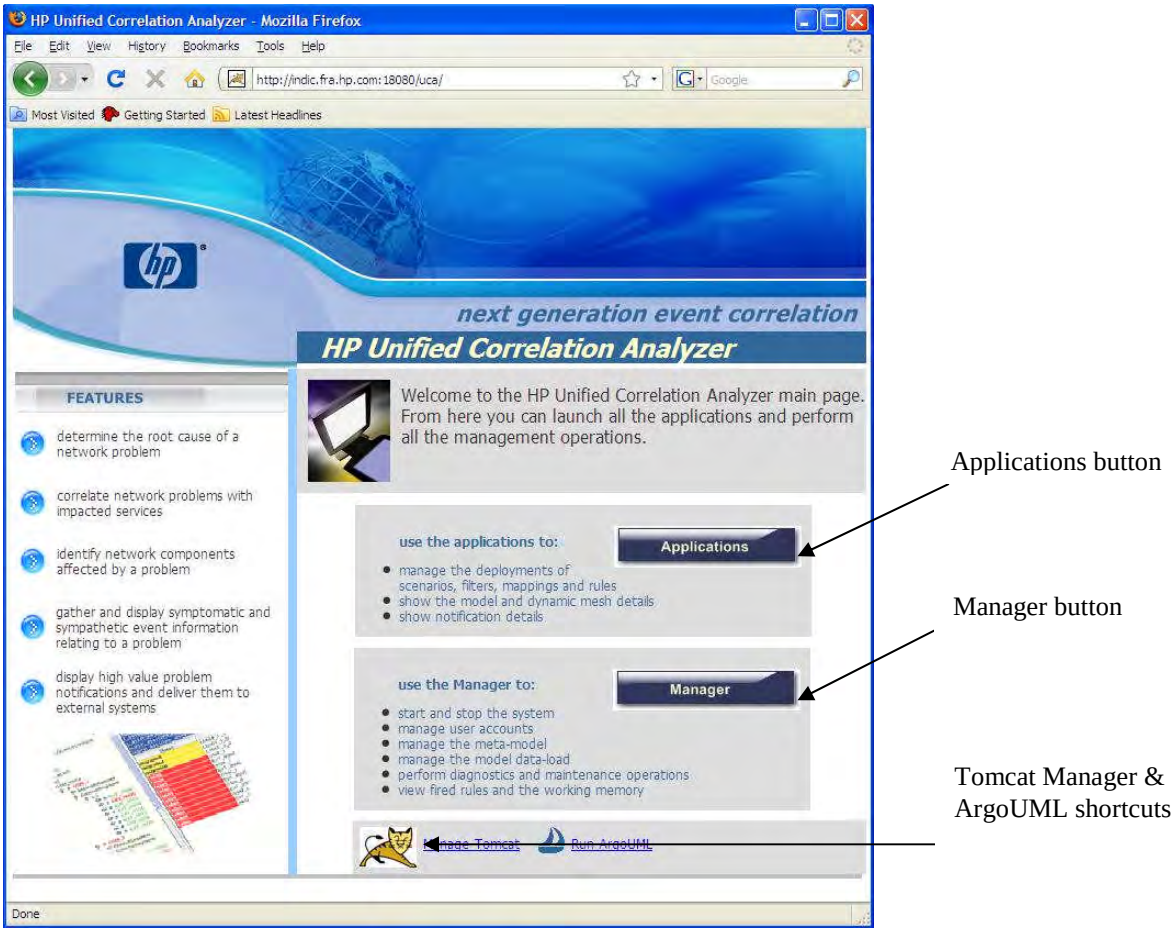


Figure 1 - The UCA home page

3. Click on the **UCA Manager** button.
4. Enter `system` as the username and `system` as the password.

Once logged in as 'system', it is strongly recommended that the username and / or password for the 'system' user is changed. A currently logged on user cannot modify their own details, so to do this, in the System Manager GUI select the Users tab, create a new user with manager role. Then exit the System Manager GUI and restart it, logging on as the newly added user. Finally select the original system user and enter the new username and / or password details and click on Update.

The UCA System Manager GUI will now be as shown below.

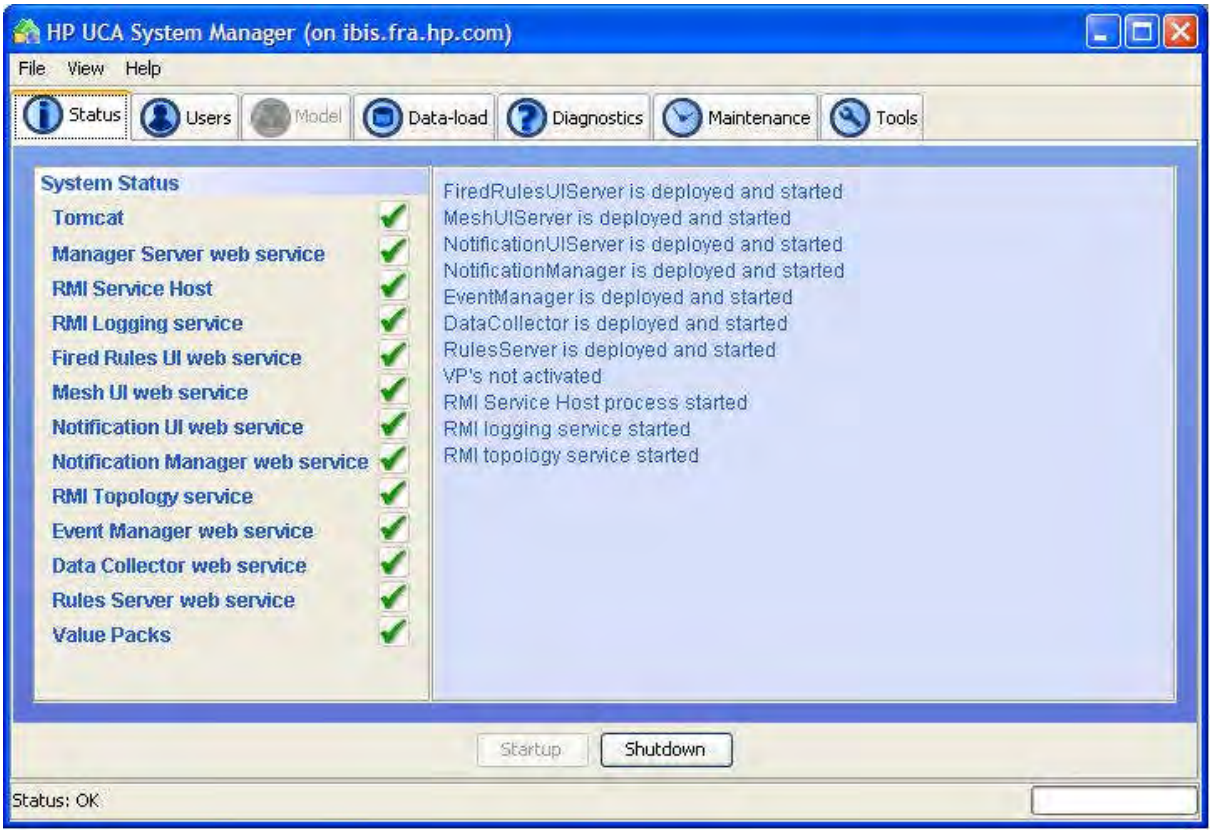


Figure 2 - The UCA System Manager

2.2 Basic System Configuration

1. From the **Users** tab, users may be added with relevant roles, modified or deleted as appropriate.
2. From the **Model** tab, new metamodels may be:

   - created manually
   - loaded from a local directory
   - saved to a local directory
   - imported from an XMI file (previously created with ArgoUML)
   - added to the metamodel Library,
   - deployed into active use

   Details are provided in later chapters

3. From the **Data-load** tab, model data may be loaded from CSV files into the model database. Details are provided in later chapters.

2.3 Running the UCA Applications

1. From the UCA home page click on the **UCA Applications** button.
2. Enter a valid username and password in the corresponding web page.
3. The UCA applications web page will be displayed showing the **Scenario Manager** button and / or the **Mesh Viewer** button, depending on the roles configured for the associated username.

4. Clicking on the **Scenario Manager** button will invoke the Scenario Manager GUI. This is used for creating and maintaining scenarios, filters, mappings and rules and deploying them into active use (see later chapters for details).
5. Clicking on the **Mesh Viewer** button will invoke the Mesh Viewer GUI. This is used for real-time monitoring of events within the UCA state mesh, viewing 'notifications' and viewing the model data (see later chapters for details).

2.4 Shutdown

1. From the **Status** tab, select **Shutdown**

Chapter 3 System Description

This chapter provides a detailed description of the UCA architectural components and describes the fundamental modelling concepts, with the aid of various examples, at the heart of UCA.
The UCA system architecture is shown in the following diagram:

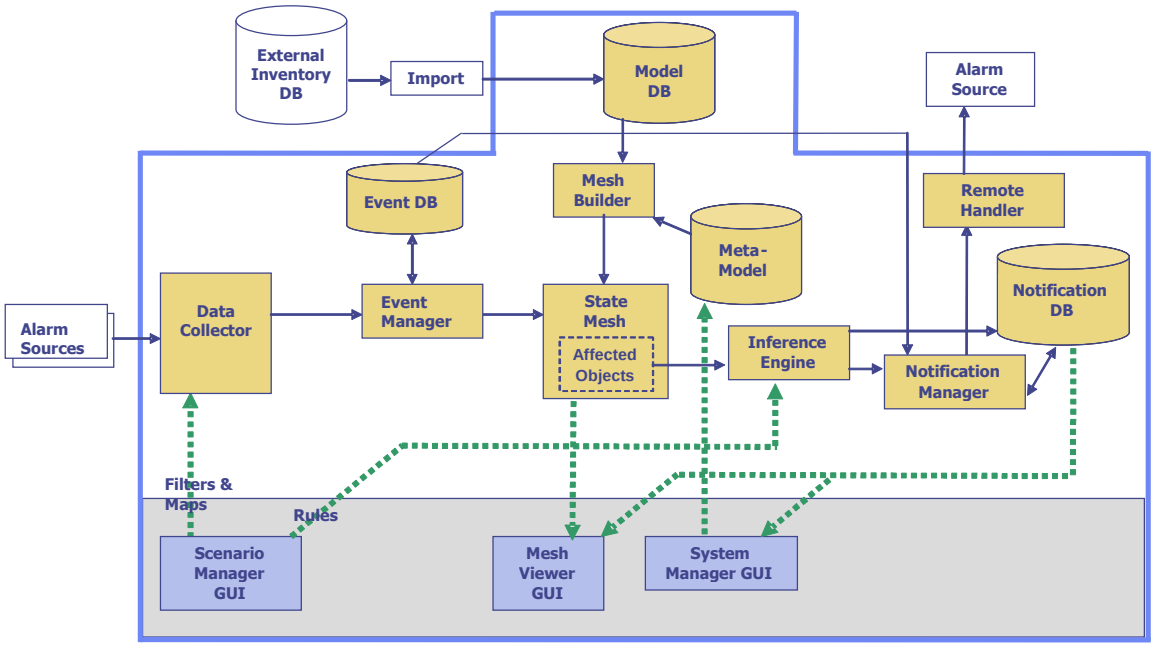


Figure 3 - UCA Architecture

The sections below provide a general description of the function and operation of each component of the system.

3.1 State Mesh

Fundamental to the operation of UCA is the state mesh. At its simplest level, it is a structural model of the managed network it is attached to. It incorporates a set of objects, each representing some (physical and / or logical) component of the managed network, linked together by a set of navigable (associative, containment and / or inheritance) relationships. On a more complex level, the state mesh is also a state model of the managed network, maintaining in real-time the current state of each modelled object and providing pre-defined or rule-driven paths for states to propagate between the component objects.
From the point of view of the user, the availability of the state mesh considerably simplifies the whole process of constructing rule-based event correlations. This is because it handles the following tasks that are traditionally the responsibility of the defined rule-base in simpler systems:

- Dynamic establishment and maintenance of relationships between modelled network components.
- Dynamic establishment and propagation of state information between modelled network components.

3.2 Mesh Objects

UCA provides a very flexible modelling capability and places no restrictions on the type of objects that can be modelled in the state mesh. In addition, there is no need for a one to one correspondence between the model types supported in UCA compared with those available in the event source because incoming events can be re-mapped to one or more destination objects of any specified type.

This degree of flexibility is achieved in part because UCA uses a single type of object – a mesh object – to provide the underlying implementation of any type of modelled entity. A mesh object is characterized by three attributes:

- Base Class – the fundamental or 'super-class' of entity that it represents e.g. Transmitter
- Sub Class – the specialized or 'sub-class' of entity that it represents e.g. Digital Transmitter, Analogue Transmitter
- Unique Reference – an identifier that uniquely identifies the object. Note that depending on mapping configuration and availability of information in the incoming event or an external source, this value may be a Fully Distinguished Name (FDN) i.e. unique throughout the entire system e.g. Site_66_Transmitter_3, or a Relative Distinguished Name (RDN) i.e. unique throughout all instances of objects of this base class relative to a parent object e.g. Transmitter_3, a child of Site_66.

This has considerable advantages for users because an event source, for example a network management system, can model monitored elements at a relatively coarse level, and events can be mapped to a more fine-grained model supported by UCA provided sufficient information is available e.g. in the event itself or an external database, to allow the mapping to occur. This potentially reduces the complexity (and cost) of implementation of a new event source system. It also removes the need to 're-engineer' an existing source system when a model is added or extended. Finally, it allows more complex analyses to be carried out by UCA than would otherwise be possible using the event source model alone.

UCA can also model elements from which events are not directly received by an event source e.g. a fibre connecting two ports, or a service implemented by a number of components that may never directly receive alarm or performance events. This capability allows UCA to build and maintain a complete correlation model. It is also possible for UCA to infer and modify the state of such objects and assign a problem 'root cause' or 'service impact' directly to them.

Because there are no restrictions on the types of objects that can be modelled, UCA is able to support objects that represent any kind of physical, logical, service or abstract entity. Examples of non-physical entities include timeslots on a communications link, a mobile network 'drive trial' carried out over a set of pre-defined network cells or a cross-domain service implemented from a number of network components and sub-services.

3.3 Mesh Object Relationships

The ability to flexibly model network entities is an important feature of UCA, however the value of such a modeling capability is limited without the corresponding ability to model relationships between those entities. For this reason, UCA provides comprehensive support for implementing relationships between mesh objects to complement those found in monitored networks. The types of relationship supported by UCA are described in the following sections.

3.3.1 Composition or Parent-Child

In composition relationships, one class of object is the parent of another and effectively 'owns' the child object. Another way to express this type of relationship is to consider the lifetime of the child object – if it cannot exist without its parent or should be destroyed when its parent is destroyed, then this is an example of such a relationship. An example of this might be Communication Ports (children) implemented by an Interface Card (parent) – the Ports cannot exist without the Card. A child object will always have a parent object and may itself have zero or more children of its own, although circular relationships are not allowed.

A parent object may have zero or more children of any number of types e.g. a Network Element might have the capacity for 10 Interface Cards and 2 PSUs and may be initially configured with a single PSU and no Interface Cards. The practical implementation within UCA is more flexible still, in that while a child object must have a parent (and can have one and only one parent at any time), the type or instance of parent object can be configured at state mesh build-time. This means that a child type can be configured with a choice of different types of parent object, with the actual type and instance being defined by the model data load. An example of this is a Network Element that may be parented by a Network i.e. standalone, or by another Network Element i.e. a slave element.

Significantly, child objects can also be 're-parented' by dynamically updating the parent type and / or object in the state mesh at runtime. Finally, child objects can be added and removed dynamically at

runtime, so in the example above, Interface Cards and a PSU can be added to the Network Element as they are configured into the actual network.

3.3.2 Aggregation or Uncle-Nephew

In aggregation relationships, one class of object (an uncle object) has an interest in the state of another sub-ordinate object and effectively 'contains' the nephew object. The important differentiator compared to composition is that both the uncle and nephew objects can exist independently of the other – the relationship implies a measure of optionality and is weaker.
As with compositions, an object that is a nephew may itself be an uncle of some other object and the relationships may be configured at build or runtime, although again circular relationships are not allowed.
The range of possible combinations of this type is wider than that provided by composition. A nephew object can have zero or more uncles and / or an uncle can have zero or more nephews. In a typical application an object will have a parent and may have one or more uncles – a good example of this situation is where a Bearer Link carries Voice and Signalling Channel traffic simultaneously in its Timeslots. The Bearer Link acts as the parent for the Timeslots – they cannot exist without it. At the same time, the Voice and Signalling Channels carried in the Timeslots act as uncles – they are interested in the state of the Timeslots but they are not the owners.

3.3.3 Association or Peer-Peer

In association relationships one object has an interest in the state of another object, but neither object has sufficient interest to warrant a composition or aggregation relationship. This type of relationship is the weakest that may exist between objects and again implies optionality.
One peer may be associated with zero or more peers of the same or different types. An example of a relationship of this type is that of a Cable joining two Communications Ports. The object representing the Cable is interested in the state of the Ports at each of its ends and an associative relationship would be used in this instance. Again, UCA provides the capability to construct associative relationships at build or run-time with the usual proviso that circular relationships are not allowed.

3.3.4 Specialization

This type of relationship is different from the previous three in that it is implemented as an attribute of the mesh object itself, rather than between instances. Each mesh object type possesses a Sub Class attribute that defines its specialization relative to other mesh objects of the same base class. This allows UCA to support some of the characteristics of object inheritance i.e. polymorphism and specialization. For example, there may exist in a monitored network a number of Transmitters with different Sub Classes e.g. 100W_Transmitter, 200W_Transmitter and 300W_Transmitter. Instances of each type are clearly Transmitters (base class = Transmitter) and the group of all affected Transmitter objects may be subject to rules that operate at the base class level i.e. they are treated as polymorphs and their specialization is ignored. Alternatively, more detailed rules may be defined to operate only on instances of a single specialization by defining the required Sub Class condition as well.

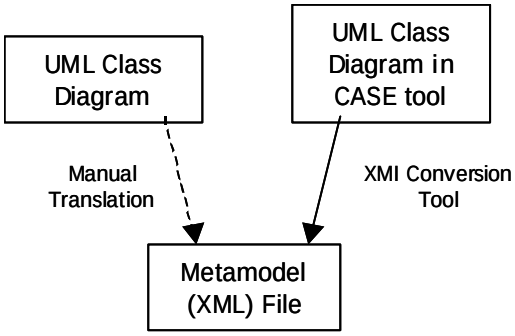
3.4 Metamodel

While the state mesh is of considerable value in reducing solution complexity, this advantage would be lost if the model had to be re-implemented by the user each time a new or updated model was required. For this reason, UCA uses an automatic data-driven approach to its construction and maintenance Central to this idea is the metamodel that defines for the state mesh:

- all possible **classes** or **types** of model object that it could contain
- all possible **relationships** that could exist between **classes** of model object
- all possible pre-defined **state propagations** that could exist between **classes** of model object

The best method to capture the metamodel structure during system configuration is for the user to construct a UML class diagram (with some additional stereotypes defined to handle state propagation). The file containing the metamodel is then simply an XML representation of that class diagram and the required syntax is described fully in later sections of this guide. Users are free to manually define their own metamodel directly in XML. Alternatively, UCA provides the capability to automatically convert

a UML class diagram (exported in XMI format from a suitable UML modelling tool) directly into the required XML format. This process is illustrated below.

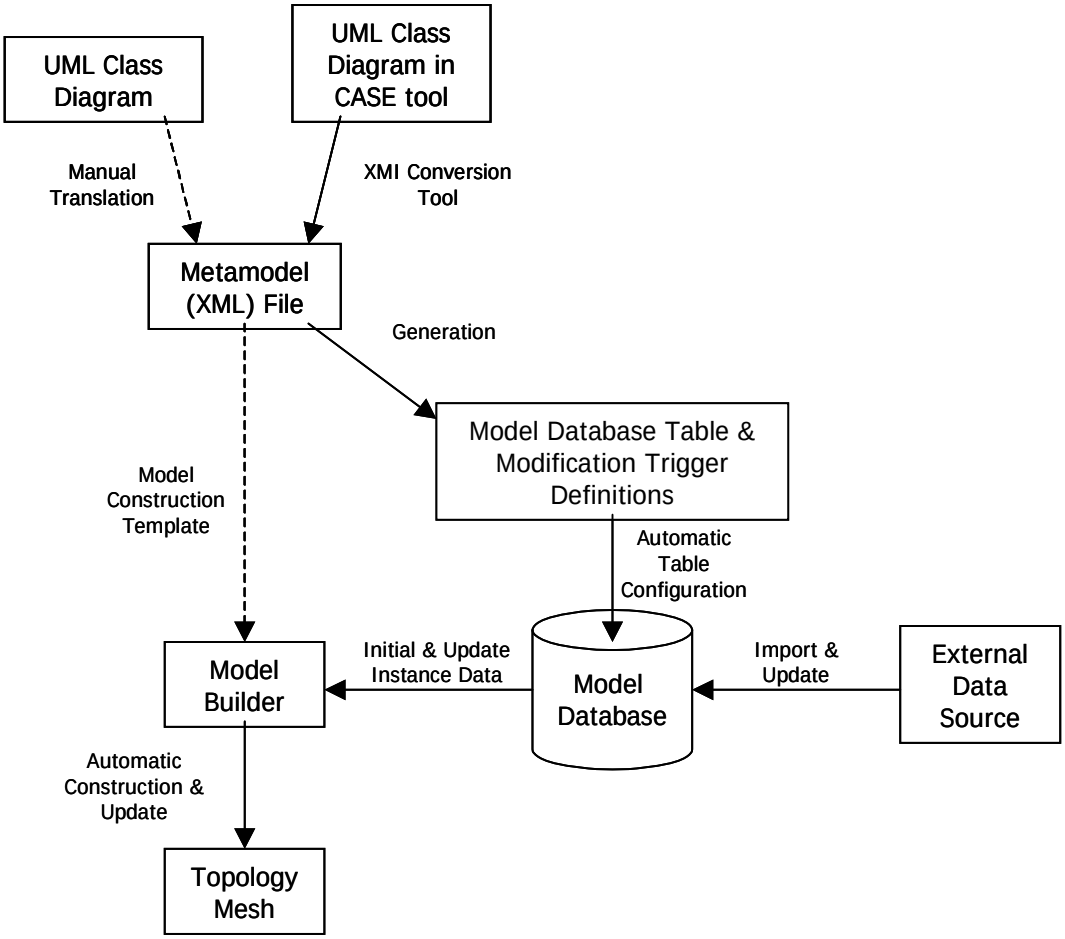


3.5 Model Builder and Model Database

The metamodel by itself defines only those model classes, relationships and automatic state propagations that the system **could** support. To create a state mesh that the system can operate on requires the user to provide a set of instance data, describing the actual model **objects** and **relationships** that exist between those objects.

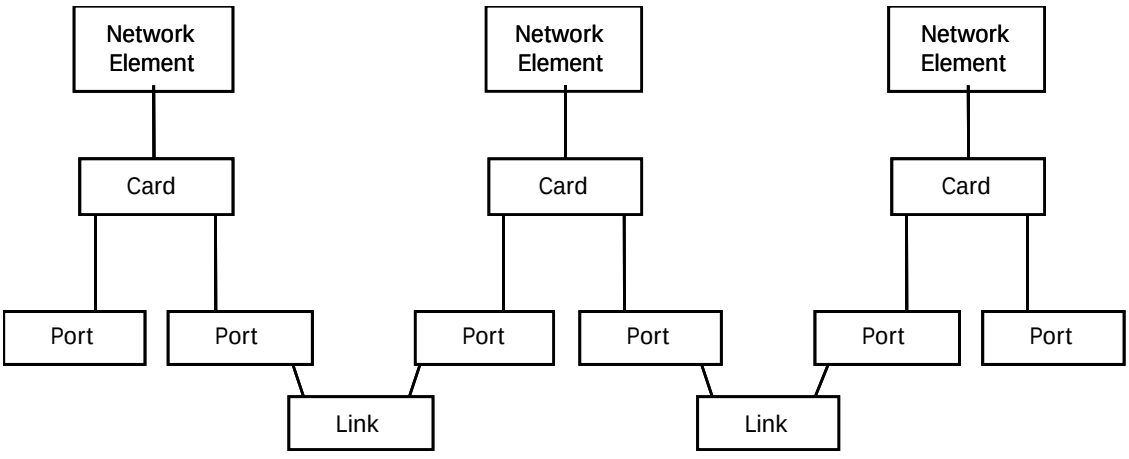
Normally, this instance data is stored in the UCA model database. Because the structure of the model database will vary with each type of user model (e.g. different classes and types and numbers of relationships), UCA automatically generates the table structures from the metamodel. UCA can also be used to easily load the instance data into the model database. More typically, a batch process would be used to regularly update the model database with the latest instance data e.g. through a CSV file import from an external network inventory database.

When UCA is started, its model builder uses the metamodel as a template of instructions to create the state mesh. Subsequently, each time the model database is updated, the model builder is automatically triggered (again using the metamodel as a managing template) and the state mesh is brought inline with the new data load. The entire process is illustrated below.



3.6 Example State Mesh

At this point it is useful to consider an example to understand how the various parts are constructed and what the resulting state mesh actually looks like. The following diagram illustrates the components of a simple communications network.

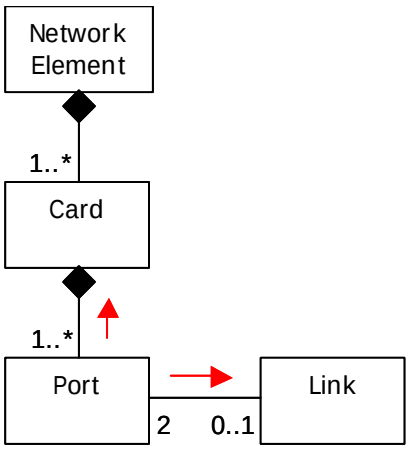


The example network operates in the following manner: Network Elements responsible for providing communications through the network have interface Cards with a number of communications Ports. Joining together Ports with Links creates a communications path through the network.
The first task is to construct the metamodel for this system. As described above, the simplest way to do this is to draw the equivalent UML class diagram. Before this can be completed however it is necessary

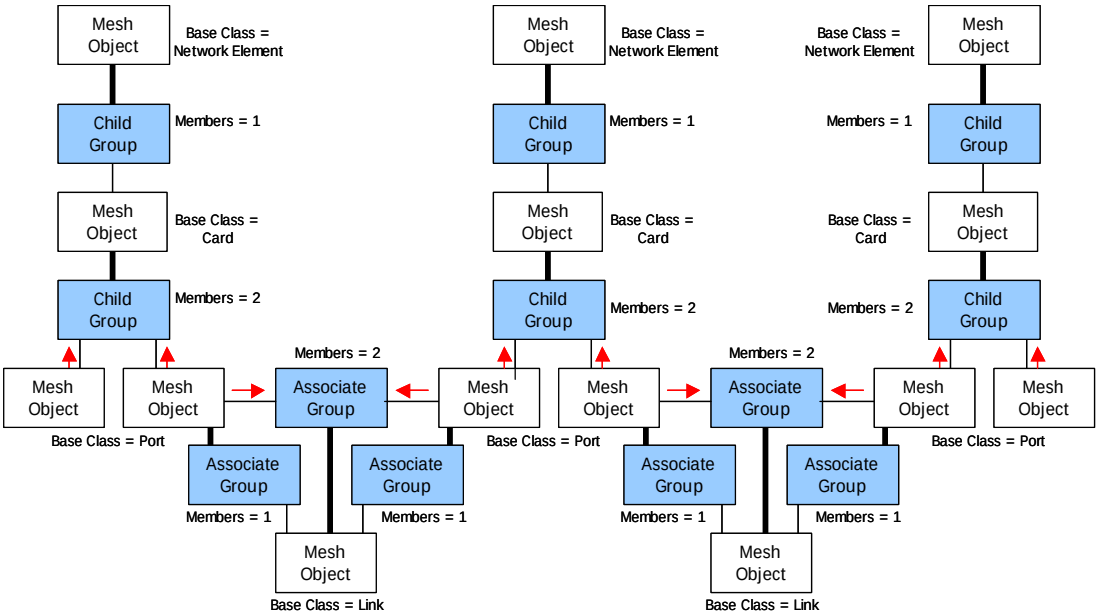
to consider what kind of automatic state propagations are required. To help decide this, the correlations that UCA is required to perform must be considered. For the purposes of this example, they are:

- Report a Card failure when all of its Ports have reported a hardware error.
- Report a Link failure when the Ports at both ends have lost the communications signal.

To detect the first condition, a Card will need to know the state of each of its child Ports. Therefore, the simplest choice is to automatically propagate the state of a Port to its parent Card. The second condition is similar in that a Link object will need to know the state of all the Ports that it is attached to. Again, the obvious choice is to automatically propagate the state of a Port to its associated Link.
The resulting UML class diagram with annotations (red arrows) to show the required automatic state propagations is as follows:



The arrows in the diagram are for illustration purposes only. In practice, a UML CASE tool requires the definition of stereotypes on affected relationships to add support for automatic state propagation. Following processing of the UML class model to create the metamodel and its combination with user supplied instance data, the state mesh would possess the internal structure shown below.



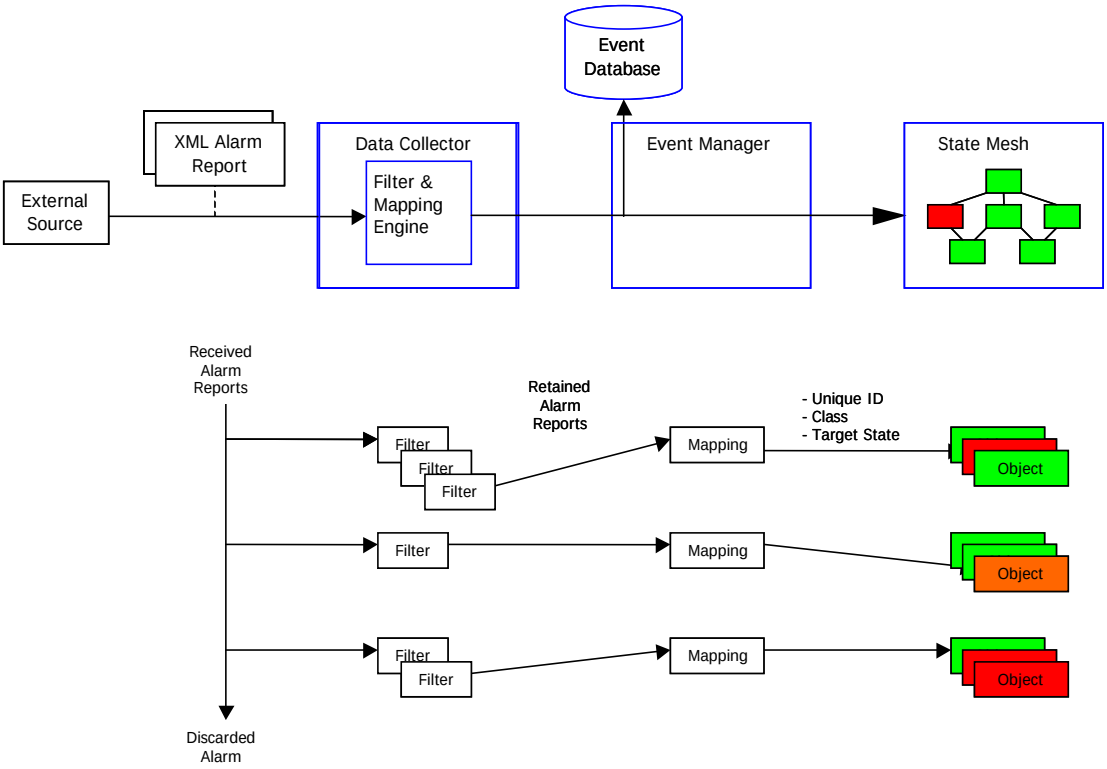
The model builder has added a number of 'helper' objects (child and associate groups) to the model to assist with the management of containment and associative relationships defined in the metamodel. These group objects serve to keep a list of child or associate mesh objects attached to a mesh object – the thick lines denote the mesh object to which the group belongs and the thin lines denote the mesh objects that they hold on behalf of that mesh object. Notice that each group object maintains a count of the mesh objects it is responsible for.

The red arrows denote the relationships defined in the metamodel for which automatic state propagation is defined. Notice that the model builder has configured the model such that automatic state propagation only exists between mesh objects that originate state change reports and the group(s) to which they belong, rather than to the mesh objects that own those groups. One of the most important features of group objects is that they are capable of maintaining a real-time state count of the mesh objects they contain i.e. total failed and degraded members. If automatic state propagation is enabled e.g. for Port mesh objects in the above diagram, then a group object's state counts will be automatically updated each time the state of one of the mesh objects it contains is updated.

Based on the relationships defined in the metamodel diagram, Port objects (represented by mesh objects with a base class = Port) are owned by Card objects (represented by mesh objects with a base class = Card) and they have appropriate child group objects to manage them. Also, Port objects are associated with Link objects (hence the associate group objects – one at each end of the associative relationship because it is potentially bi-directional). Because of this dual relationship, a state change of a Port mesh object will be simultaneously reported to both its parent's child group object and its associate's associate group object.

3.7 Data Collector and Event Manager

Mesh objects in the state mesh are state aware in that they can exist in one of three possible states - normal, degraded and failed, and can propagate this information to other objects if required. UCA is driven by events gathered from the monitored network and therefore needs a mechanism that allows them to modify the states of mesh objects in the state mesh. The process and information flow employed by UCA is shown below.



The first component in this mechanism is the UCA Data Collector. This is responsible for providing an external interface into which alarms from an external source are delivered. To accommodate wide variations in the type and content of alarms from different sources, UCA has a well-defined XML input format, derived from the CCITT ITU X.733 standard. Alarm reports delivered to UCA must conform to this format. UCA responds to 'alarm raise', 'alarm clear' and 'alarm termination' reports received from external sources.

Once the Data Collector receives alarm reports, a hierarchical set of filters (configured through the Scenario Manager) is applied in turn to fields within them. The filters are necessary to remove unused alarms – network management systems are sometimes not selective in the reports they deliver and unwanted reports consume valuable system resources for no benefit. Continuing with the example communications network model described above, that system's filters would be configured to retain only those alarm reports that signify the onset and recovery of a hardware failure or loss of communications signal on a Port.

Alarms are also subjected to a mapping (again, configured through the Scenario Manager). The following actions are performed during a mapping:

- a unique object identifier is extracted from one or more fields of the alarm
- a target mesh object is located in the state mesh with a specified base class and a name equal to the extracted unique identifier
- the alarm report is attached to the target mesh object (if it is a Raise report) or removed from the target object (if it is a Clear or Terminate report). Alarm reports attached to a mesh object are held in its current problems list.

Alarm reports that pass to the end of a filter chain are mapped according to the mapping definition(s) at the end of the chain and stored in the UCA event database by the Event Manager for future reference. Each alarm report is assigned a target state (normal, degraded or failed) defined in the mapping and each time an alarm report is attached to or removed from a mesh object in the state mesh, the system will re-evaluate the mesh object's overall state. This will be set to the highest state of all attached alarm reports or normal if none remain.

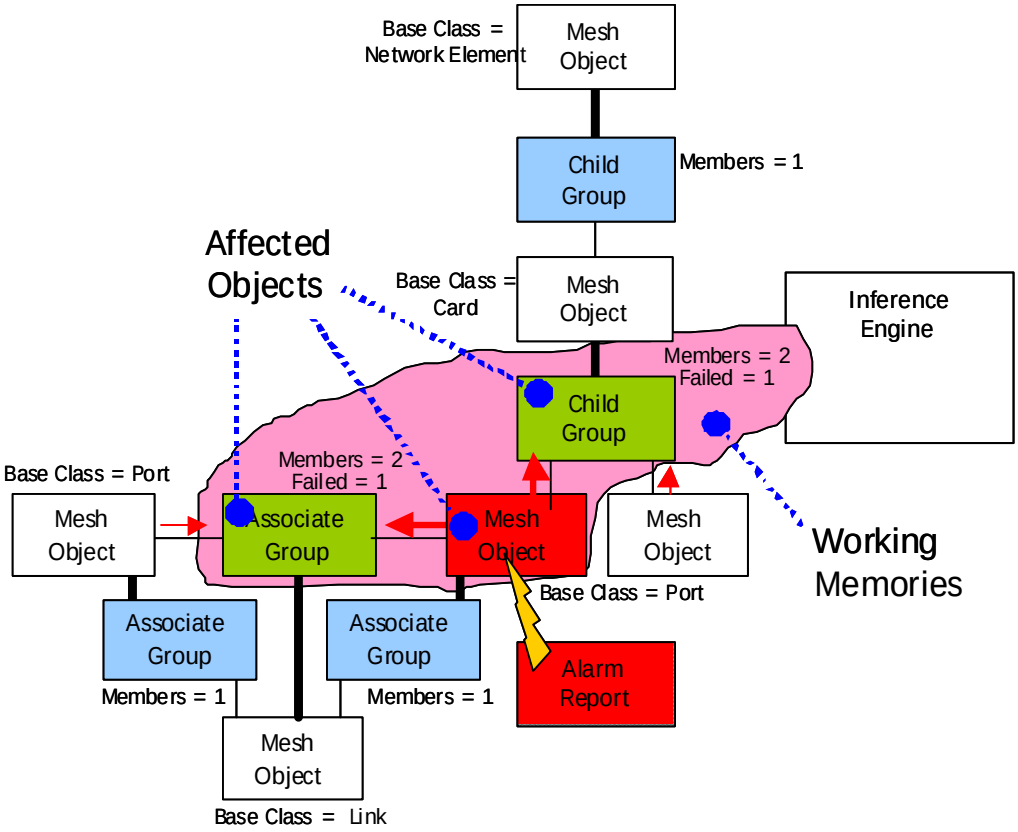
3.8 Affected Objects

Alarm reports that pass the system's filters and are then mapped to target mesh objects in the state mesh can result in one or more state-related changes. These include:

- Target mesh objects may change their overall state.
- If automatic state propagation is activated and the overall state of a target mesh object is changed, state counts maintained by any group objects containing that mesh object will be updated.

Mesh and group objects in the state mesh altered in either of these ways are termed affected objects and they have a special significance. UCA will insert target mesh objects whose state has changed from normal (as a result an alarm report being mapped onto them) into each of the working memories of the UCA inference engine (there may be one or more working memories defined). Similarly, group objects whose degraded or failed member counts have increased from zero (as a result of an alarm report being mapped onto a contained target mesh object and automatic state propagation taking place) will also be introduced into each of the working memories.
Alternatively, if a target mesh object or group object is already inserted in the working memories (as a result of a previous state change) then UCA will update its state or affected member counts respectively.

Finally, if a target mesh object is already inserted and all attached alarm reports are removed, then its state will return to normal and it will be updated in the working memories. Similarly, a group object whose affected member counts have all returned to zero will also be updated in the working memories. Note that under these conditions, neither of these object types is automatically removed at this stage from the working memories. This method of operation has been chosen specifically to allow the user an opportunity to build rules that depend on objects returning to the normal state.
Using the previous example of a simple communications network, the following diagram illustrates the process of creating affected objects and insertion into working memories when an alarm report is received.



Following a Port failure in the actual network, an alarm report received by the system is mapped onto an equivalent Port target mesh object in the state mesh. UCA uses the target state from the mapping to set the state of the Port target mesh object to failed, resulting in its automatic insertion into the inference engine's working memories. Automatic state propagation from the Port target mesh object to its containing child and associate groups has also incremented their failed member counts above zero, causing them also to be automatically inserted into the working memories. Note that mesh and group

objects that are inserted into the working memories remain part of the state mesh and continue to be attached to their unaffected counterparts by their existing relationships.
The UCA Mesh Viewer GUI allows a user to view and monitor the state mesh in real-time. It provides a comprehensive, navigable view of all target mesh objects in the currently loaded model and also maintains a dynamically updated list of mesh objects that are in non-normal states.

3.9 Inference Engine

The purpose of the inference engine is to provide an efficient and highly optimised decision-making tool that can be controlled by a set of user-defined rules to infer information about the condition of the monitored network. It achieves this by evaluating affected objects that have been inserted or updated in its working memories against the specific set of rules defined for each such working memory. Once a rule has been satisfied, the system will carry out one or more actions (chosen from a list of actions during rule configuration).
It is important to clearly understand the relationship between objects in the state mesh and affected objects in the working memories, as illustrated in the following diagram.

Objects that are part of the state mesh always remain so, regardless of their state. Affected objects represent a sub-set of objects in the state mesh that are in a non-normal condition and as a result have been temporarily inserted into the working memories, where they have in turn become visible to the rules controlling the inference engine. Objects that are part of the state mesh (and not affected objects) are normally invisible to the inference engine (because they are not inserted into the working memories). There is one exception to this rule however, which is that they are indirectly accessible to rules and their resulting actions where they can be reached by navigating the relationships between them and affected objects that are visible to the inference engine.
Rules are created with the Scenario Manager and comprise arbitrarily complex ‘when (rule is true) then (do action)’ constructs. The system takes care of translating these constructs into the low-level rules language that the inference engine understands and automatically deploys them into the specified working memory. A major benefit of this approach is that most users can create rules using familiar concepts and terminology e.g. “is there a card where 100% of the ports have failed”, without the need to understand the complicated language syntax and associated programming techniques normally associated with inference engines.
Rules created in this way may have general conditions to test for the existence or otherwise of affected objects in the working memory e.g. when (there is a not a Card) then (…). Alternatively, they may have a number of specific conditions that are compared with the attributes of affected objects e.g. when (there is a Port object with state Failed) then (...).
Rules may also be targeted, for example aimed at the existence of a particular affected mesh object in the working memory e.g. when (there is a Card whose name starts with “ABC”) then (…).
Alternatively, they may operate at the class level, in which case they will be applied equally to all affected mesh objects of the defined type (and / or subtype) that satisfy their conditions e.g. when (there is a Card of subtype SDH) then (…). Rules may also be defined to operate on affected group objects e.g. when (there is a group owned by a Card where 100% of its Port members have failed) then (…).
When all of the conditions attached to a rule are satisfied, they are placed on a list of rules waiting to be ‘fired’ or executed. The inference engine will remove and execute the next rule on the list, carrying out one or more actions associated with it. After each rule is fired, the remaining rules on the agenda are re-evaluated to see if they are still valid (any that have become invalid as a result of the previous rule execution are removed without being processed). An important characteristic of inference engines is that once a rule has fired for a particular set of conditions, it will not do so again until a change has happened and those conditions are again satisfied. This prevents a rule from firing continuously when a particular set of conditions remains true.
Rules may be assigned a priority that can be used to control the order in which satisfied rules are removed from the list and executed. For example, UCA is used with a set of low priority ‘maintenance’ rules whose actions are responsible for removing affected objects (mesh and group objects) from the working memory when they return to their normal state. By setting the priority of these rules at a low level, the user is provided with the opportunity to define higher priority rules that detect normal state objects and carry out some other action before they are removed from the working memory.
UCA provides a comprehensive range of pre-defined actions, including the ability to:

- Create, acknowledge, demote, terminate and clear alarms in the originating network management system, depending on its ability to support such operations.
- Modify the state of mesh objects in the state mesh.
- Create, modify and delete ‘notifications’ attached to mesh objects, designed to report significant events to users via the Notification Dialog (see Mesh Viewer GUI details).
- Associate contributory alarm reports responsible for the creation of affected objects to notifications.
- Identify mesh objects in the state mesh that may be affected by a problem in another part of the model and associate their sympathetic alarm reports to a notification.
- Execute user-defined scripts on both the local and remote platforms and to incorporate the results into further correlation scenarios.

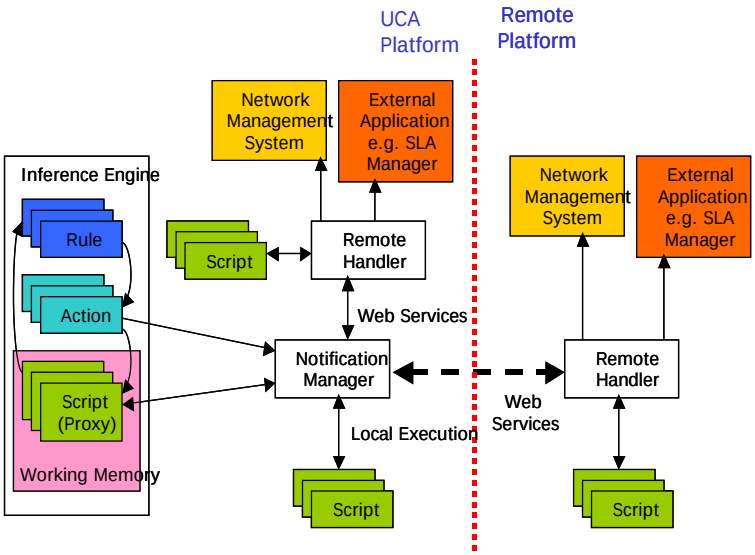
In addition, it is possible for a user to define additional actions to carry out special tasks. These require the creation of additional action functions written in Java using the UCA API, and to add action function details to the UCA action properties files to enable them to be accessed from the Scenario Manager.
Certain actions, including those that initiate notifications and allow user-defined scripts to be executed, create corresponding dynamic objects in specific working memories. These dynamic objects (notification and script (proxy) objects) are visible to rules defined in those working memories and allow users to construct correlations that depend on their existence or attributes.
The properties of notification objects are such that they may exist in a maximum of two working memories at any time – typically they are created in a source working memory (context) and may be made visible in a destination working memory (context). This powerful concept allows for ‘communication’ of the results of a correlation in a source context (with a certain set of rules) to drive another correlation (with a different set of rules) in a destination context. Updates to a notification object are obviously made visible to the rules in both the source and destination working memories.
A script (proxy) object is created by an action when the corresponding script is first executed and (depending on configuration) may persist past the execution lifetime of the script itself, recording the status and results of its execution for use in later stages of a correlation. Scripts executed by actions are launched in separate threads to avoid contention and blocking and only exist in the source working memory.
The UCA System Manager GUI also provides a Fired Rules dialog for users to monitor the execution of rules and their associated actions (this information is also stored in the UCA notification database and is available for subsequent analysis).
The UCA Mesh Viewer also provides a Notification dialog to allow users to examine the set of notifications associated with an object (again, the information contained in each Notification is stored in the notification database and is available for analysis).

3.10 Notification Manager and Remote Handler

It is the responsibility of the UCA Notification Manager to handle any interactions between UCA and external systems, including:

- Manipulation of alarm reports in the external network management system via the Remote Handler.
- Execution of scripts in separate threads on the local platform.
- Execution of scripts in separate threads on local and remote platforms via the Remote Handler.
- Updating corresponding script dynamic objects with execution status, exit codes and results from locally and remotely executed scripts.
- Managing external system interactions on behalf of user-defined actions e.g. starting/stopping SLA monitoring for service impact correlations.

Operation of the Notification Manager and Remote Handler are illustrated in the following diagram.



UCA provides a simple, flexible API to manage external interactions from within the rules/action context.
The Remote Handler is normally executed as a separate process on system restart. It provides the ability to interface to external systems and execute scripts on both local and remote platforms, returning results and output information back to UCA. It utilises web services to minimise communications problems associated with firewalls between the UCA and remote system and again requires straightforward integration with remote applications.
UCA may also directly execute scripts on the local platform without the need for a Remote Handler.

Chapter 4 The UCA Home Page and System Manager

4.1 Starting the Tomcat 'Minimal Web Server'

UCA uses Tomcat for:

- serving static web pages
- serving dynamic web pages, using JSP
- handling web services requests from the client, executing the appropriate Java code and sending the response, as appropriate. i.e. using Tomcat as a 'servlet container'.
- handling role-based authentication to web pages and UCA applications

In order for UCA to start up, Tomcat must be running. In addition, when UCA is shut down, Tomcat must be forced to release all of its resources. This could be done manually, but UCA provides a 'minimal web server' called **tomcatserver** to automatically control this.
After UCA has been installed and configured, **tomcatserver** must to be started. This only needs to be done once and under normal circumstances **tomcatserver** should never need to be stopped.
tomcatserver is started as follows:

For HP-UX

```
cd $UCA_HOME/bin
./tomcatserver.sh
```

tomcatserver actually listens on a port (defined by the **tomcatserver.port** property in the **uca.properties** file) for web services requests - accepting 'start' and 'stop' requests that have the effect of starting and stopping Tomcat itself. When **tomcatserver** is first started, it automatically starts Tomcat.
When UCA is shutdown from the System Manager GUI (see the following section), a 'stop' followed by a 'start' request is automatically sent to **tomcatserver –** this has the effect of stopping and re-starting Tomcat.
As mentioned above, once **tomcatserver** is started, normally nothing more needs to be done by a user other than to interact with the UCA GUIs. However, should Tomcat need to be stopped or started manually, this can be done as follows:

For HP-UX

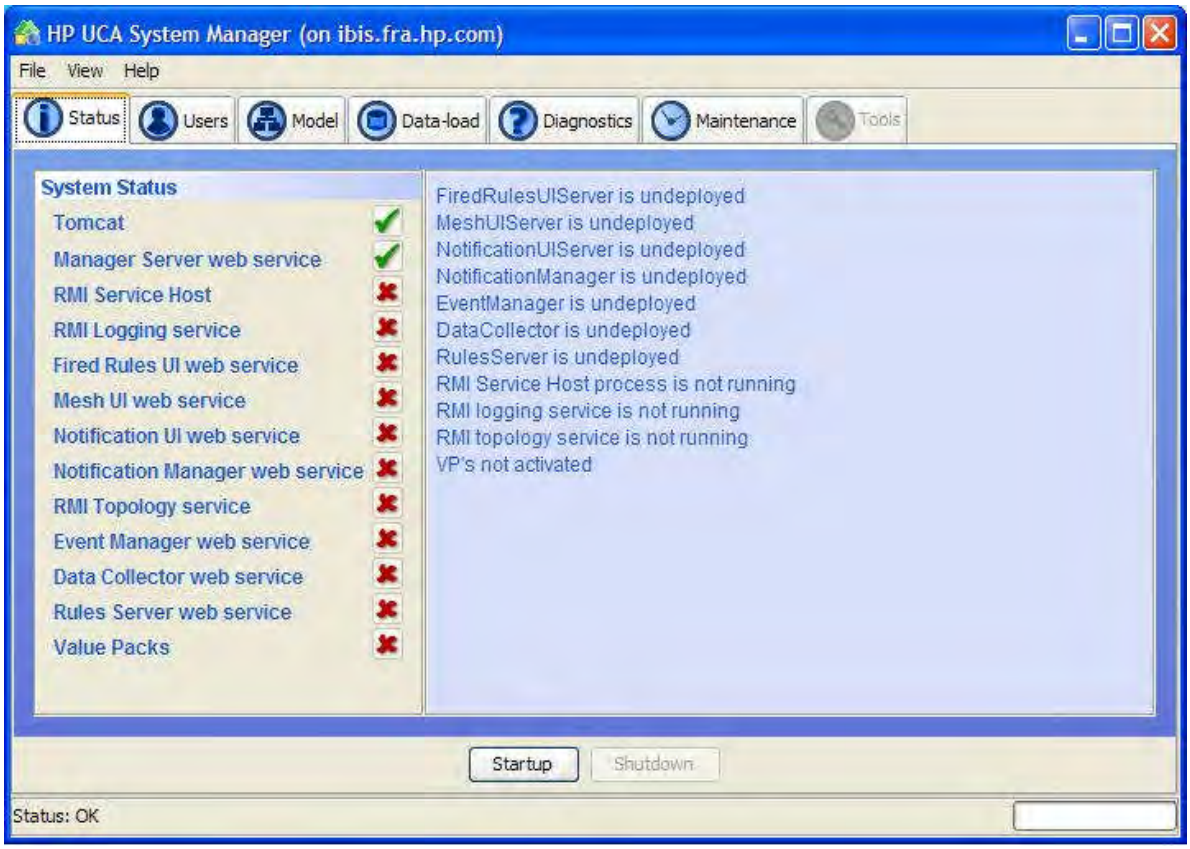
```
cd $UCA_HOME/bin
./tomcat.sh stop or ./tomcat.sh start
```

4.2 Starting the System Manager

Assuming the system is installed and properly configured (see the ***HP UCA Installation and Configuration Guide*** for details), and **tomcatserver** has been started as described above, entering the following URL in a web browser will result in the UCA Home Page being shown, as follows:
http://***hostname***:18080/uca
where ***hostname*** is the DNS name or IP address of the server machine on which UCA is installed. Note that it is possible to configure a port other than 18080 for use by UCA – see the ***HP UCA Installation and Configuration Guide*** for details.

Figure 4 - The UCA Home Page

Clicking on the **UCA Manager** button will invoke a username / password dialog. When UCA is first installed, a username and password of 'system' and 'system' is pre-configured with 'manager' role. Entering this username and password will cause the UCA Manager GUI to be displayed with the 'Status' tab selected, as follows



The status tab shows details on the left hand side of all the major UCA software components and their current status. The green tick indicates that the component is running and the red cross indicates that the component is not started. In normal circumstances, Tomcat and the 'Manager Server Web Service' should always be running. Note that during start-up, the web applications will temporarily be shown with a yellow question mark symbol; this indicates that Tomcat has deployed the service but it has yet to be initiated.
The tabs across the top of the window provide access to a number of different system management features. Certain operations, such as defining and loading the model, can only be performed when the system is not started, whilst others can only be done when the system is running. For this reason, tabs are enabled or disabled depending on the running state of the system. The table below summarises the state of the tabs depending on the state of the system.

Tab	System not started	System started
Status	Enabled	Enabled
Users	Enabled	Enabled
Model	Enabled	disabled
Data-load	Enabled	disabled
Diagnostics	enabled (see 1.)	Enabled
Maintenance	enabled (see 2.)	Enabled
Tools	Disabled	Enabled

1. viewing and enabling/disabling pre/post filter event logging is disabled
2. mesh update and archive update settings disabled

4.3 Adding, Modifying and Deleting Users

New users may be added or existing users modified or deleted from the 'Users' tab.
To add a new user:

- Enter a username and password (the password must be at least 6 characters long)
- Select the appropriate role(s)
- Select **New**

The roles are uses as follows:

- manager – a user must have manager role to invoke the System Manager GUI
- administrator - a user must have administrator role to invoke the Scenario Manager GUI
- operator - a user must have operator role to invoke the Mesh Viewer GUI
- read-only – with read-only role, a user cannot deploy scenarios, filters, mappings or rules from the Scenario Manager GUI.
- tester – a user with tester role may invoke the Scenario Manager and Mesh Viewer GUIs. In addition, from the Mesh Viewer GUI, the user may inject a set of alarms from an external file (using the 'Inject alarms from file' File pull-down menu) or inject a single user-specified alarm by right-clicking an item in the Instances tree and selecting the 'create alarm' popup menu item.

To update an existing user:

- Select the username from the list in the left panel
- Update the username, password or roles as appropriate. ***Note that, for security reasons, the existing (or a new) password must be re-entered for the update to successfully apply.***
- Select **Update**

  Note that a user currently logged on to the System Manager cannot remove manager role from his/her own details. To do this, you must exit the System Manager GUI and restart it, logging on as a different user (with manager role privilege), then select the original user and remove manager role.

To delete an existing user:

- Select the username from the list in the left panel
- Select **Delete**

  Note that a user currently logged on to the System Manager cannot delete his/her own entry. To do this, you must exit the System Manager GUI and restart it, logging on as a different user (with manager role privilege), then select the original user and delete it.

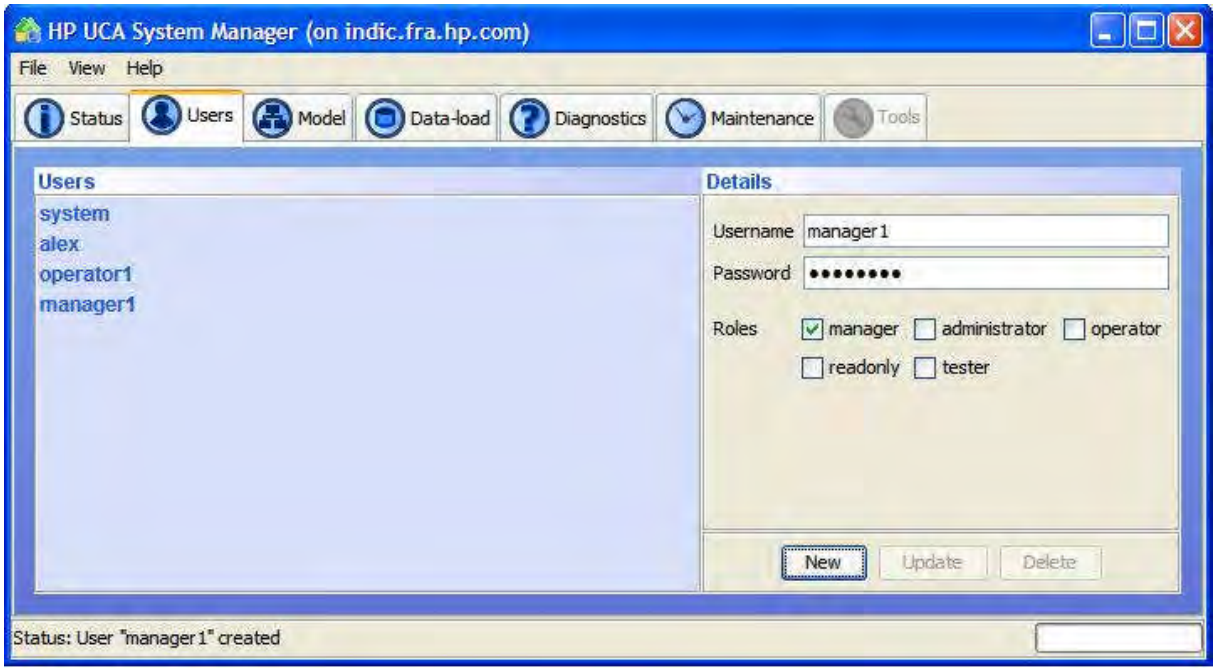


Figure 5 - The System Manager Users Tab

4.4 Starting UCA

From the Status tab of the Scenario Manager, click on the **Startup** button. Following a confirmation prompt, each of the sub-systems will then be started and the icons next to each sub-system will change to reflect their

status. Progress is described in the text area on the right side of the window and any error messages will be displayed here and / or in the status bar area at the bottom of the window. UCA is fully started up when a green tick appears against each sub-system, as shown below.

Figure 6 - The Status tab showing the system started

4.5 Stopping UCA

Before the system is stopped, all Scenario Manager and Mesh Viewer GUIs in use by all users should be closed.

From the Status tab of the Scenario Manager, click on the **Shutdown** button. Following a confirmation prompt, each of the sub-systems will then be shut down and the icons next to each sub-system will change to reflect their status. Progress is described in the text area on the right hand side of the window and any error messages will be displayed here and / or in the status bar area at the bottom of the window. UCA is fully shut down when a green tick appears against 'Tomcat' and 'Manager Server web service' and a red cross appears against all other sub-components. Note that during the shut-down process, Tomcat is automatically re-started (see Starting the Tomcat 'Minimal Web Server' section above) – this may take 15 to 20 seconds depending on the capability of the server. ***Shutting down the system will cause any corresponding session on a user's web browser to end. This means that if a user has any UCA web pages displayed and the system is shut down, then those pages will become 'stale' and the page must be re-loaded after UCA has been re-started***.

It is recommended that the web browser is closed after UCA has been shut-down and re-opened after UCA is re-started – **this is important prior to re-starting the Scenario Manager GUI or Mesh Viewer GUI after a system re-start**.

4.6 Configuring the Metamodel

The Model tab of the Scenario Manager provides all the functions necessary for a user to configure the metamodel structure used by UCA to fulfil all the needs of the set of required scenarios. Chapter 5 provides details of how this metamodel is constructed and used within UCA.

4.7 Loading Data into the Model

The Data-load tab of the Scenario Manager provides functions that may be used to load data from CSV text files into the UCA model database. Chapter 6 provides details of how this is performed.

4.8 Diagnostics

The UCA diagnostic facilities are accessed from the Diagnostics tab of the System Manager GUI.

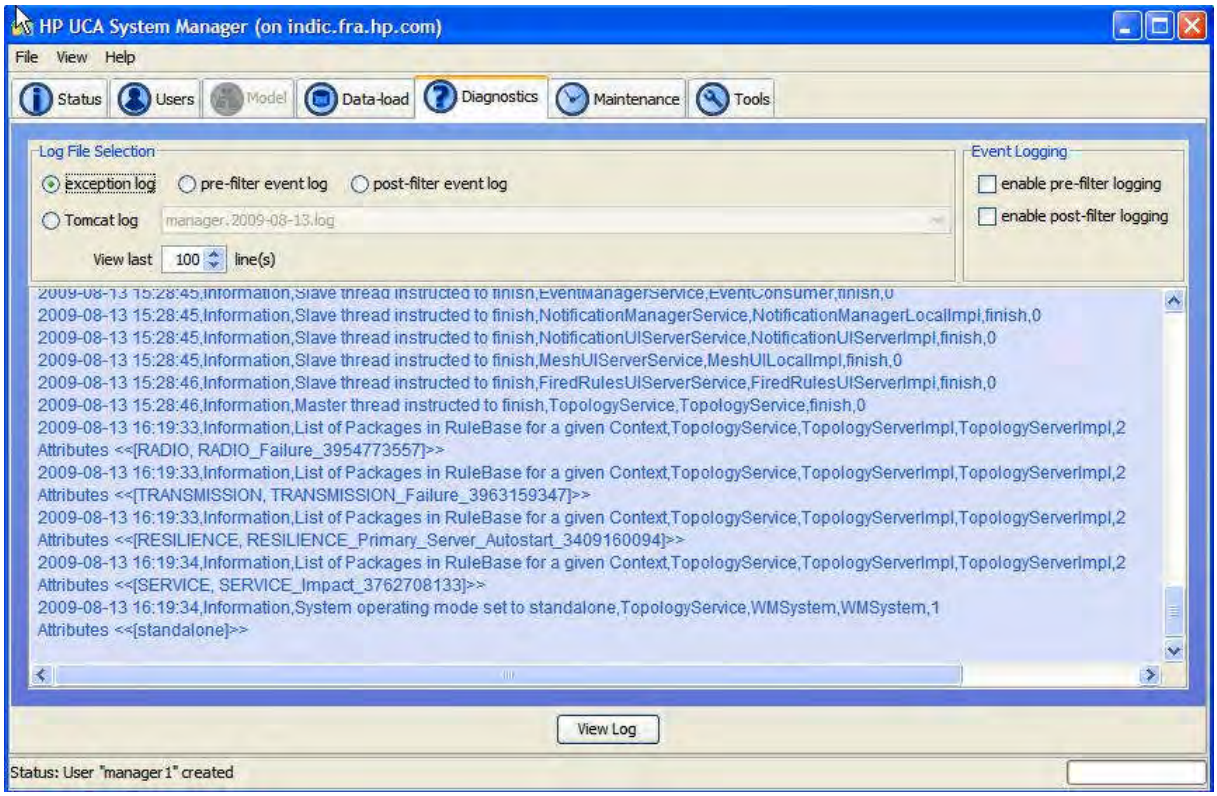


Figure 7 - The System Manager – Diagnostics tab

Within this tab, the following information can be displayed:

- The contents of the centralised UCA exception log.
- The contents of the non-empty Tomcat logs.
- The contents of the pre-filter event log.
- The contents of the post-filter event log.

Since these log files can be large, the number of lines to display from these logs may be selected.

To view the exception log details:

- Select the ‘exception log’ radio button
- Select the desired number of lines to display in the ‘View last’ spinner (between 1 and 500)
- Click the **View Log** button.

To view a non-empty Tomcat log:

- Select the **Tomcat log** radio button
- Select the desired Tomcat log file from the drop-down list of filenames.
- Select the desired number of lines to display in the ‘View last’ spinner (between 1 and 500)

- Click the **View Log** button.

The pre-filter event log maintains a list of all incoming events **before** they have passed through the Filters, in the same format as described in section 10.3.2. This log file is a useful source of events to replay into the system (for example using the UCA Event Injector tool).

To view the pre-filter event log details:

- Select the **pre-filter event log** radio button (the system must have been started in order to view the pre-filter event log)
- Select the desired number of lines to display in the 'View last' spinner (between 1 and 500)
- Click the **View Log** button.

The post-filter event log maintains a list of all incoming events **after** they have passed through the Filters. The details in this log are in a similar format to section 10.3.2, but with the following tags added:

- <uniqueReference>
- <baseClass>
- <status>
- <pathToMapping>

The uniqueReference tag contains the value of the uniqueReference after mapping has been performed (see Section **8.3**).
The baseClass tag contains the value of the baseClass after mapping has been performed (see Section **8.3**).
The status tag contains the value of the status after mapping has been performed (see Section **8.3**).
The pathToMapping tag contains a separated list of numbers, each enclosed in square brackets. The numbers represent the internal unique Ids for each filter in the Scenario Builder Tree (see section 7.1.1). The tag value represents the path that the event took through the hierarchy of filters. It is useful to analyse the pathToMapping values to optimise the position of the filters in the Scenario Builder Tree. An example log entry in the post-filter event log is as follows:

```
<Event>
   <uniqueReference>10001</uniqueReference>
   <baseClass>Site</baseClass>
   <status>failed</status>
   <pathToMapping>[-1][3074555833][-1][3074520804][3074519544]</pathToMapping>
   <additionalText>Site Power Failure</additionalText>
   <alarmType>EquipmentAlarm</alarmType>
   <dataType>X.733</dataType>
   <eventId>1003</eventId>
   <eventRank>original</eventRank>
   <moClass>Site</moClass>
   <moInstance>10001</moInstance>
   <originatingTime>2005-06-10 12:16:32</originatingTime>
   <probableCause>PowerProblem</probableCause>
   <severity>critical</severity>
   <systemClass>sidonis_nms</systemClass>
   <systemInstance>V5</systemInstance>
</Event>
```

To view the post-filter event log details:

- Select the **post-filter event log** radio button (the system must have been started in order to view the post-filter event log)
- Select the desired number of lines to display in the 'View last' spinner (between 1 and 500)
- Click the **View Log** button.

To enable or disable pre-filter event logging:

- Ensure the **enable pre-filter logging** checkbox is ticked / un-ticked (the system must have been started in order to enable or disable the pre-filter event log).

To enable or disable post-filter event logging:

- Ensure the **enable post-filter logging** checkbox is ticked / un-ticked (the system must have been started in order to enable or disable the post-filter event log

4.9 Maintenance

The UCA maintenance facilities are accessed from the Maintenance tab of the System Manager GUI.

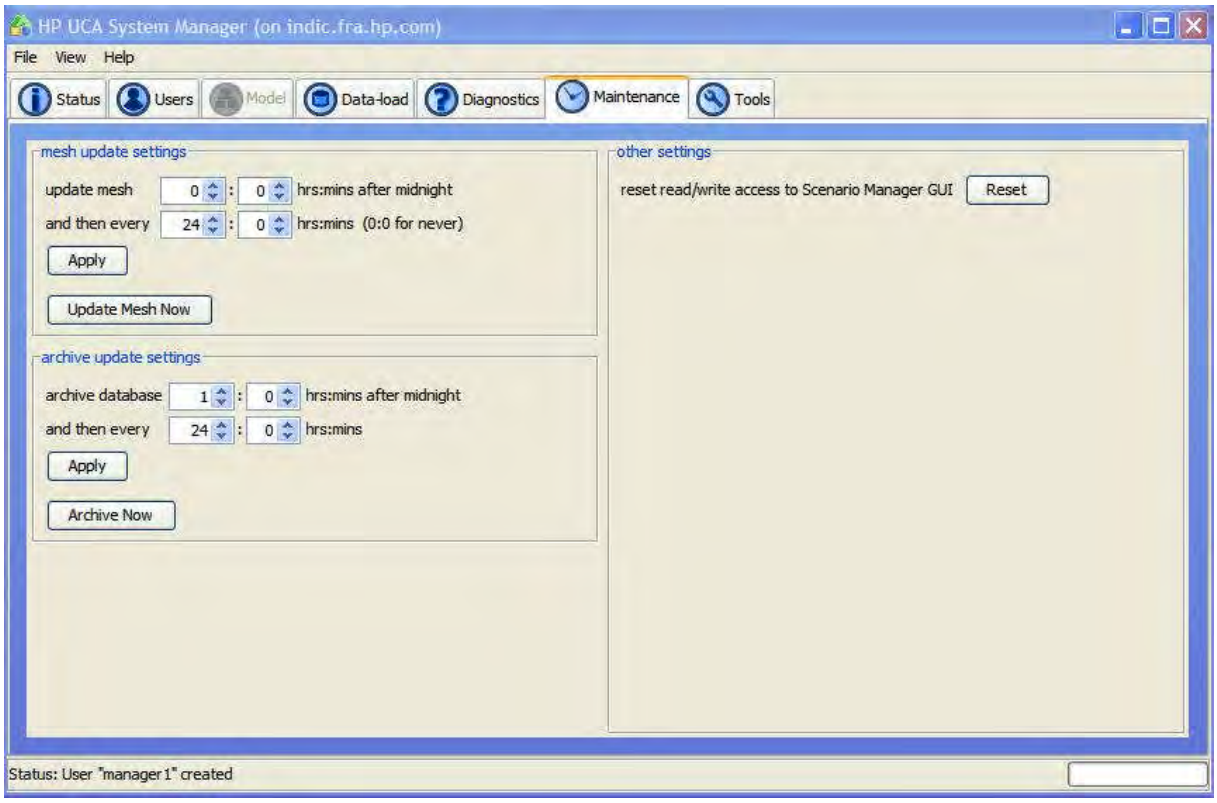


Figure 8 - The System Manager – Maintenance tab

Within this tab, the following actions can be performed:

- Configure the automatic Mesh update settings
- Manually apply a Mesh update
- Configure the automatic notification and event database archive settings
- Manually apply a notification and event database archive
- Reset read/write access to the Scenario Manager

Any model data that is inserted, deleted or modified in the UCA model database may be automatically propagated into the in-memory state mesh according to a configurable schedule (see sections 3.5 and 6.2.2 for further details).

To configure the automatic model update settings:

- In the Mesh Update Settings area, use the '**update mesh'** hours and minutes spinners to set the number of hours and minutes after midnight that you wish automatic Mesh updating to start.
- In the Mesh Update Settings area, use the '**and thereafter every'** spinner to set the interval, in hours and minutes, at which automatic Mesh updating is to be repeated.
- Click the **Apply** button in the mesh update settings area to apply the settings.

The default settings are to start at midnight and repeat once every 24 hours.

To manually force the in-memory state mesh to immediately update according to any recent model database changes:

- In the Mesh Update Settings area, click the **Update Mesh Now** button

UCA maintains many different types of data in its event and notification database tables. Without adequate management, these tables will grow bigger over time and will eventually reach available capacity. UCA provides the facility to intelligently archive this data (i.e. redundant data that is no longer needed by any outstanding correlation) and free up event and / or notification database space. This process occurs in two stages:

- Event processing is temporarily suspended and the event and notification databases are analysed to identify redundant data.
- Event processing is resumed and the previously identified redundant data is archived as a low priority background task.

Archiving may be configured to run on a scheduled basis. A user may also manually force an immediate archive of data.

The data that is archived comprises:

- Events received from external sources.
- Notifications
- Fired rule actions that have been configured to be logged in the notification database.
- Contributory Event Lists
- Affected Object Lists
- Sympathetic Event Lists

The data is archived as separate, time-stamped CSV files in the 'archives' directory under the UCA installation directory. The format of these files is such that they can be easily re-imported into another UCA database instance using standard database tools.

To configure the automatic archive update settings:

- In the Archive Update Settings area, use the **archive database** spinner to set the number of hours and minutes after midnight that you wish automatic archiving to start.
- In the Archive Update Settings area, use the **and thereafter every** spinner to set the interval, in hours and minutes, at which automatic archiving is to be repeated.
- Click the **Apply** button in the Archive Update Settings area to apply the settings.

The default settings are to start at 1.00 a.m. and repeat once every 24 hours.

To manually force the archiving of the event and notification databases to happen immediately:

- In the Archive Update Settings area, click the '**Archive Now'** button

Only one user at a time is allowed full read-write access to the Scenario Manager GUI. This is to stop simultaneous deployments of Scenarios, Filters, Mappings and Rules from interfering with each other. The UCA manager database maintains details of who the current read-write user is. Once a user is granted read-write access, no other users can use the GUI to deploy data until the user with the 'read-write' lock has exited the GUI. Should a failure ever occur at a client machine running the Scenario Manager GUI, it is conceivable that this 'lock' could be left in the 'granted' state in the manager database. Should this ever occur, the lock status may be cleared so that a user may again be granted read-write access via the Scenario Manager GUI.

To reset read/write access to the Scenario Manager:

- In the Other Settings area, click the **Reset** button next to **reset read/write access to Scenario Manager GUI**.

4.10 Tools

UCA provides a number of useful facilities and tools, accessed from the Tools tab of the System Manager GUI, to assist during the rules development stage.

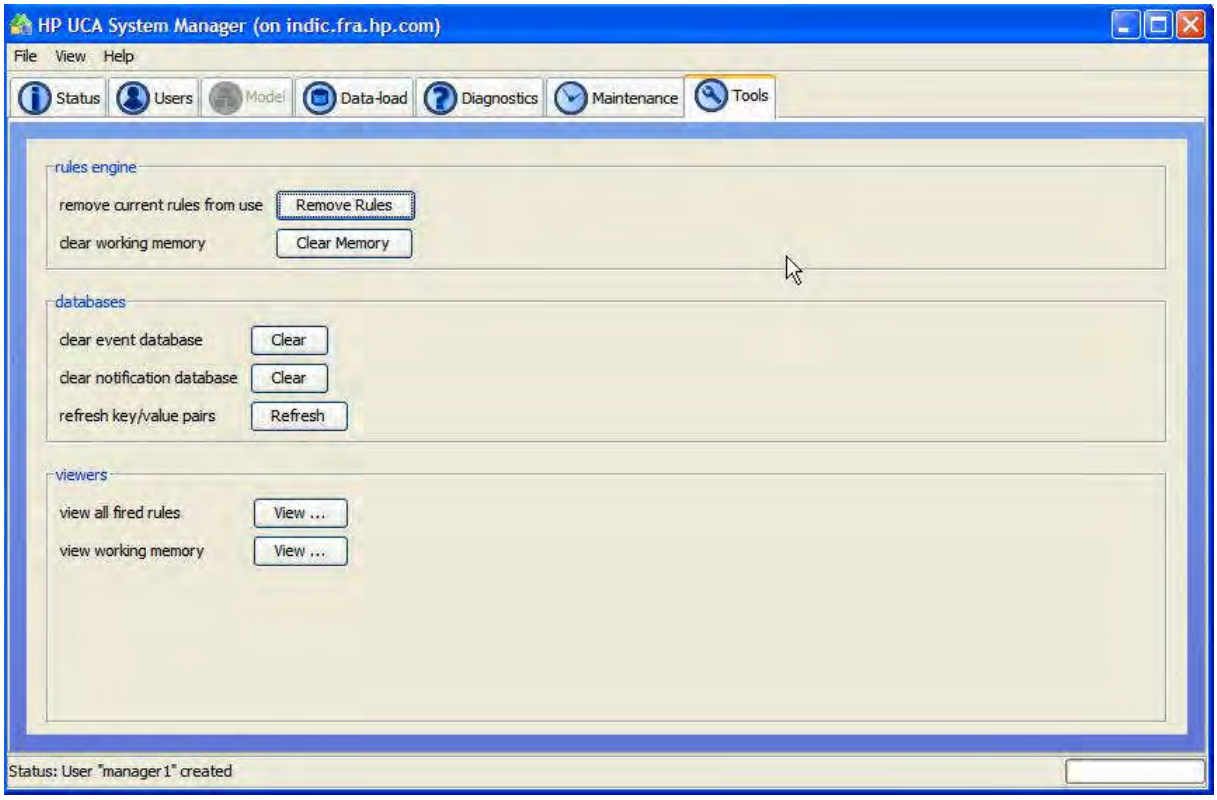


Figure 9 - The System Manager – Tools tab

Within this tab, the following actions can be performed:

- Remove all currently active rules from the inference engine.
- Clear the contents of all working memories monitored by the inference engine.
- Clear the event database.
- Clear the notification database.
- Refresh the dynamic property values (key / value pairs). This causes the UCA rules engine to re-scan the key / value properties held in the 'mg_properties' database table.
- View details of all the fired rule actions that have been logged in the UCA notification database.
- Graphically view the working memory contents.

During the testing stage of rule development, it may be necessary to remove all the currently active rules from the inference engine in case they are behaving in an unexpected manner. In this sense, this facility acts as a 'panic' button to immediately stop and remove all rules.

There is also a facility to clear all objects from all working memories within the inference engine. This essentially 'resets' the memories and is useful during the development and test stage, e.g. before starting a particular test run of sample alarms. During the reset process, mesh and group objects are returned to their normal states and all attached alarm reports are removed. All primary & marker notifications and script (proxies) are removed and destroyed.

To remove all currently active rules from the inference engine:

- In the Rules Engine area, click the **Remove Rules** button.

After use, it is recommended that the system be shut down and re-started.

To clear the contents of all the working memories:

- In the Rules Engine area, click the **Clear Memory** button.

UCA also provides the facility to clear the contents of the event and notification databases. Again, this is useful during the development and test stage, e.g. before starting a particular test run to de-clutter the system of any existing events or notifications.

To clear the event database:

- In the Databases area, click the **Clear** button next to ‘clear event database’.

To clear the notification database:

- In the Databases area, click the **Clear** button next to ‘clear notification database’.

UCA supports the use of ‘dynamic properties’. A dynamic property is a key / value pair set up in the UCA ‘mg_properties’ database table. These key / value pairs are accessible to rules. When the value of a dynamic property is changed in the database, any rules using that dynamic property will not be aware of the change in value. To make the rules aware of any changes to the dynamic properties:

- In the Databases area, click the **Refresh** button next to ‘refresh key/value pairs’.

The UCA System Manager provides two graphical tools that are very useful during rule development and testing. These are:

- The ‘Fired Rules’ viewer.
- The ‘Working Memory Viewer’

The Fired Rules Viewer

This viewer is used to view all the details of the rules that have fired (where database logging has been selected in the associated actions), together with details of any contributory events associated with the fired rules and the actions that have been carried out. All columns are re-sizable and movable and their headers may be clicked on to toggle the sort order.

To view the fired rules details:

- In the Viewers area, click the **View …** button next to **view all fired rules**.

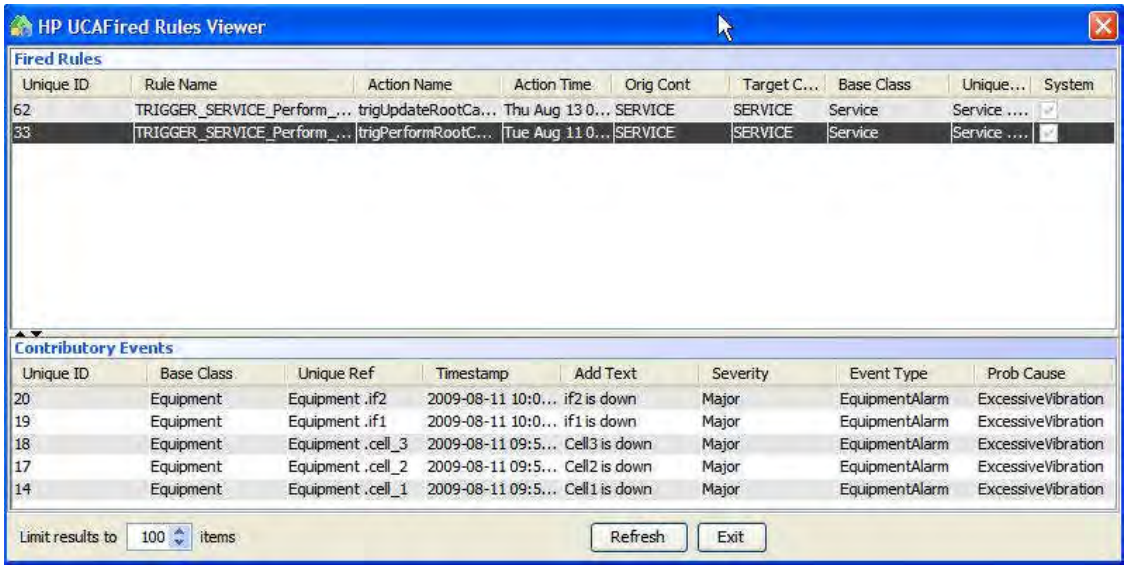


Figure 10 - The Fired Rules Viewer

The top table of the Fired Rules Viewer lists details of each fired rule. The details provided are:

- The Unique Id of the fired rule.
- The textual name of the rule. Rules fired from trigger conditions start with “TRIGGER_” and rules fired from teardown conditions start with “TEARDOWN_”.
- The mnemonic or short-hand name of the fired action.
- The time the action was fired.
- The originating and target contexts associated with the action.
- The trigger or teardown object’s base class.
- The trigger or teardown object’s unique reference.

When a row in the fired rules table is selected, details of the associated contributory events are shown on the bottom table. Note also that a single rule firing may result in more than one row in the fired rules table i.e. there is a row in the table for each logged action rather than each fired rule.

To refresh the view of fired rules details:

- Click the **Refresh** button.

To number of fired rules may be very large. To limit this in the viewer, the maximum number of most recent fired rules details may be set. To set the maximum number to view:

- Select the required number (minimum 1, maximum 200, default 100) from the 'Limit results to' spinner. This will take effect after the 'Refresh' button is clicked.

The Working Memory Viewer

This viewer is used to view all the details of the objects within the inference engine's working memories. The table columns are re-sizable and movable and their headers may be clicked on to toggle the sort order.

To view the working memory details:

- In the Viewers area, click the 'View …' button next to 'view working memory'.

To 'Contexts and Object Tree' shows, for each named context i.e. working memory, the different object types that may be inserted. These object types are displayed as nodes under a parent branch, where the parent branch represents the context name. The object types are:

- notifications
- mesh objects
- child groups
- associate groups
- script objects
- time objects
- system objects
- system key / value pairs

Contexts and Objects Tree

Objects Summary Table

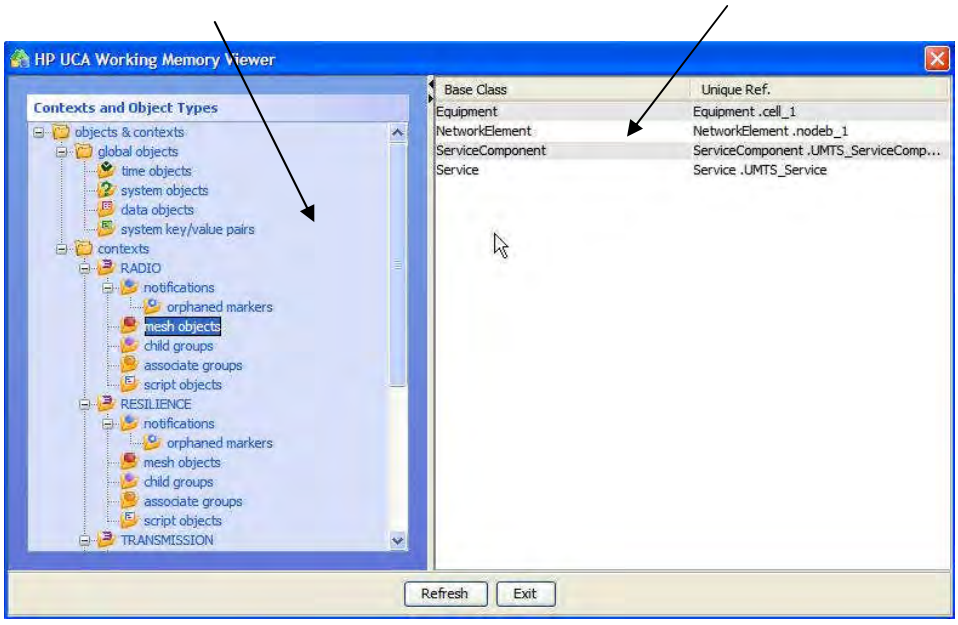


Figure 11 - The Working Memory Viewer

When an object type node is selected in the tree, the 'Objects Summary Tree' will display summary information for all objects of that type for the associated working memory. The summary details vary depending on the object type selected.

To view details of any items listed in the 'Objects Summary Tree',

- double-click the associated row in the 'Objects Summary Tree', or

- right-click the associated row in the ‘Objects Summary Tree’ and select **view details …** from the pop-up menu.

When an object type of ‘notifications’ has been selected in the ‘Contexts and Object Tree’, right-clicking an object in the associated ‘Objects Summary Tree’ will show an additional pop-up menu item – ‘show marker notifications’. The effect of this is to replace the contents of the ‘Objects Summary Tree’ with a summary of all the marker notifications associated with the notification that had been selected. When the marker notifications are displayed, right-clicking one will display a pop-up menu, similar to normal notifications, but with ‘back to parent notification’ instead of ‘show marker notifications’. Selecting ‘back to parent notification’ will return to the display of normal notifications, as previously displayed.

To refresh the view of fired working memory details:

- Click the **Refresh** button.

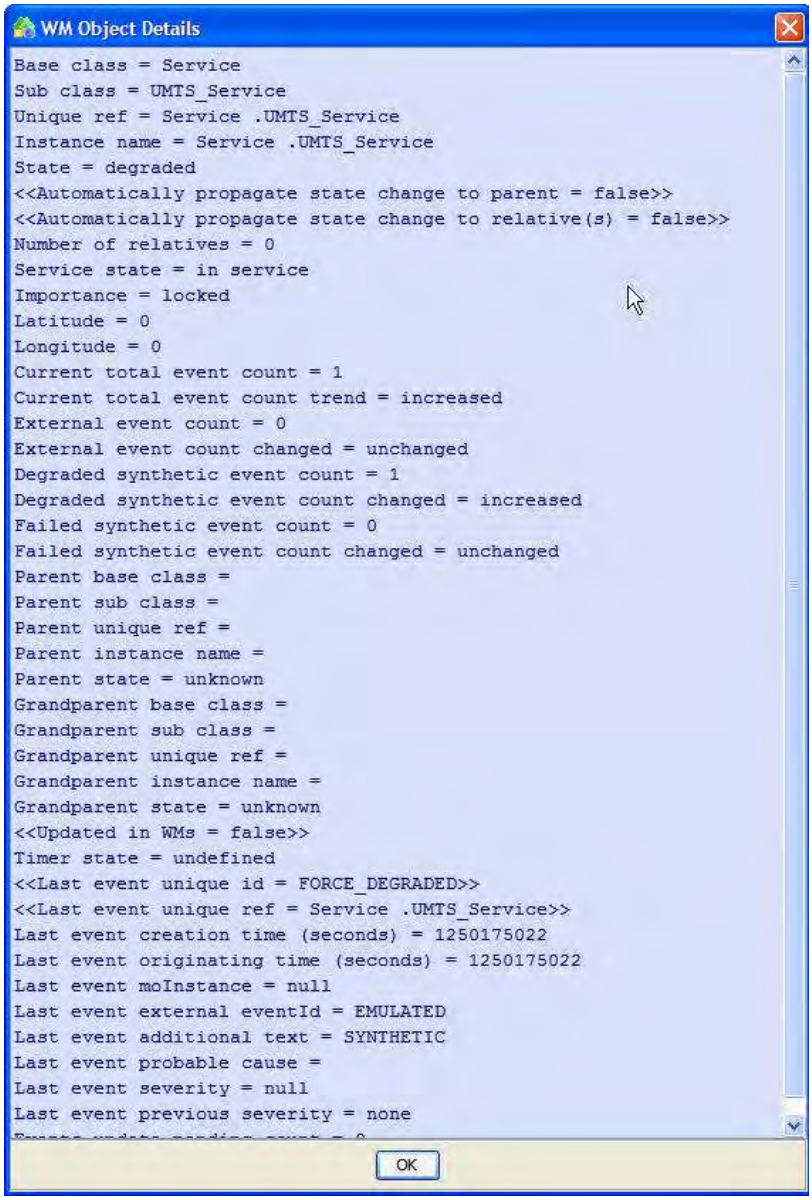


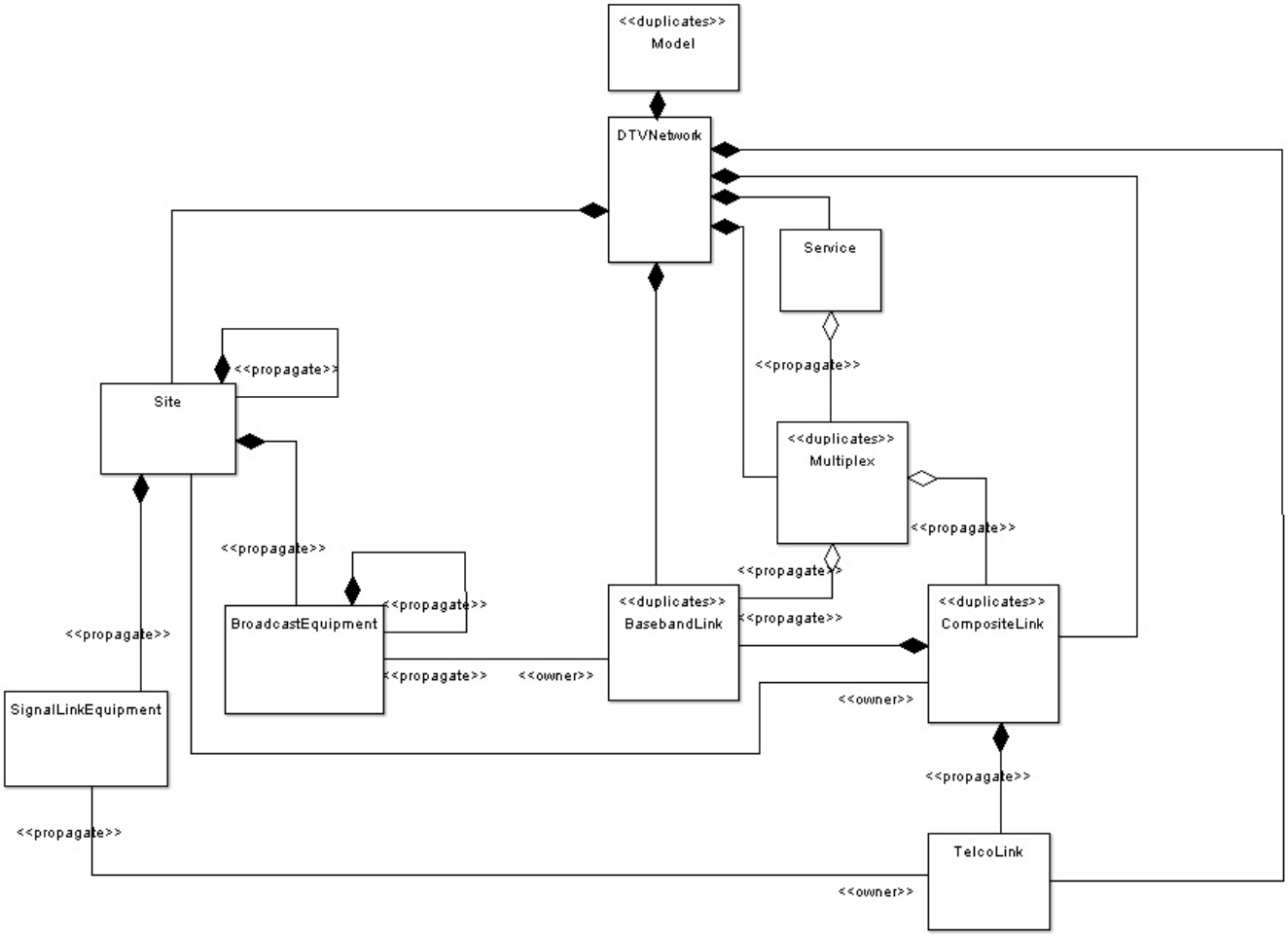
Figure 12 - The Working Memory Object Details window

Chapter 5 Defining the Metamodel

This chapter describes in detail the various aspects of building and deploying the UCA metamodel. In essence, a UCA metamodel is a UML class diagram in the form of an XML file. Although the XML could be created manually, UCA provides a feature that allows a UML class diagram that represents the metamodel to be imported and automatically converted into the UCA XML format.
To illustrate the complete process of building and deploying a metamodel, an example correlation model of a simple digital TV broadcast network is used.

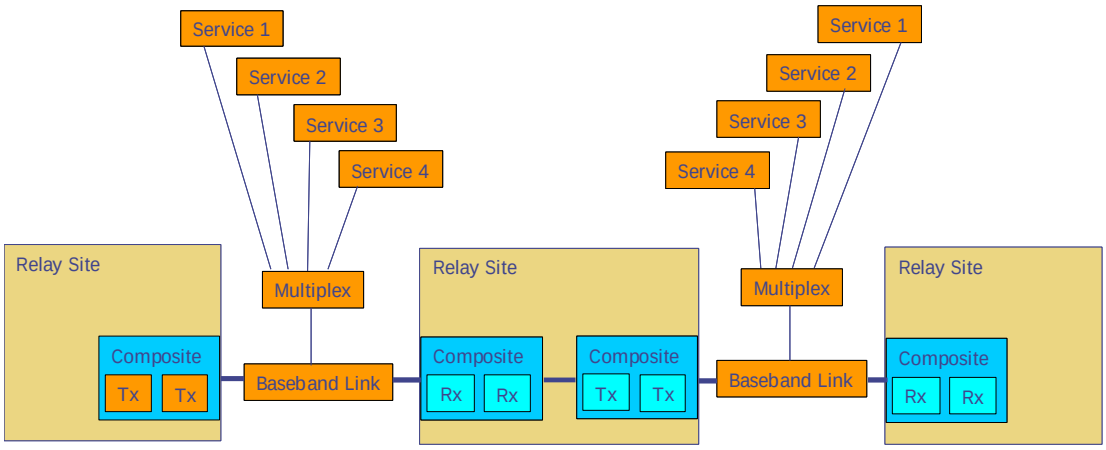
5.1 Example Class Model

The following diagram illustrates the UML class model of a simple digital TV broadcast network that will be converted into an equivalent UCA metamodel.



The <<propagate>> text in the diagram denote those relationships on which automatic state propagation is required (the <<propagate>> text is placed at the 'from' end of the relationship).
The Model class is a top-level container class that must exist in any model converted into a UCA metamodel. It exists to identify the model loaded into UCA and acts as a top-level container for all other classes in the model. In the example, it contains the DTVNetwork class that itself acts a container (directly or indirectly) for all classes in the DTV network model. The Model class may act as a parent to any number of child classes but is unique in that it does not itself have a parent class.
The model shown allows for an arbitrary hierarchy of broadcasting Sites, each of which can contain SignalLinkEquipment classes (representing fixed communications link equipment) and BroadcastEquipment classes (representing on-air broadcast communications link equipment). BroadcastEquipment objects can be joined together by TelcoLink and BasebandLink objects, either separately or at a higher level by CompositeLinks between the Sites themselves. The Multiplex class represents a multiplexed digital TV transmission channel carried over a fixed, on-air or composite Link. Finally, the Service class represents a digital TV service, comprised of one or more components from the Multiplex that it is carried over. This compact set of

objects is all this is required to create a model network of broadcasting sites and is sufficient to perform simple correlations on a DTV network. A fragment from an example model network composed from objects of these classes is shown below:



5.2 Automatic Creation

UCA provides a feature that converts a metamodel in the form of a UML class diagram into the UCA metamodel XML syntax. The class diagram can be created in a UML case tool that supports the export of class diagrams in XMI 1.2, UML version 1.4. Because of UML tool idiosyncrasies and inconsistent compliance to standards, UCA currently supports a single UML tool (ArgoUML) for this purpose. This tool can be invoked from the link at the bottom of the UCA home page.
When creating the class diagram in the UML case tool, UML 'Stereotypes' and 'Tagged Values' are used as follows:

- 'duplicates' is defined as a Stereotype on a class
- 'propagate' is defined as a Stereotype on a relationship endpoint
- 'owner ' is defined as a Stereotype on an association relationship endpoint
- 'hops' is defined as a Tagged Value on an association relationship endpoint
- 'metamodelName' and 'metamodelVersion' are defined using Tagged Values on the 'Model' class.

The class diagram must have 'Model' defined as the top-level class.

To automatically convert the UML class diagram to the corresponding UCA XML syntax, the following steps are needed:

- Create the UML class diagram, making use of UML stereotypes and tagged values as described above (sections below described the meaning of 'duplicates', 'propagate', 'owner' and 'hops').
- Export the UML class diagram as an XMI file. With ArgoUML, this can be done from the '*File* -> *Export as XMI*' menu.
- From the Model tab of the UCA System Manager GUI, select **Import ….**
- Locate and select the XMI file exported from ArgoUML.

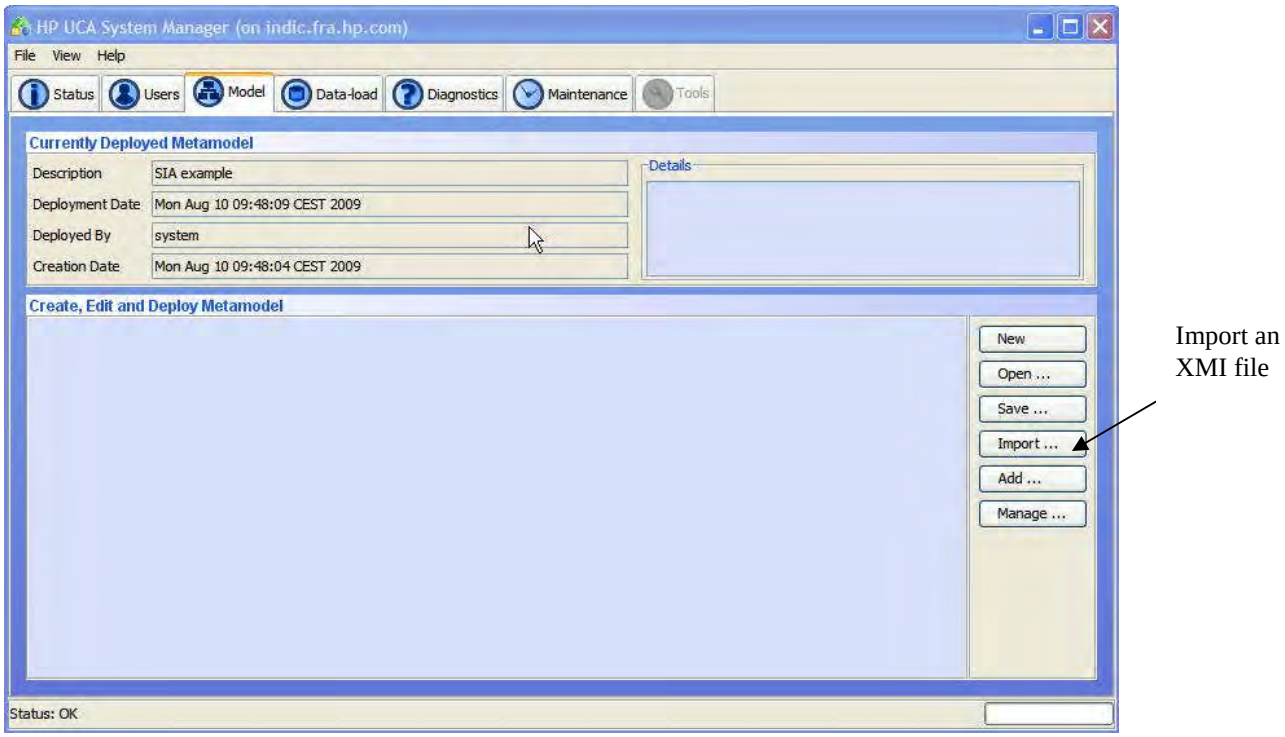


Figure 13 – The Model Tab – Importing an XMI File

If the import is successful, information will be shown in the status bar area. If the import fails for any reason, an error message in red text will be displayed in the status bar area.

5.3 Manual Creation

The UCA metamodel may be created manually, either using a separate text file editor or from within the Model tab of the System Manager GUI.
If the System Manager GUI is used, clicking on the 'New' button will display a template XML definition, including standard header and DTD definition. The user may then manually edit the XML within the section marked as:

```
<metamodel metamodelName="xxxx" metamodelVersion="x.x">
      insert all <element>...</element> definitions here
</metamodel>
```

If a separate text editor is used, then the XML metamodel file may be read in to the text area of the System Manager's Model tab by selected 'Open …' and locating and selecting the appropriate file. In addition, any model file created or read in to the text area of the Model tab may be saved to a local file by selecting 'Save …'. The metamodel XML file for the example digital TV broadcast network model is listed below:

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

<!DOCTYPE metamodel [
<!ELEMENT metamodel (element+)>
      <!ATTLIST metamodel metamodelName CDATA #REQUIRED metamodelVersion CDATA
#REQUIRED>
      <!ELEMENT element (parent | relative | associate | child)*>
            <!ATTLIST element type NMTOKEN #REQUIRED duplicates (TRUE|FALSE)
#REQUIRED>
            <!ELEMENT parent (class, propagate)>
            <!ELEMENT child (class, propagate)>
            <!ELEMENT associate (class, propagate, hops)>
                  <!ATTLIST associate owner (TRUE|FALSE) #REQUIRED>
            <!ELEMENT relative (class, propagate)>
            <!ELEMENT class (#PCDATA)>
            <!ELEMENT propagate (#PCDATA)>
            <!ELEMENT hops (#PCDATA)>
```

```
]>

<metamodel metamodelName="Sidonis DTV Metamodel" metamodelVersion="1.0">

      <element type="DTVNetwork" duplicates="FALSE">
                  <parent>
                        <class>Model</class>
                        <propagate>FALSE</propagate>
                  </parent>
                  <child>
                        <class>BasebandLink</class>
                        <propagate>FALSE</propagate>
                  </child>
                  <child>
                        <class>Site</class>
                        <propagate>FALSE</propagate>
                  </child>
                  <child>
                        <class>CompositeLink</class>
                        <propagate>FALSE</propagate>
                  </child>
                  <child>
                        <class>Service</class>
                        <propagate>FALSE</propagate>
                  </child>
                  <child>
                        <class>Multiplex</class>
                        <propagate>FALSE</propagate>
                  </child>
                  <child>
                        <class>TelcoLink</class>
                        <propagate>FALSE</propagate>
                  </child>
      </element>

      <element type="BasebandLink" duplicates="TRUE">
                  <parent>
                        <class>DTVNetwork</class>
                        <propagate>FALSE</propagate>
                  </parent>
                  <parent>
                        <class>CompositeLink</class>
                        <propagate>TRUE</propagate>
                  </parent>
                  <relative>
                        <class>Multiplex</class>
                        <propagate>TRUE</propagate>
                  </relative>
                  <associate owner="TRUE">
                        <class>BroadcastEquipment</class>
                        <propagate>FALSE</propagate>
                        <hops>0</hops>
                  </associate>
      </element>

      <element type="Site" duplicates="FALSE">
                  <parent>
                        <class>DTVNetwork</class>
                        <propagate>FALSE</propagate>
                  </parent>
                  <parent>
                        <class>Site</class>
                        <propagate>TRUE</propagate>
                  </parent>
                  <child>
                        <class>Site</class>
                        <propagate>FALSE</propagate>
                  </child>
                  <child>
```

```
                        <class>BroadcastEquipment</class>
                        <propagate>FALSE</propagate>
                </child>
                <child>
                        <class>SignalLinkEquipment</class>
                        <propagate>FALSE</propagate>
                </child>
                <associate owner="FALSE">
                        <class>CompositeLink</class>
                        <propagate>FALSE</propagate>
                        <hops>0</hops>
                </associate>
</element>

<element type="CompositeLink" duplicates="TRUE">
                <parent>
                        <class>DTVNetwork</class>
                        <propagate>FALSE</propagate>
                </parent>
                <child>
                        <class>BasebandLink</class>
                        <propagate>FALSE</propagate>
                </child>
                <child>
                        <class>TelcoLink</class>
                        <propagate>FALSE</propagate>
                </child>
                <relative>
                        <class>Multiplex</class>
                        <propagate>TRUE</propagate>
                </relative>
                <associate owner="TRUE">
                        <class>Site</class>
                        <propagate>FALSE</propagate>
                        <hops>0</hops>
                </associate>
</element>

<element type="TelcoLink" duplicates="FALSE">
                <parent>
                        <class>CompositeLink</class>
                        <propagate>TRUE</propagate>
                </parent>
                <parent>
                        <class>DTVNetwork</class>
                        <propagate>FALSE</propagate>
                </parent>
                <associate owner="TRUE">
                        <class>SignalLinkEquipment</class>
                        <propagate>FALSE</propagate>
                        <hops>0</hops>
                </associate>
</element>

<element type="Multiplex" duplicates="TRUE">
                <parent>
                        <class>DTVNetwork</class>
                        <propagate>FALSE</propagate>
                </parent>
                <child>
                        <class>BasebandLink</class>
                        <propagate>FALSE</propagate>
                </child>
                <child>
                        <class>CompositeLink</class>
                        <propagate>FALSE</propagate>
                </child>
                <relative>
                        <class>Service</class>
```

```
                        <propagate>TRUE</propagate>
                  </relative>
      </element>

      <element type="Service" duplicates="FALSE">
                  <parent>
                        <class>DTVNetwork</class>
                        <propagate>FALSE</propagate>
                  </parent>
                  <child>
                        <class>Multiplex</class>
                        <propagate>FALSE</propagate>
                  </child>
      </element>

      <element type="BroadcastEquipment" duplicates="FALSE">
                  <parent>
                        <class>Site</class>
                        <propagate>TRUE</propagate>
                  </parent>
                  <parent>
                        <class>BroadcastEquipment</class>
                        <propagate>TRUE</propagate>
                  </parent>
                  <child>
                        <class>BroadcastEquipment</class>
                        <propagate>FALSE</propagate>
                  </child>
                  <associate owner="FALSE">
                        <class>BasebandLink</class>
                        <propagate>TRUE</propagate>
                        <hops>1</hops>
                  </associate>
      </element>

      <element type="SignalLinkEquipment" duplicates="FALSE">
                  <parent>
                        <class>Site</class>
                        <propagate>TRUE</propagate>
                  </parent>
                  <associate owner="FALSE">
                        <class>TelcoLink</class>
                        <propagate>TRUE</propagate>
                        <hops>1</hops>
                  </associate>
      </element>

</metamodel>
```

Before describing the structure and syntax of a metamodel file in some detail, it should be noted that whilst it is possible to include examples of class specializations into a UML class diagram, such information would not be included in the equivalent metamodel file. This is because the build (run-time data load) provides the specialization i.e. the definition of the Sub Class attribute for each sub-type.
The first section of the file contains XML header and DTD information together with the opening **<metamodel>** tag. This contains mandatory **metamodelName** and **metamodelVersion** attributes for the metamodel itself:

```
            .
            .
            <metamodel metamodelName="Sidonis DTV Metamodel" metamodelVersion="1.0">
```

Each distinct class described in the metamodel class diagram requires an entry in the XML file bounded by the **<element>** **</element>** tag pair. For example, the DTVNetwork class has the following entry:

```
            <element type="DTVNetwork" duplicates="FALSE">
                  ...
            </element>
```

The **<element>** tag has two attributes: the **type** or base class name and whether or not **duplicates** are allowed. Both are mandatory and the latter field is normally FALSE, however it is set to TRUE when the same object can be loaded with different unique references e.g. if it has a number of alias names.
Within the **<element> </element>** tag pair, additional tag pairs may be defined, specifying the possible types of relationships that objects of this type can enter into. In addition, a relationship must be defined in each class that participates in that relationship i.e. at both ends, and this must be done in context. For example, with a parent/child relationship, the parent class defines a <child> relationship and the child class defines an equivalent <parent> relationship. The additional tag pairs are:
<parent> </parent>. This tag pair is required at least once in every aggregated or child class (because every class has a parent) and defines the parent class type in a composite relationship. It contains two additional tag pairs:

<class>{PARENT BASE CLASS}</class>, the base class of the parent
<propagate>{TRUE|FALSE}</propagate>, whether automatic state propagation is required to the parent object
e.g.

```
<element type="Site" duplicates="FALSE">
        <parent>
            <class>DTVNetwork</class>
            <propagate>FALSE</propagate>
        </parent>
        ...
</element>
```

Note that the top-most class in a state mesh - usually some type of network - has a parent class of Model, for which UCA automatically generates the required support and no entry is required in the XML file.

<relative> </relative>. This tag pair is used in a contained or nephew class and defines the containing or uncle class type in a containment relationship (hence the name <relative>). It contains two additional tag pairs:

<class>{RELATIVE BASE CLASS}</class>, the base class of the relative
<propagate>{TRUE|FALSE}</propagate>, whether automatic state propagation is required to the relative
e.g.

```
<element type="Multiplex" duplicates="TRUE">
...
    <relative>
        <class>Service</class>
        <propagate>TRUE</propagate>
    </relative>
</element>
```

Note in this example that the **duplicates** attribute is set to TRUE. This allows a Multiplex object to be data loaded several times if it supports more than one Service uncle object (only one uncle object can be specified at a time in a single data load block). Also note that the **propagate** attribute is set to TRUE. This means that any state change on a Multiplex object will be propagated automatically to all Service uncle objects.

<child> </child>. This tag pair is used in a composite (or parent) class OR in an aggregate (or uncle) class. In a composite relationship, it defines the child class type. In an aggregate relationship, it defines the contained or nephew class type. It contains two additional tag pairs:

<class>{CHILD BASE CLASS|NEPHEW BASE CLASS}</class>, the base class of the child or nephew class.
<propagate>{TRUE|FALSE}</propagate>, whether automatic state propagation is required to the child or nephew
e.g. Nephew

```
<element type="Service" duplicates="FALSE">
    ...
    <child>
        <class>Multiplex</class>
        <propagate>FALSE</propagate>
    </child>
</element>
```

Note that in this example the **duplicates** attribute is set to FALSE. This is because a Service object is only loaded once even though it may have many Multiplex nephew objects

e.g. Child

```
<element type="Site" duplicates="FALSE">
		...
		<child>
			<class>Site</class>
			<propagate>FALSE</propagate>
		</child>
		<child>
			<class>BroadcastEquipment</class>
			<propagate>FALSE</propagate>
		</child>
		<child>
			<class>SignalLinkEquipment</class>
			<propagate>FALSE</propagate>
		</child>
	...
</element>
```

Note that a Site parent object may singly or simultaneously have a number of different types of child object, including other Site objects (this allows a hierarchy of Sites as defined in the metamodel UML class diagram).

<associate> </associate>. This tag pair is used in 'associate' or peer classes and defines the peer class type in an association relationship. The <associate> tag has an **owner** attribute that defines the class type at one end only of an association relationship as the nominal owner, i.e. set to TRUE. Obviously, the class type at the other end of the same relationship must have this attribute set to FALSE. The purpose of this tag is to force the bi-directional relationship in the state mesh to be constructed from the 'owning' end only. The **<associate> </associate>** tag pair contains three additional tag pairs:

<class>{PEER BASE CLASS}</class>, the base class of the remote peer class.
<propagate>{TRUE|FALSE}</propagate>, whether automatic state propagation is required to the remote peer
<hops>{0|n}</hops>, the extent of automatic state propagation, usually 0 or 1 objects (0 means don't propagate to associate, even if **propagate** attribute is set to TRUE. 1 means propagate to associate and no further if **propagate** attribute is set to TRUE).

e.g. Relationship owner, BasebandLink objects does not propagate state changes to associate BroadcastEquipment objects

```
<element type="BasebandLink" duplicates="TRUE">
	...
		<associate owner="TRUE">
			<class>BroadcastEquipment</class>
			<propagate>FALSE</propagate>
			<hops>0</hops>
		</associate>
</element>
```

e.g. Relationship non-owner, BroadcastEquipment objects propagate state changes to associate BasebandLink objects.

```
<element type="BroadcastEquipment" duplicates="FALSE">
	...
		<associate owner="FALSE">
		<class>BasebandLink</class>
		<propagate>TRUE</propagate>
		<hops>1</hops>
	</associate>
	 </element>
```

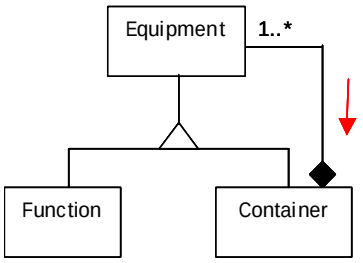
5.4 Metamodel Design Patterns

Developing UCA solutions in a number of application areas has resulted in the use of some common design patterns for metamodel components. The following sections describe some of the more useful examples using annotated UML class diagrams and where appropriate, associated correlation models.

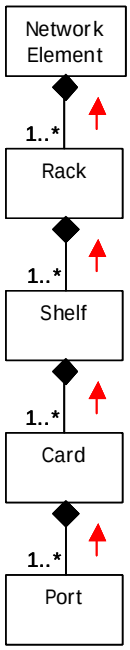
5.4.1 Equipment Tree

The Equipment Tree pattern describes a common arrangement for building hierarchical network equipment models in UCA and includes annotations for model-driven state propagation between equipment layers. The following UML class diagrams illustrate the general form of the pattern and an example Equipment Tree metamodel fragment for a telecommunications Network Element containing a hierarchical arrangement of sub-components:

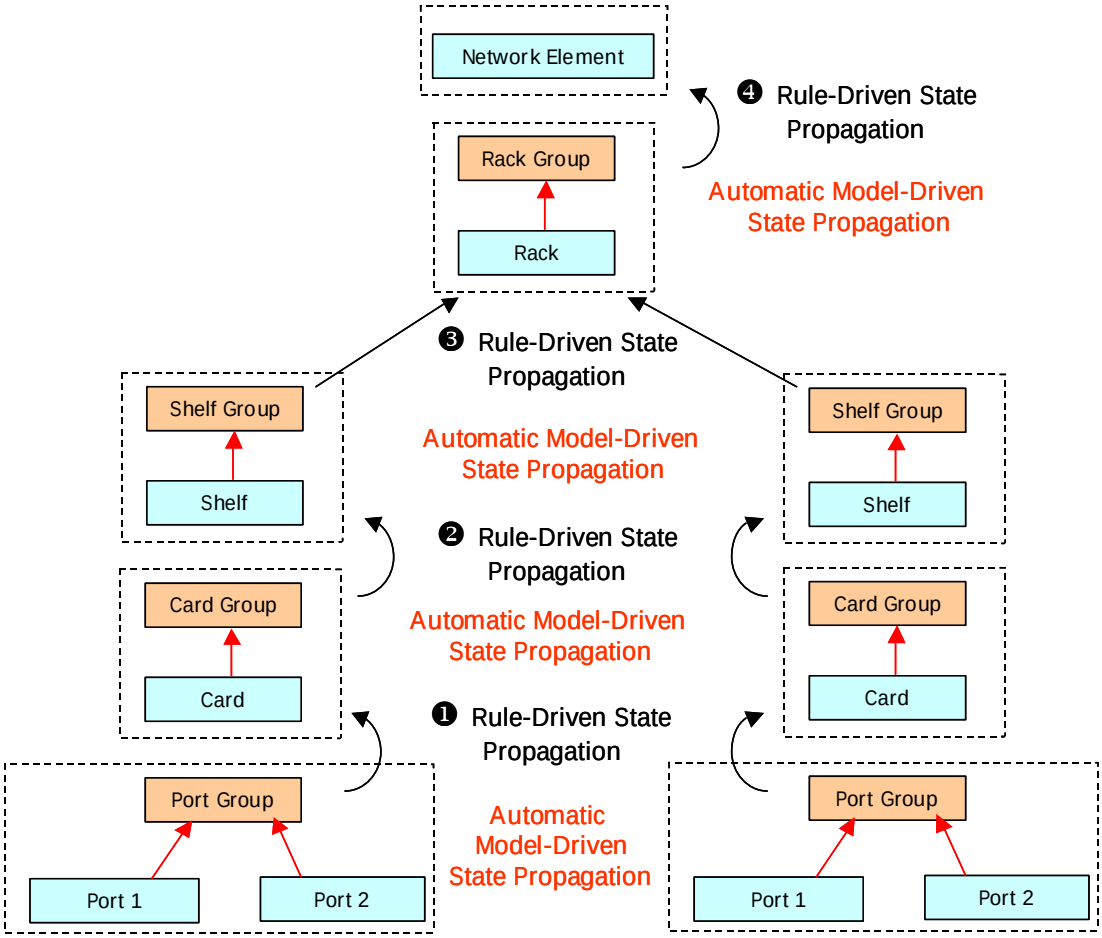
General Form



Example Equipment Tree



The general form of the pattern allows the user to construct an arbitrarily complex layered equipment model including model-driven state propagation between the layers. The example Equipment Tree illustrates how the general form may be used to construct a specialisation for a particular application domain and this would normally be used as part of a UCA metamodel.
A correlation model, based on the example Equipment Tree, is illustrated in the following diagram to show how the pattern specialisation would be used in practice.



The import facility provided with UCA may be used to convert the Equipment Tree UML class model (in XML format) into a UCA metamodel, capable of supporting the correlation model including automatic model-driven state propagation.
The user is then left to construct and deploy the simple rules and actions necessary to handle the propagation of state changes between layers of the model and carry out consequent actions. For example, a design choice might be to build into rule ❶ an assumption that a Card has failed when 75% of the Ports on that Card have themselves failed. As well as reporting the failure into the enclosing Shelf, the designer could instigate an action to attempt an automatic reset of the Card itself.
The important point to consider is that the combination of automatic model-driven state propagations and the flexibility of user-defined rule-driven state propagations allows the correlation designer to achieve a very flexible handling and reporting structure. In addition, the single metamodel definition and accompanying rule/action set will apply equally to all Network Elements for which data is loaded into UCA, regardless of the actual number of Ports, Cards, Racks and Shelves in each instance.

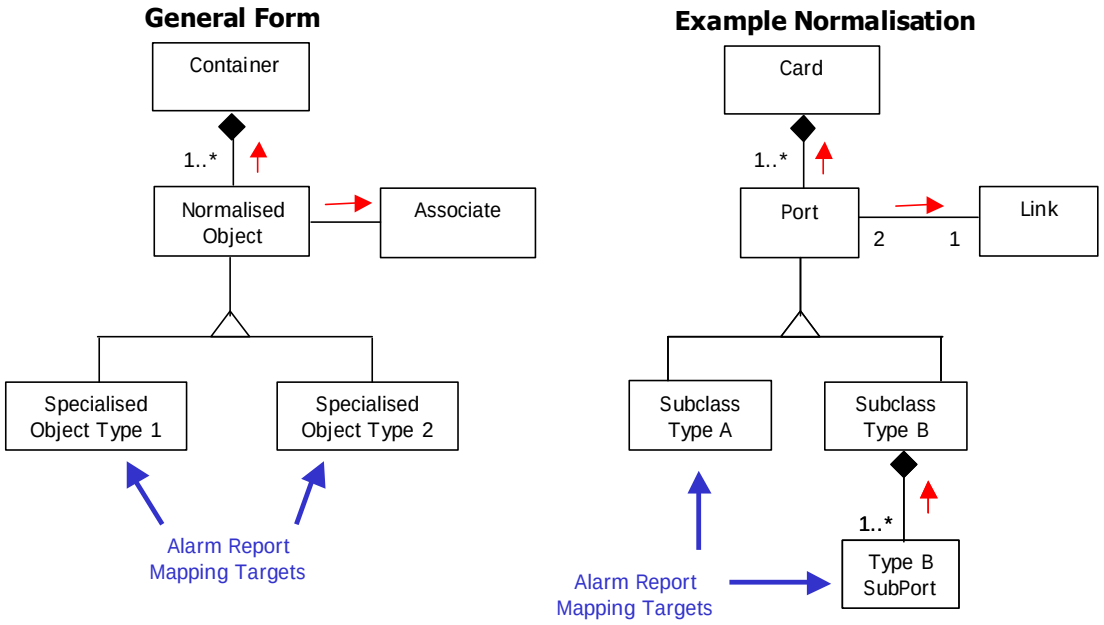
5.4.2 Normaliser

Networks are constructed from a diverse range of components within and across the ranges of equipment supplied by different manufacturers. For simple correlation scenarios or test implementations, it may be advantageous to provide individual correlation models for each variation, however as the extent of the implementation increases it becomes important to adopt techniques that promote simplification and re-use and minimise the maintenance effort.
One technique that can be usefully employed at the lowest level of the correlation ‘pyramid’ is to normalise this diversity into a common logical representation that then drives the correlation layers above in a uniform manner. Diversity at this level usually manifests itself in the following ways:

- Alarm reports received from network equipment vary widely in their reporting standard, complexity and severity (even in different revisions of the same equipment supplied by a manufacturer).

- The complexity of the logical implementation model for the same type of equipment varies widely across product ranges and manufacturers. The result is that the same problems are often reported in completely different ways.

UCA supports the normalisation of this diversity into a common logical form using the Normaliser pattern described in the following UML class diagrams.



The Normaliser pattern is derived from the widely used class form of the Adapter pattern. It achieves the normalisation process through two mechanisms:

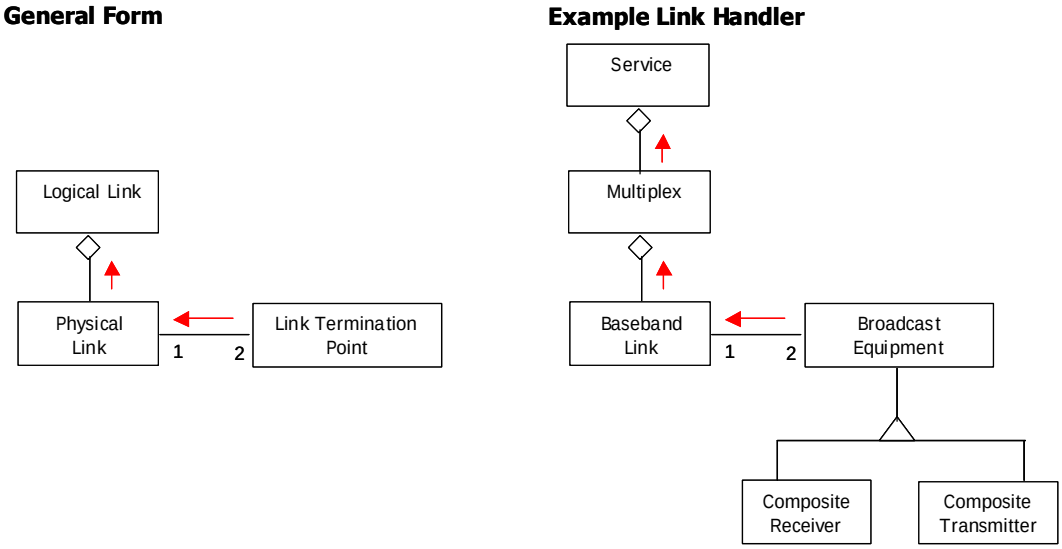
- Diverse alarm reports are mapped onto instances of the specialised object types in the correlation model, using the comprehensive target object mapping capabilities provided by UCA. The mapping is configured such that regardless of the type of alarm report, it causes the same state change to be applied to the target object.
- Specialised object types (reflecting the diversity of the network implementation) are provided with a common base class that serves as the normalised 'logical' driver for correlation at higher levels. The state change caused by mapping alarm reports onto a specialised object affects the encapsulated base class instance equally.

The Rules that drive higher-level correlations are then written to operate on instances of objects with the common base class – they effectively ignore the diversity and look only for objects of the base class type in working memory (rather than their subclass type which reflects their diversity).
Of particular interest in the example normalisation model shown above is the extra level of diversity in the Type B specialised object type. The Type A subclass has alarm reports attached directly by UCA as described above. In contrast, the Type B subclass has alarm reports attached to its set of SubPorts (because it does not itself directly generate alarm reports and mapping those from the SubPorts to the owning Type B instance would not achieve the correct effect).
In order to achieve the required normalisation from Type B objects, the designer is required to provide a simple rule to detect when the required proportion of Type B SubPorts have themselves changed state and a corresponding action to force the owning Type B subclass instance to the failed state. Once this has been implemented however, the correlation scenario will operate equally with either Type A or Type B objects.

5.4.3 Link Handler

Most communication networks are constructed at the equipment level from a mesh of control or switching elements and some type of transport medium to communicate information or switch resources between them e.g. radio link, fibre optic cable. In general, this level of complexity is insufficient to provide the level of resilience to failure required by modern service level agreements or to support the diverse range of services offered.
In practice therefore, these types of network employ a logical network model above the physical level, supporting a number of layers of increasing abstraction (and usually complexity). A common characteristic of

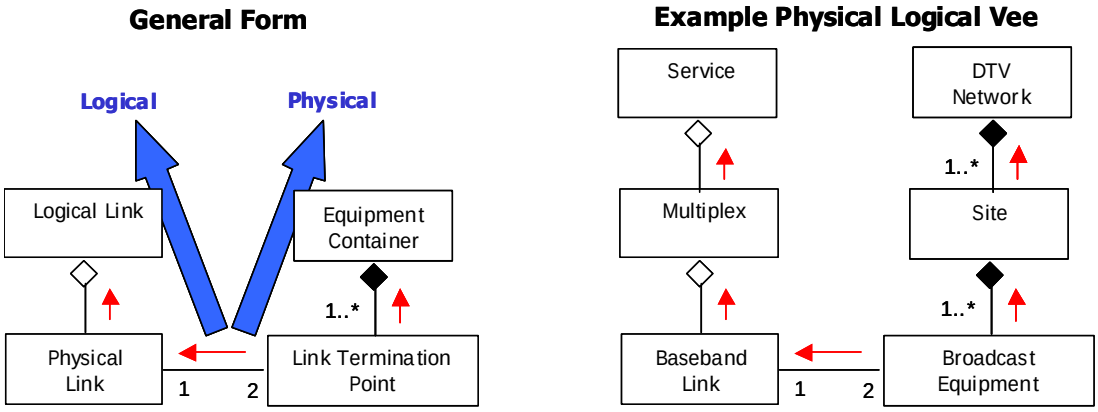
each layer however is that it is dependent on a lower layer for service and correlation scenarios usually involve determining the effect of a problem at a lower layer on those above.
The practical problem in constructing these scenarios is surprisingly not the issue of modelling the inter-layer dependencies or associated state propagations but that of obtaining a suitable generic 'driver' from the physical layer to provide an initial trigger. This is because the physical links e.g. cables, radio links, fibres etc. on which logical links are carried do not themselves generate Alarm Reports. In general, the only vaguely useful Alarm Reports are those reported against the equipment at each end of the physical links and by themselves they are unsuitable for reliably triggering a scenario. This is because the receipt of an Alarm Report from one end of a link is not always a reliable indicator of link failure.
The purpose of the Link Handler pattern is twofold. It provides the connection between physical equipment and associated link problems and supports the generation of a reliable generic 'driver' into the logical layer above. The general form of the pattern and a metamodel fragment that employs it (from the DTV network example included with UCA) are illustrated in the following UML class diagrams.



The pattern operates in two stages. First, Alarm Reports delivered against the Link Termination Point objects result in state changes that are propagated to a Physical Link object using automatic model-driven state propagation. One or more rules provided by the correlation designer detect when the ends of the Physical Link have attained the required combination of states e.g. failed + failed or failed + degraded (as required by the correlation scenario) and the consequent action forces the state of the Physical Link to failed. Next, automatic model-driven state propagation reports the state change of the Physical Link upwards to the carried Logical Link, thus achieving the requirement to provide a generic 'driver' into that layer.
The example Link Handler illustrates how this pattern may be incorporated into the UCA metamodel. The structure shown actually reports a Baseband Link (i.e. the Physical Link) failure up to the Multiplex transported over it, which in turn reports a problem to the Services carried on the Multiplex. It also utilises the Normaliser pattern to handle equipment diversity, using the Broadcast Equipment base class to represent the Composite Receiver and Transmitter objects at either end of the Baseband Link and thus simplifying the implementation.

5.4.4 Physical-Logical Vee

The Physical-Logical Vee pattern is in fact a combination of the Equipment Tree (Physical) and Link Handler (Logical) patterns described above and forms the basis of many correlation scenarios that operate on communications networks. The pattern allows a designer to implement scenarios that simultaneously handle two important aspects of correlation analysis – problem detection on the (physical) equipment level and impact analysis on the (logical) service impact level. The following UML class diagram illustrates the general form of the pattern and shows the practical application of it in the DTV network metamodel.



Considering the general form, alarm reports attached to Link Termination Points have two simultaneous effects. State changes are propagated directly upwards to the Equipment Container in the physical arm of the 'Vee', allowing the designer to construct problem detection correlation scenarios. The same state changes are propagated towards the Physical Link and consequential state changes are propagated upwards to the Logical Link in the logical arm of the 'Vee', allowing the designer to build simultaneous service impact correlation scenarios. Of particular interest in this pattern is the simultaneous use of relative, parent and peer relationships to achieve the desired results.

The metamodel fragment shown is taken directly from the included DTV network example and illustrates a practical application of this pattern.

Chapter 6 Creating the Model Database Using the System Manager

One of the purposes of the UCA metamodel is to act as a template for structuring the UCA model database. The classes and relationships defined within the metamodel drive the whole process of setting up the structure of the tables within the UCA model database. Assuming the metamodel is defined, UCA automates the entire process of generating these tables and defining their structure. Once the model database tables have been created, the remaining task is to populate these tables with actual model data.
This chapter describes the process of creating and populating the model database tables.

6.1 Generating the Model Database Structure

Before a metamodel can be 'deployed' (i.e. used to automatically create the model database tables), it must first be added to the metamodel library within UCA. This library acts as a storage repository for deployable metamodels. Any number of them may be stored in the library but only one metamodel can be deployed into active use at any time.

- To store the metamodel currently displayed in the text area of the Model tab of the System Manager, select 'Add …', supply a description and additional information that distinguishes this metamodel, then select 'OK'.
- To view the current set of deployable metamodels stored in the metamodel library, select 'Manage …' from the Model tab. This will list the details of all metamodels within the library.
- To view the contents of a particular metamodel stored in the metamodel library, select 'Manage …' from the Model tab, select the metamodel of interest and click on 'Open'. The metamodel will then be listed in the text area of the System Manager's Model tab.
- To deploy a metamodel in to active use, select 'Manage …' from the Model tab, select the required metamodel to deploy and click on 'Deploy'. ***Note that this is a destructive operation - all data held in the model database will be destroyed***. After accepting the warning confirmation, a dialog will prompt for the model database maximum field size – enter a value at least as big as the largest data item (usually the Unique Reference) expected to fill a model database field during population and click on 'OK'. The model database tables will then be automatically created from the metamodel.

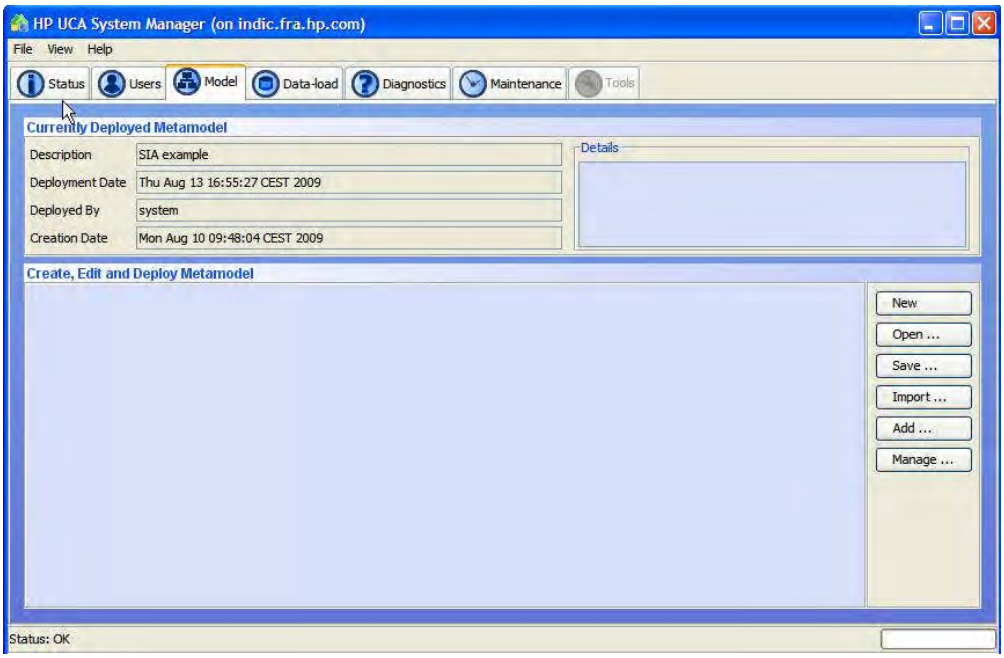


Figure 14 – The Model Tab – meta-model management

The fragment below illustrates the type of definition generated within the model database from the metamodel for a CompositeLink class.
Note that the 'create table' statement will vary according to the types of relationship defined for the class in the metamodel. The Relative and Associate sections marked in the example above will only exist if the class participates in composition and association relationships respectively. On the other hand, the Parent and Class Section will always exist (a class always has a parent even if it the special top level Model class).
Even though a class can have any number of composition and association relationships, only one set of data is allowed for these sections per row in the appropriate model database table. To allow data loading of these multiple relationships for a given target object, the user simply provides multiple data sets with the same Parent and Class attribute values and different Relative and/or Associate attributes. Further, to prevent the model builder reporting errors for duplicate entries for the same instance of a class, the user is required to set the **duplicates** attribute in the metamodel for the class to TRUE. The model builder then constructs a single target object instance but correctly sets up the multiple Relative and/or Associate relationships as required.

```
CREATE TABLE UCA.MD_COMPOSITELINK (
      "Parent_Ref" varchar2(250) default '',
      "Parent_Subclass" varchar2(250) default '',
      "Parent_Class" varchar2(250) default '',
      "Relative_Ref" varchar2(250) default '',
      "Relative_Subclass" varchar2(250) default '',
      "Relative_Class" varchar2(250) default '',
      "A_Associate_Ref" varchar2(250) default '',
      "A_Associate_Subclass" varchar2(250) default '',
      "A_Associate_Class" varchar2(250) default '',
      "Z_Associate_Ref" varchar2(250) default '',
      "Z_Associate_Subclass" varchar2(250) default '',
      "Z_Associate_Class" varchar2(250) default '',
      "Class_Name" varchar2(250) default '' NOT NULL,
      "Subclass_Name" varchar2(250) default '' NOT NULL,
      "Instance_Name" varchar2(250) default '' NOT NULL,
      "Unique_Ref" varchar2(250) default '' NOT NULL,
      "Service_State" varchar2(20)
                        default 'IN_SERVICE' NOT NULL,
      "Importance" varchar2(5) default '0' NOT NULL,
      "Latitude" varchar2(20) default '0' NOT NULL,
      "Longitude" varchar2(20) default '0' NOT NULL
) TABLESPACE UCA;
```

Parent Section

Relative Section

Associate Section

Class Section

6.2 Populating the Model Database

Once the UCA model database tables have been created, they must be populated with real data representing the actual Sites, CompositeLinks etc. There are two aspects to this:

- The initial data population, starting from empty tables
- The 'day-to-day' updating of the tables due, for example, to periodic inventory changes in an operational network.

The following sections describe the two processes involved.

6.2.1 Initial Population

There are many possible techniques for populating the model database tables with data. For example, if Comma Separated Value (CSV) data files are to be imported then Oracle's SQL*Loader or the PostgreSQL COPY command might be used. Alternatively, table data may be directly imported using facilities provided with the DBMS.
Alternatively, UCA provides a facility for CSV file import intended for use when a relatively small (tens of thousands) number of objects are to be imported. To use this facility, select the Data-load tab in the UCA Manager (note that loading UCA with an initial set of model data can only be done when UCA is not started).

- The available classes of model data, as defined in the metamodel, will be listed on the left side. The right side lists the CSV files available for import. Files available for import are those in the 'import' subdirectory of the UCA installation directory on the server. Files may be uploaded to

this directory from a client using the **Upload …** button on the Data-load tab or manually copied in from another location. These files may also be deleted from the server by selecting the file and then clicking the **Delete** button.

- When creating CSV files to import, it is useful to know the exact order of fields to use on each line of CSV data. To assist with this, clicking the **CSV Help** button will list all the tables, field names, types and sizes.
- To associate a class with a CSV file, select the class on the left of the window and the associated CSV file on the right. Then click on **Associate**. Details for this class / file association will then be listed in the text area at the bottom of the window. Repeat this process for all classes and CSV files to be associated.
- Finally, to import all the CSV files for each class, click on the **Import** button. You will be given the choice of over-writing existing table data or appending the new data to the existing data.

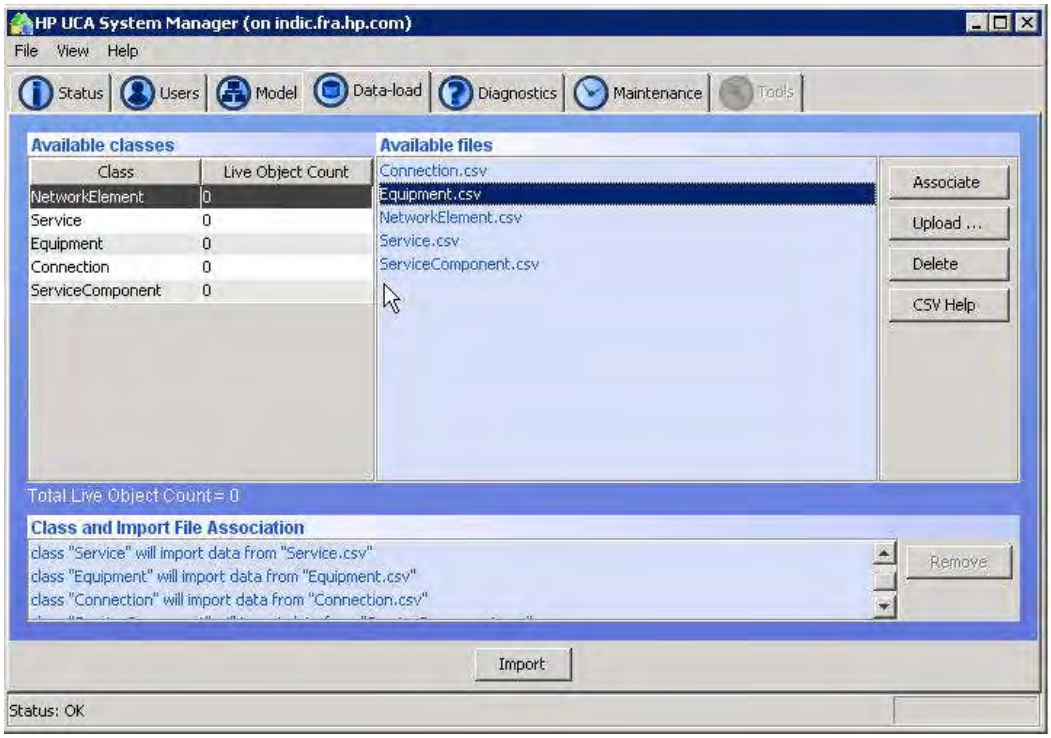
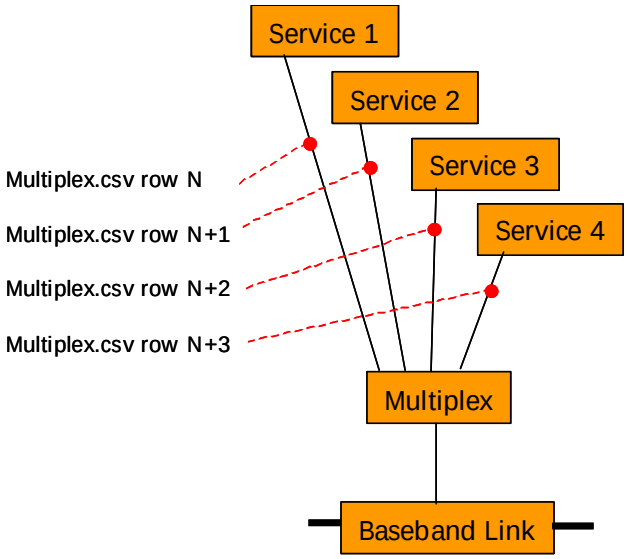


Figure 15 – The Data-load Tab – inventory management

Below is an example where multiple rows of data are provided for a single instance of an object to configure multiple Relative relationships to different 'uncle' objects, as described at the end of the preceding section. Each Multiplex object is listed several times to allow a number of Relative relationships to be defined to different uncle Service objects. This is illustrated in the following fragment of the example network:

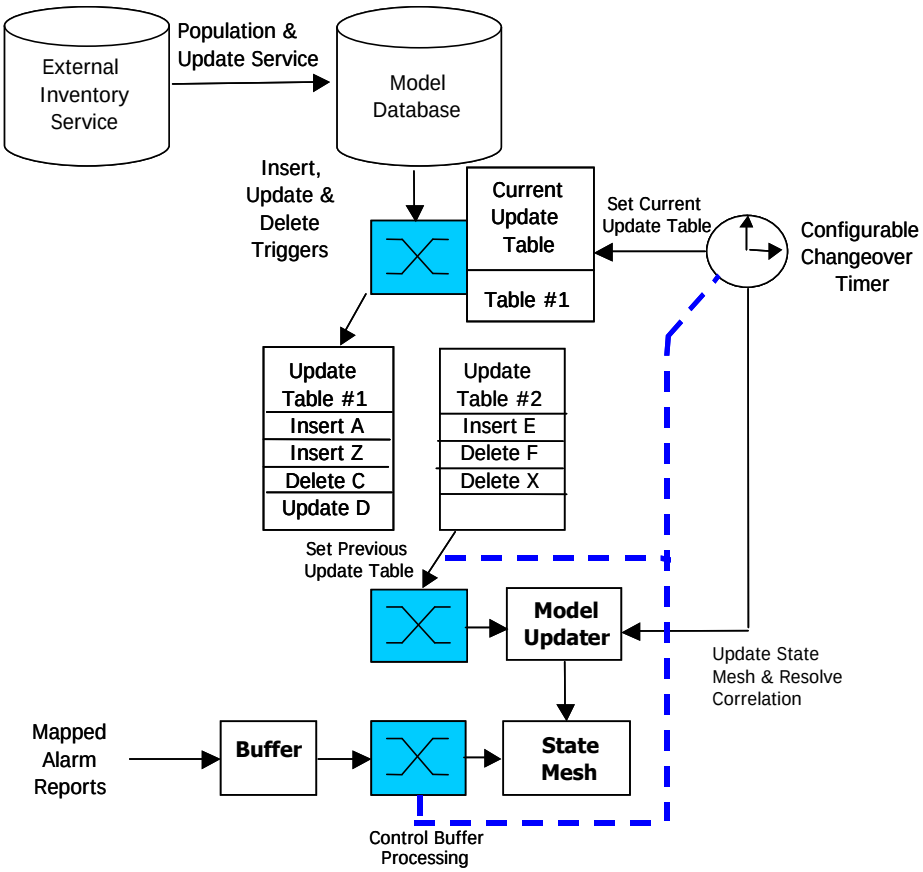


6.2.2 Updating the Database

Once the model database has been populated with target object and relationship instance data, the UCA system may be started as described previously.

Assuming that the UCA system is operational, the model database may be updated at any time, with the effect that the state mesh will be dynamically updated and any ongoing correlations automatically resolved as far as possible to maintain system consistency. The dynamic update mechanism is illustrated below:

For efficiency the UCA system is designed to gather a set of model database updates over a configurable period and apply them in a single operation. Therefore, the state mesh will only be updated at those times defined by system configuration e.g. once an hour at 30 minutes past the hour. This means that changes applied to the model database are unlikely to be applied to the state mesh immediately unless this coincides with the next state mesh update time or the user selects the 'Update Mesh Now' option in the UCA System Manager Maintenance tab. The frequency of update will have been configured by the system administrator (using the UCA System Manager Maintenance tab - see the *HP UCA Installation and Configuration Guide* for details). The time and frequency should be chosen to provide a balance between operational needs and system efficiency, bearing in mind that an update requires the system to temporarily suspend (and buffer) the processing of alarm reports.



When a change is applied to any model database table during the 'gathering' period, details of the change will be recorded in a special 'Update' table. At the next update time, any changes recorded in the 'Update' table are applied to the state mesh. To ensure that updates are not lost during this operation, the system maintains a pair of 'Update' tables that are used alternately – while one set of updates is being applied, any new updates will be recorded in the alternate 'Update' table. If for consistency reasons it is important that a set of updates should not be split between two successive updates, care should be taken to ensure that a model database update is not carried out close to an update time.
It is then the responsibility of the user to implement and configure a regularly repeated task e.g. a 'cron' or batch job, to extract a set of updates from the external inventory service and apply these to the appropriate model database tables. As described above, the state mesh will then be automatically updated at the next update time.

Chapter 7 The UCA Applications

UCA provides three main Graphical User Interface (GUI) applications:

- the System Manager
- the Scenario Manager
- the Mesh Viewer

The System Manager is used for system administration, model loading, diagnostics and maintenance and is covered in Chapter 4 and the *HP UCA Installation and Configuration Guide*.
The Scenario Manager is used for defining and deploying scenarios, filters, mappings and rules. The Mesh viewer is used for viewing the structure and contents of the model as well as the real-time state of mesh events and notifications. These two GUIs are described in detail in the following chapters.
To invoke the Scenario Manager or Mesh Viewer, click on the **UCA Applications** button in the UCA Home Page. A web page will be displayed requesting a username and password, as follows:

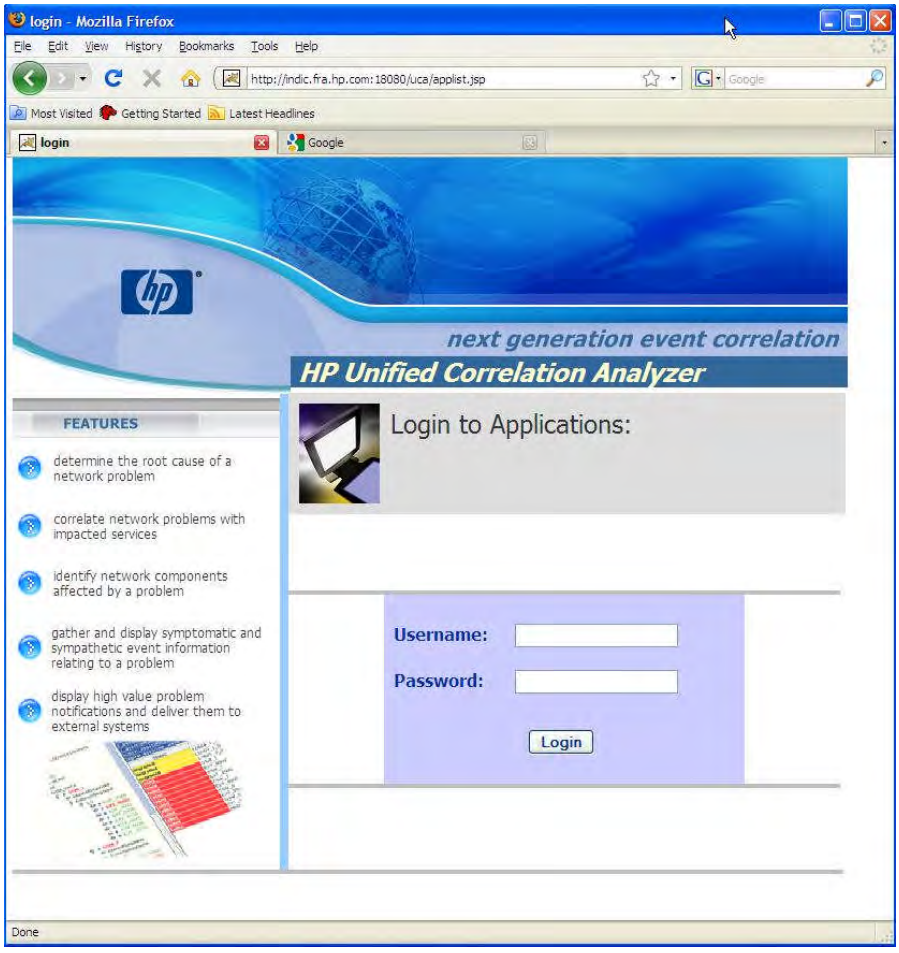


Figure 16 - The Applications Login Page

After entering a valid username and password and clicking on the Logon button, a page will be displayed showing the UCA applications that the user is authorised to use, as shown below (see section 4.3 for details of how roles affect allowed applications).
To start the Scenario Manager or the Mesh Viewer, click on the appropriate button.

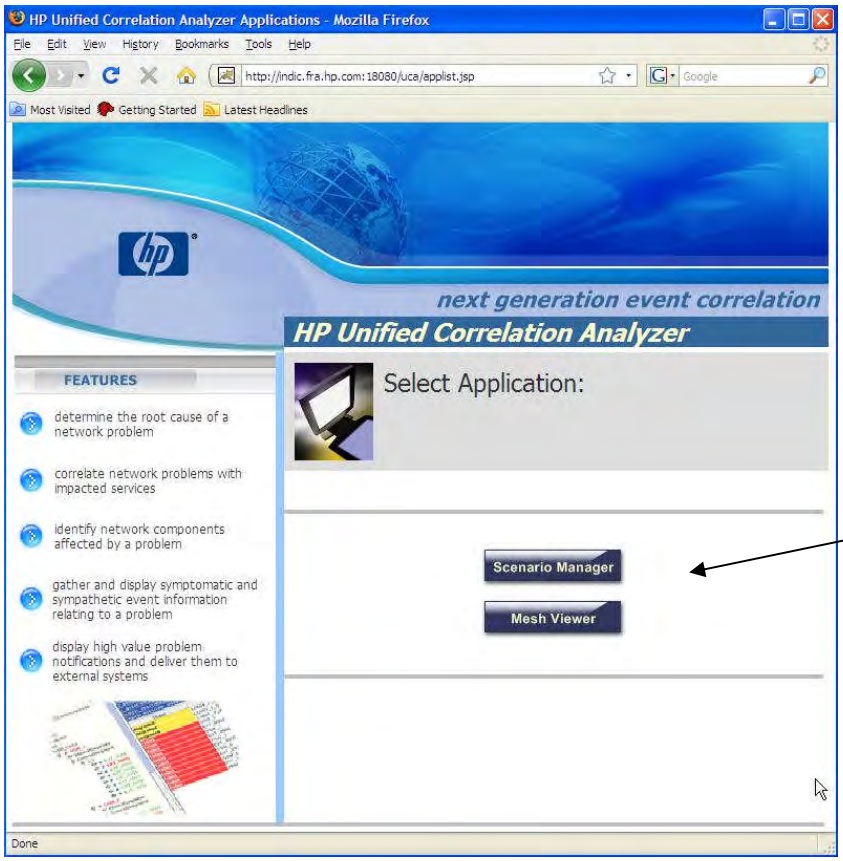


Figure 17 - The UCA Applications Page

7.1 The Scenario Manager

This section describes the features available in the Scenario Manager in terms of the basic menu items, toolbar items, pop-up menu options and so on. A detailed description of how to actually configure the scenarios, filters, mappings and rules is provided in the subsequent chapters.
The Scenario Manager is used for:

- creating, modifying and deleting scenarios, filters, mappings and rules
- validating the 'correctness' of scenarios before deploying them
- deploying a set of scenarios, filters, mappings and rules into active use
- listing details of previous deployments
- maintaining and using a 'library' of deployments

The following screenshot shows the Scenario Manager with the main component areas labelled.

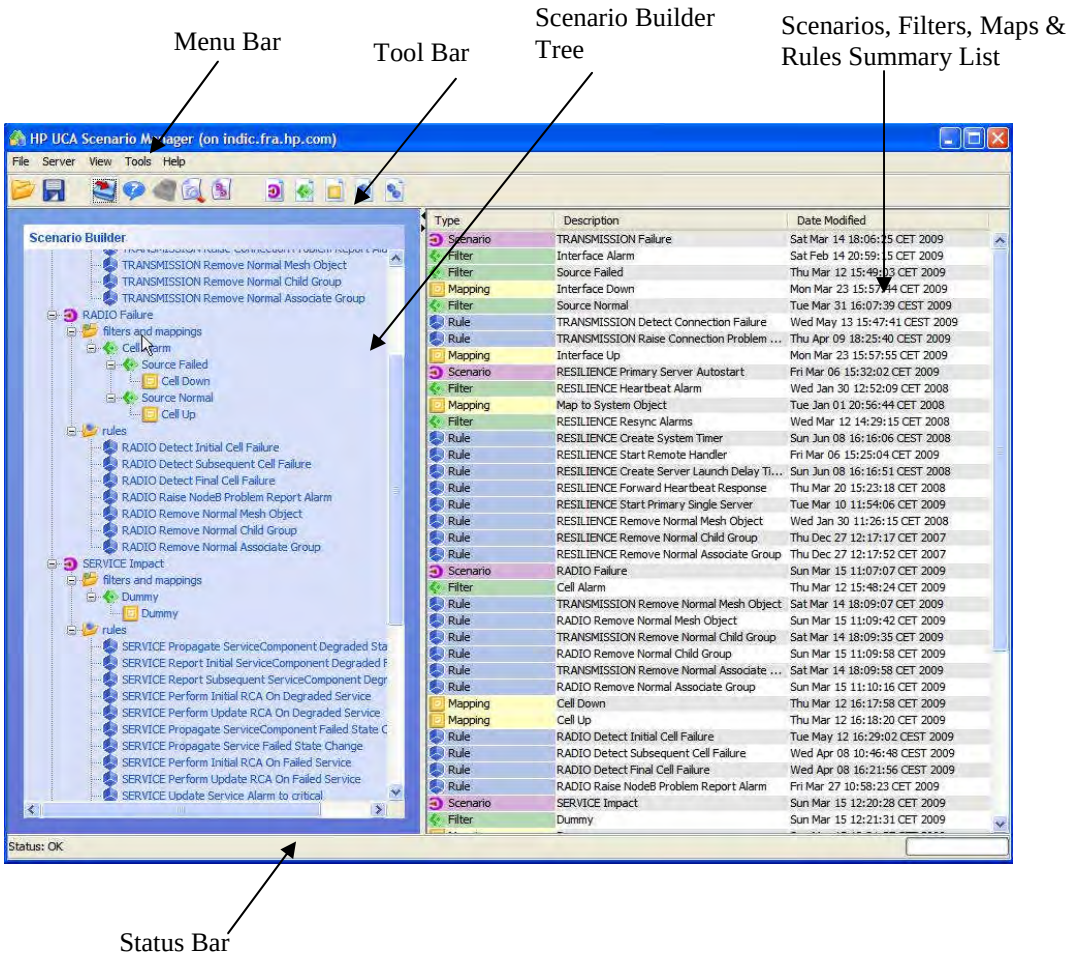


Figure 18 - The Scenario Manager

7.1.1 Menu Bar

The following menu items are available:

File → New→ Scenario	Opens the 'Add New Scenario' dialog box.
File → New→ Filter	Opens the 'Add New Filter' dialog box.
File → New→ Mapping	Opens the 'Add New Mapping' dialog box.
File → New→ Rule	Opens the 'Add New Rule' dialog box.
File → Open from local file	Opens a local file of scenarios, filters, mappings and rules.
File → Save to local file	Saves the current set of scenarios, filters, mappings and rules to a local file.
File → Save multiple scenarios	Saves a selection of scenarios from the scenario builder tree.
File → Print→ Table Summary	Prints the current contents of the Scenarios, Filters, Mappings and Rules Summary List..
File → Print→ Tree Summary	Prints the current contents of the Scenario Builder Tree.
File → Print→ All Details as XML	Prints all details of all configured scenarios, filters, mappings and rules in XML format.
File → Exit	Exits the application.
Server → Load Data	Loads the currently deployed scenarios, filters, mappings and rules from the server into view, replacing all currently displayed data.
Server → Validate Data	Validates the scenarios, filters, mappings and rules in the Scenario Builder Tree. Data cannot be deployed until it has been validated.
Server → Deploy Data	Deploys the validated scenarios, filters, mappings and rules in the Scenario Builder Tree to the server. The user is

	prompted to enter a description and additional information related to the deployment.
Server → Show Deployments	Shows the 'Deployments' dialog listing details of username, date, description and additional information for every deployment. A deployment may be selected in the 'Deployments' dialog and Opened, so that the Scenario Builder Tree and Summary List contents are replaced with of the selected deployment.
Server → Show Library	Shows the 'Scenario Library' dialog listing details of username, date, description and additional information for each scenario exported to the library. An exported scenario may be selected in the 'Scenario Library' dialog and Merged, so that the scenario contents are merged into the scenarios branch of the Scenario Builder Tree.
View → Look and Feel → …	Changes the look and feel of the GUI according to those supported on the client platform eg. CDE/Motif, Windows, Metal.
View → Toggle tree node Ids	Toggles the display of the internal unique Ids for each scenario, filter, mapping and rule in the Scenario Builder Tree.
Tools → Purge Summary Table	Deletes all scenarios, filters, mappings and rules in the Summary List that are not in the Scenario Builder Tree.
Tools → allow rules to loop?	Enables or disables the looping of rules using the JBoss Rules internal looping activation / deactivation.
Help → Scenario Manager Help	Displays Scenario Manager help information in a web page.
Help → Sidonis web site	Displays the Sidonis web page.
Help → About	Displays a dialog showing the UCA and Scenario Manager version numbers.

7.1.2 Tool Bar

Clicking on an icon in the tool bar performs the action as follows:

Icon	Action
	Opens a local file of scenarios, filters, mappings and rules.
	Saves the current set of scenarios, filters, mappings and rules to a local file.
	Loads the currently deployed scenarios, filters, mappings and rules from the server into view, replacing all currently displayed data.
	Validates the scenarios, filters, mappings and rules in the Scenario Builder Tree. Data cannot be deployed until it has been validated.
	Deploys the validated scenarios, filters, mappings and rules in the Scenario Builder Tree to the server. The user is prompted to enter a description and additional information related to the deployment.
	Shows the 'Deployments' dialog listing details of username, date, description and additional information for every deployment. A deployment may be selected in the 'Deployments' dialog and Opened, so that the Scenario Builder Tree and Summary List contents are replaced with of the selected deployment.
	Shows the 'Scenario Library' dialog listing details of username, date, description and additional information for each scenario exported to the library. An exported scenario may be selected in the 'Scenario Library' dialog and Merged, so that the scenario contents are merged into the scenarios branch of the Scenario Builder Tree.
	Opens the 'Add New Scenario' dialog box.

	Opens the ‘Add New Filter’ dialog box.
	Opens the ‘Add New Mapping’ dialog box.
	Opens the ‘Add New Rule’ dialog box.
	Opens the ‘Create New Rule Set’ dialog box (see description of ‘Rule Templates’).

The toolbar may be dragged and repositioned on the top, left or right side of the GUI or may be detached completely.

7.1.3 Scenario Builder Tree

Scenarios, filters, mappings and rules listed in the ‘Summary List’ may be dragged and dropped into position in the Scenario Builder Tree. The tree represents all scenarios, filters, mappings and rules that will be deployed into live use. When dropping an item into the tree, the following constraints apply:

- only scenarios can be dropped onto the tree root node, i.e. the ‘scenarios’ node
- a filter may be dropped under the ‘filters and mappings’ node
- a filter may be dropped under another filter
- a mapping may be dropped under a filter provided the filter has no other filter ‘children’ nodes underneath it.
- a rule may be dropped under the ‘rules’ node

When the tree is configured with a set of scenarios, filters, mappings and rules, it may be validated and subsequently deployed (providing it is valid).

Pop-up Menu Options

The following pop-up menu items are available by right-clicking a node in the Scenario Builder Tree:

All nodes:

fully expand / collapse	→	expands or collapses all descendent nodes below the selected node, provided there are descendents to expand or collapse.

All nodes except the root node:

move down	→	moves the selected node down one (provided it is possible to do so), but maintaining the same level of nesting.
move up	→	moves the selected node up one (provided it is possible to do so), but maintaining the same level of nesting.

The ‘scenarios’ root node:

un-highlight all	→	removes the red highlighting from any nodes highlighted in the tree (see the ‘highlight’ pop-up menu item available for the Summary List rows).
import from library	→	Opens the ‘Scenario Library’ dialog listing details of username, date, description and additional information for each scenario exported to the library. An exported scenario may be selected in the ‘Scenario Library’ dialog and Merged, so that the scenario contents are merged into the scenarios branch of the tree.

Scenario nodes:

export to library	→	exports the currently selected scenario and all its associated filters, mappings and rules to the scenario library. The user is prompted via a dialog for a description and additional information to be associated with the exported scenario.

Scenarios, filters, mappings and rules nodes:

Delete from tree	→	removes the selected item from the tree, but not from the Summary List.

7.1.4 Scenarios, Filters, Mappings and Rules Summary List

When a new scenario, filter, mapping or rule is first created, it appears as an item in the 'Scenarios, Filters, Mappings and Rules Summary List'. Thereafter, it may be viewed, modified, duplicated, highlighted in the Scenario Builder tree, or deleted. Any row in the Summary List may be dragged and dropped into the Scenario Builder tree, according to the constraints described above. The Summary List shows details of the item's type (scenario, filter, mapping or rule), description and modification date. The columns are re-sizable and movable and their headers may be clicked on to toggle the sort order.

Pop-up Menu Options

The following pop-up menu items are available by right-clicking a row in the Summary List:

view / modify	→	opens the appropriate dialog box for viewing or modifying the selected scenario, filter, mapping or rule.
create copy	→	makes a copy of the selected scenario, filter, mapping or rule. The new copy will have the same Description but preceded with 'copy of '.
highlight	→	highlights in red the selected scenario, filter, mapping or rule in the Summary List. Also all occurrences of the selected scenario, filter, mapping or rule are highlighted in red in the Scenario Builder tree. This is useful if the tree is very large and it is difficult to spot all nodes related to an item selected in the summary list.
un-highlight	→	un-highlights a previously highlighted scenario, filter, mapping or rule in the Summary List. Also all occurrences of the selected scenario, filter, mapping or rule are un-highlighted in the Scenario Builder tree.
delete	→	deletes the selected scenario, filter, mapping or rule from the Summary List. If the item has been copied to the Scenario Builder tree, all such occurrences will also be deleted. ***Note that once an item has been deleted in this way, it will have been be permanently removed. A safeguard would be to create a backup copy on the local disk of all scenarios, filters, mappings and rules by clicking on the toolbar button.***

Double-clicking with the left mouse button on a row in the Summary List has the same effect as selecting the 'view / modify' pop-up menu item.

7.1.5 Status Bar

The Status Bar displays informational and warning messages – these are shown in the left hand area. Warning messages are highlighted with a red background. The progress of various operations is shown in the progress bar area on the right hand side of the Status Bar.

7.2 The Mesh Viewer

The Mesh Viewer is used for:

- Viewing in real-time the state of the mesh objects.
- Viewing in real-time the notification details associated with the displayed mesh objects.
- Viewing the full hierarchy of mesh objects in the state mesh, in terms of a model tree of classes, subclasses, instances and instance details.
- Navigating around the model tree.

The screenshot below shows the Mesh Viewer with the main components areas labelled.

7.2.1 Menu Bar

The following menu items are available:

File → Inject alarms from file	Allows a user with 'tester' role privilege to select an XML file of alarms to inject into UCA.
File → Exit	Exits the application.
View → Look and Feel → …	Changes the look and feel of the GUI according to those supported on the client platform eg. CDE/Motif, Windows, Metal.
View → Pause	Pauses the update of the Mesh Object List. See the 'pause' icon description under the Toolbar section below.
View → Filter	Filters the objects displayed in the Mesh Object List. See the 'filter' icon description under the Toolbar section below.
Help → Mesh Viewer Help	Displays Mesh Viewer help information in a web page.
Help → HP web site	Displays the HP web page.
Help → About	Displays a dialog showing the UCA and Mesh Viewer version numbers.

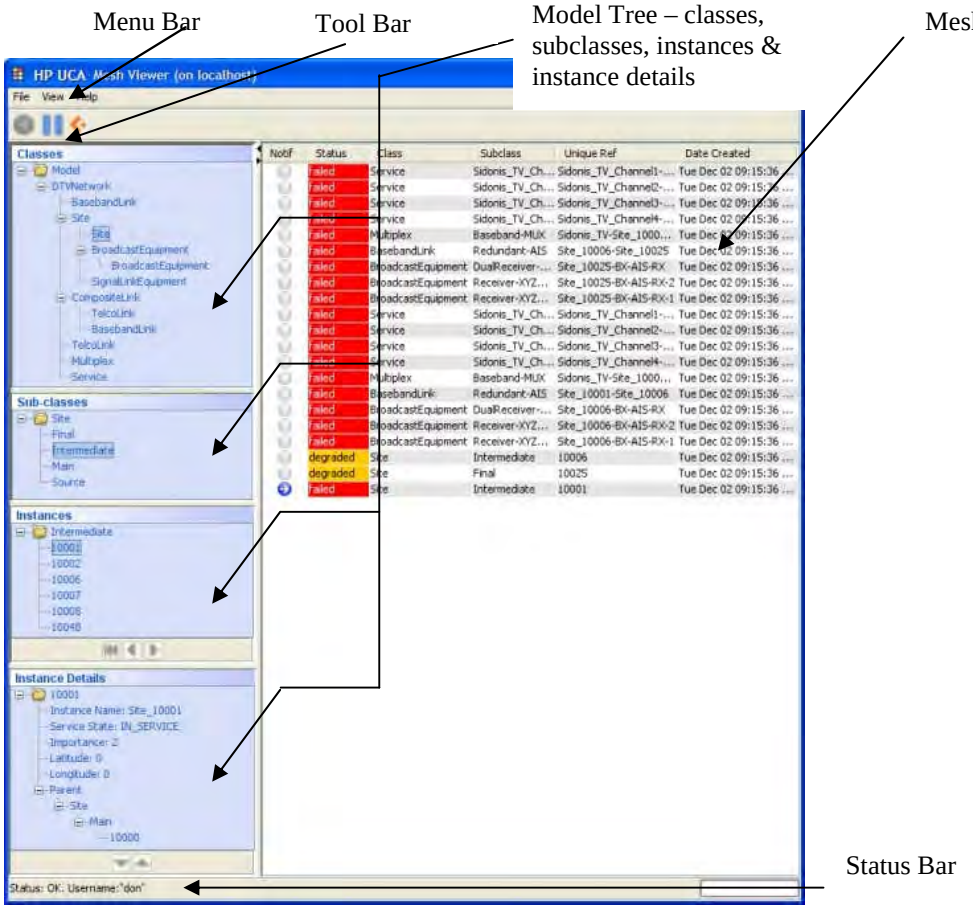


Figure 19 - The Mesh Viewer

7.2.2 Tool Bar

Clicking on an icon in the tool bar performs the action as follows:

Icon	Action
	Displays the Notifications Viewer Dialog (see below), showing notification details associated with the object currently selected in the associated Mesh Object List.
	Toggles the pausing / un-pausing of the Mesh Object List. When paused, updates to the

	Mesh Object List are received but not displayed. When un-paused, the Mesh Object List will work as normal i.e. the correct state of the failed or degraded Mesh Objects will be displayed dynamically. When the display is paused, the menu bar will change colour and the pause icon will change.
	Enables display filtering of failed or degraded Mesh Objects in the Mesh Object List. When selected, the tool bar will display text boxes to allow entry of the filtering conditions. Filtering may be performed on all columns or any individual column. The filtering criteria can include regular expressions, in which case the regular expression wizard can be used. When filtering is de-selected, the display will revert to normal un-filtered behaviour.

The toolbar may be dragged and repositioned on the top, left or right side of the GUI or may be detached completely.

7.2.3 Model Tree

The Model Tree is split into four re-sizable sections – classes, subclasses, instances and instance details. Each section displays a tree structure.
The classes tree is essentially the parent-child relationship information between the classes as described by the metamodel. When a class node is selected, the subclasses tree shows all the subclass types (as derived from the actual model data) related to that class.
When a subclass node is selected, the instances tree shows all the mesh object instances (as derived from the actual model data) related to that subclass. If there are a large number of instances, they are presented one 'page' at a time. The pages may be navigated one page forward, one page backward and back to the first page by selecting  ,  and  respectively from just below the instances tree.
When an instance node is selected, the instance details tree shows all the mesh object instance details (as derived from the actual model data) related to that instance. The instance's details include not just information about specific attributes, such as importance, latitude, longitude etc., but model relationship data. For example, there will be a tree branch showing the Parent details in terms of parent class, parent subclass and parent instance. There may also be a branch showing similar 'relative' or 'associate' details, depending on whether the instance has relatives or associates defined in the metamodel and data has been provided for them in the model database.
If a parent, relative or associate instance node is selected in this tree and the  button is clicked, the Model Tree will change to display the class, subclass, instance and instance details associated with that node. Subsequently, if the  button is clicked, the Model Tree will revert to the object that was previously navigated from (i.e. the one that was navigated from using the  button).

Pop-up Menu Options

The following pop-up menu item is available by right-clicking a node in the subclasses tree:

find instances … → opens the Search dialog, as shown below. This dialog is used to specify an instance name (or names) to search for. An exact instance name or a wild-carded expression may be entered as the search criteria. When the OK button is clicked, the instances tree will show those instances related to the currently selected subclass, according to the search value entered.

Figure 20 - The Search for Instances dialog

After displaying 'searched for instances', to reset the list of displayed instances to the full set, CTRL-click the subclass node to deselect it, then re-select it with a left mouse-click.

The following pop-up menu items are available by right-clicking a node in the instances tree:

show all notifications … → Displays the Notifications Viewer Dialog (see below), showing notification details associated with the object whose class, subclass and instance is currently selected in the associated trees.

Create alarm → Displays the Create Alarm Dialog (see below), allowing a user with 'tester' role privilege to enter all alarm fields for an alarm to be injected into UCA.

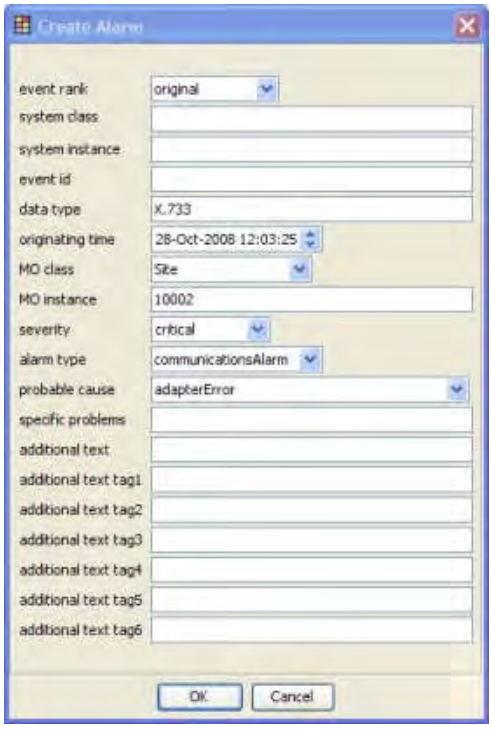


Figure 21 - The Create Alarm dialog

7.2.4 Mesh Object List

When a mesh object changes state to failed or degraded, the Mesh Object List will update in real-time to display details of that object, including its status (failed or degraded), class name, subclass name and the instance's unique reference, as well as the timestamp that the GUI received the update. If an object represented in the Mesh Object List is associated with one or more notification, the leftmost column will show either the icon or the icon, depending on whether the associated notification(s) are normal or locked. When a mesh object changes state from failed or degraded back to normal, the corresponding row will be removed from the Mesh Object List.

When an item in the list is selected, its corresponding class, subclass, instance and instance details are highlighted in the Model Tree.

The columns in the Mesh Object List are re-sizable and movable and their headers may be clicked on to toggle the sort order.

Pop-up Menu Options

The following pop-up menu item is available by right-clicking a row in the Mesh Object List:

highlight object in model	→	highlights the class, subclass, instance and instance details in the Model Tree associated with the object in the selected row
show all notifications …	→	Displays the Notifications Viewer Dialog (see below), showing notification details associated with the object whose class, subclass and instance is currently selected in the associated trees.

Double-clicking with the left mouse button on a row in the Mesh Object List has the same effect as selecting the 'show all notifications …' pop-up menu item.

7.2.5 Notifications Viewer Dialog

The notifications viewer dialog provides useful dynamic information about notification(s) and data related to those notifications. It is used for:

- Viewing current notifications in real-time. A notification is an indication of the problem detected and is the result of an action being fired from a rule.
- Viewing details of contributory events associated with a notification. A contributory event is an event that contributed to the problem i.e. it is an event that is wholly or partially indicative of the problem.
- Viewing details of the affected objects associated with a notification. An affected object represents a mesh object within the model that has been affected as a by-product of the problem e.g. downstream sites affected by a main site failure.
- Viewing details of sympathetic events associated with an affected object. A sympathetic event represents an event that has occurred as a by-product of the problem e.g. an event from a downstream site that was generated as a result of a main site failure.

To view the notification(s) associated with a failed or degraded mesh object, double click a row in the Mesh Object List (or right click the row and select 'show all notifications …'.
To view the notification(s) associated with an object in the Model tree, select the desired class, subclass and instance nodes, then right-click the instance node and select 'show all notifications' from the pop-up menu.

The screenshot below shows the Notifications Viewer Dialog with the main components areas labelled.

From the Notifications Viewer Dialog, the following operations may be performed:

select a notification in the Notifications Table	→	this will display all contributory events and affected objects associated with the notification.
select an affected object in the Affected Objects Table	→	this will display all sympathetic events associated with the affected object.

The columns in the Notifications Viewer Dialog tables are re-sizable and movable and their headers may be clicked on to toggle the sort order.

Note that all information presented in the Notifications Viewer Dialog is potentially available to be passed on to an external system, for example in the form of a 'master' problem alarm together with the event details that might be used to de-clutter an alarm display in a network management system.

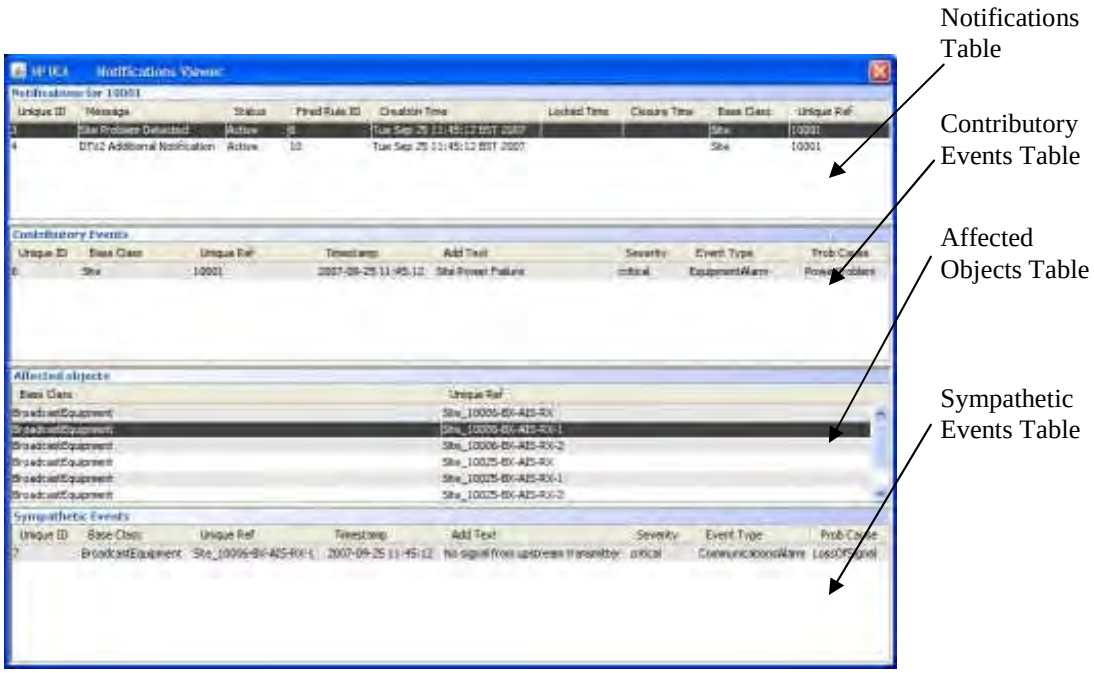


Figure 22 - The Notifications Viewer Dialog

The Notifications Viewer Dialog updates dynamically with any changes to the notification details.

Note that only a single Notifications Viewer Dialog can be displayed at any one time. If the dialog is invoked for a different object, then any currently displayed Notifications Viewer Dialog will be replaced with the new one.

7.2.6 Status Bar

The Status Bar displays informational and warning messages – these are shown in the left hand area. Warning messages are highlighted with a red background. The progress of various operations is shown in the progress bar area on the right hand side of the Status Bar.

Chapter 8 Creating Scenarios, Filters, Mappings and Rules

8.1 Scenarios

Scenarios provide a container for a set of filters, mappings and rules. A scenario typically represents a set of filters, mappings and rules that are a logical, self-contained grouping e.g. a scenario might relate to handling power failures, for dealing with SDH correlations or simply for housekeeping purposes. One of the key attributes of a scenario is its 'context name'. A 'context name' essentially relates to a 'working memory' within the inference engine component of UCA. Being able to have separate working memories is very useful to demarcate groups of rules that must be kept independent of each other. Any number of scenarios may be created and each one may have a different context name if desired; alternatively, they may all have the same context name, or there may be some sharing the same context and others with different ones. The idea of a context name (i.e. essentially a working memory) therefore allows potentially conflicting logical correlations to execute in isolation, if required, or to co-exist in the same context. Furthermore, 'Notifications' provide a user-defined and controllable communications path between contexts, allowing hierarchies of correlations to be constructed.
To create a new scenario:

- Click on the button in the UCA Scenario Manager toolbar or select File → New → Scenario from the menu-bar.
- In the 'Add New Scenario' dialog, enter a description, some additional information and a context name.
- Click on the **OK** button.

Figure 23 - The Add New Scenario Dialog

The new scenario will be listed in the Scenarios, Filters, Mappings and Rules Summary List in the UCA Scenario Manager.

To view an existing scenario:

- Double-click the scenario in the Scenarios, Filters, Mappings and Rules Summary List in the UCA Scenario Manager, or right-click the scenario and select the **view / modify** pop-up menu item.

To modify an existing scenario:

- Double-click the scenario in Scenarios, Filters, Mappings and Rules Summary List in the UCA Scenario Manager, or right-click the scenario and select the **view / modify** pop-up menu item.
- Make the necessary changes and click OK.

To include a scenario in a deployment:

- If the scenario is to be included in the set of scenarios, filters, mappings and rules for an active deployment, it must be dragged from the Scenarios, Filters, Mappings and Rules Summary List

and dropped onto the root node (i.e. the ‘scenarios’ node) of the Scenario Builder Tree. Once this has been done, the new scenario will be shown in the tree with two automatically created sub-nodes – ‘filters and mappings’ and ‘rules’, as shown in the following example.

To remove a scenario from a deployment:

- If the scenario is to be removed from the set of scenarios, filters, mappings and rules for an active deployment, right-click the scenario in the Scenario Builder Tree and select **delete from tree** in the pop-up menu. ***Note that when this is done, all children nodes underneath the removed node will also disappear from the tree.***

8.2 Filters

UCA supports a powerful and highly configurable alarm filtering capability. Alarms may be allowed to pass into the system based on filter conditions applied to any combination of any event fields (see section 10.3.2 for the available event fields). The filter conditions include the operators: ‘equals’, ‘not equals’, ‘contains’, ‘does not contain’, ‘starts with’, ‘ends with’ and ‘matches’.
The ‘matches’ filter condition operator allows use of a regular expression. In addition to entering an expression directly, a graphical ‘regular expression wizard’ is provided that allows a user to create regular expression statements without needing any knowledge of regular expression syntax.
Filter conditions are grouped according to conditional logic, including:

- All conditions being satisfied
- Any conditions being satisfied
- Any conditions not being satisfied
- None of the conditions being satisfied

Any of these logic groups may be contained in any other logic group. In this way it is possible to effectively create arbitrarily complex logic expressions.

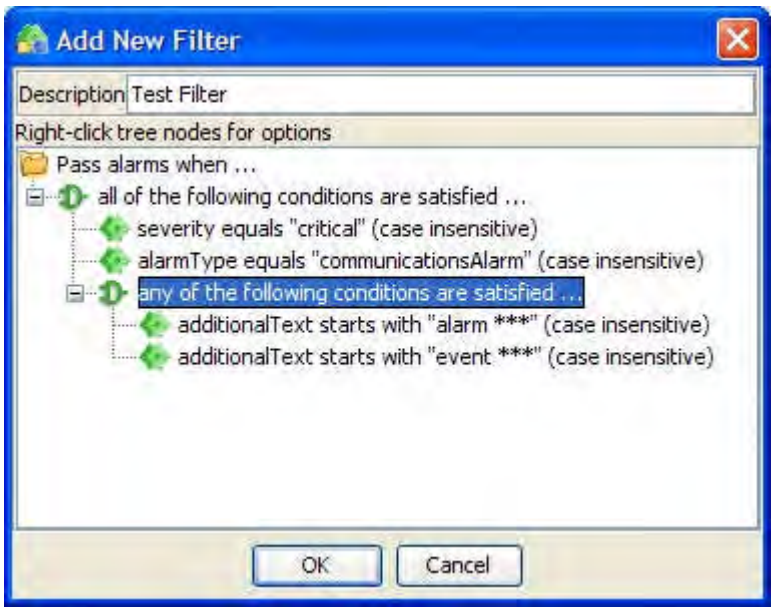
To create a new filter:

- Click on the  button in the UCA Scenario Manager toolbar or select File → New → Filter from the menu-bar.
- In the ‘Add New Filter’ dialog, enter a description.
- In the ‘Add New Filter’ dialog, right-click the tree root node (‘Pass alarms when …’) and select the required logic group from the ‘condition ►’ sub-menu.
- Right-click the logic group that will have been added to the tree and select either ‘insert new filter condition’ or ‘condition ►‘ from the pop-up menu.
- If ‘insert new filter condition’ was selected, select the required field and operator values from the drop down lists and enter (or select from a drop-down list) the value, as shown in the example screenshot below.

Figure 24 - The Add New Filter Dialog

- If 'condition ►' was selected, select the required logic group sub-menu item.
- Continue to build new filter statements and logic groups in this manner as necessary.
- To modify or delete a filter statement or logic group, right-click on the associated tree node item and select 'modify' or 'delete' as appropriate.

The example screenshot below shows a reasonably complex filter that will allow events into the system provided the severity is 'critical' and the alarmType is 'communicationsAlarm' and the additionalText either starts with 'alarm ***' or it starts with 'event ***'.



- Finally, to complete the filter definition, click on the OK button.

The new filter will now be listed in the Scenarios, Filters, Mappings and Rules Summary List in the UCA Scenario Manager.

To view an existing filter:

- Double-click the filter in the Scenarios, Filters, Mappings and Rules Summary List in the UCA Scenario Manager, or right-click the filter and select the **view / modify** pop-up menu item.

To modify an existing filter:

- Double-click the filter in the Scenarios, Filters, Mappings and Rules Summary List in the UCA Scenario Manager, or right-click the filter and select the **view / modify** pop-up menu item.
- Make the necessary changes and click **OK**.

To include a filter in a deployment:

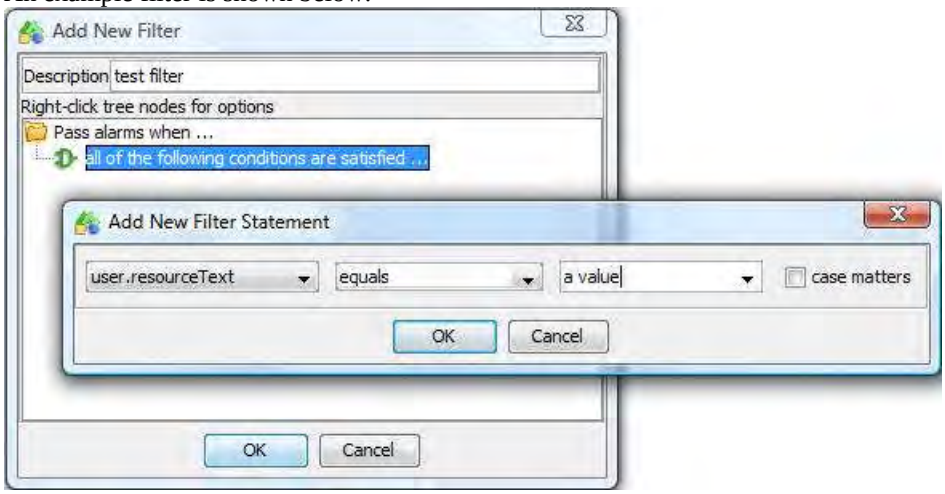
- If the filter is to be included in the set of scenarios, filters, mappings and rules for an active deployment, it must be dragged from the Scenarios, Filters, Mappings and Rules Summary List and dropped onto either the 'filters and mappings' node, or underneath an existing filter in the Scenario Builder Tree. Once this has been done, the new filter will be shown in the tree. The example below shows two filters, one below the other.

To remove a filter from a deployment:

- If the filter is to be removed from the set of scenarios, filters, mappings and rules for an active deployment, right-click the filter in the Scenario Builder Tree and select 'delete from tree' in the pop-up menu. ***Note that when this is done, all children nodes underneath the removed node will also disappear from the tree.***

8.2.1 Using user-Defined event fields in a filter

It is possible to include user-defined event fields in filter conditions. User-defined fields are found on the drop-down, listed after the default event fields.
An example filter is shown below:



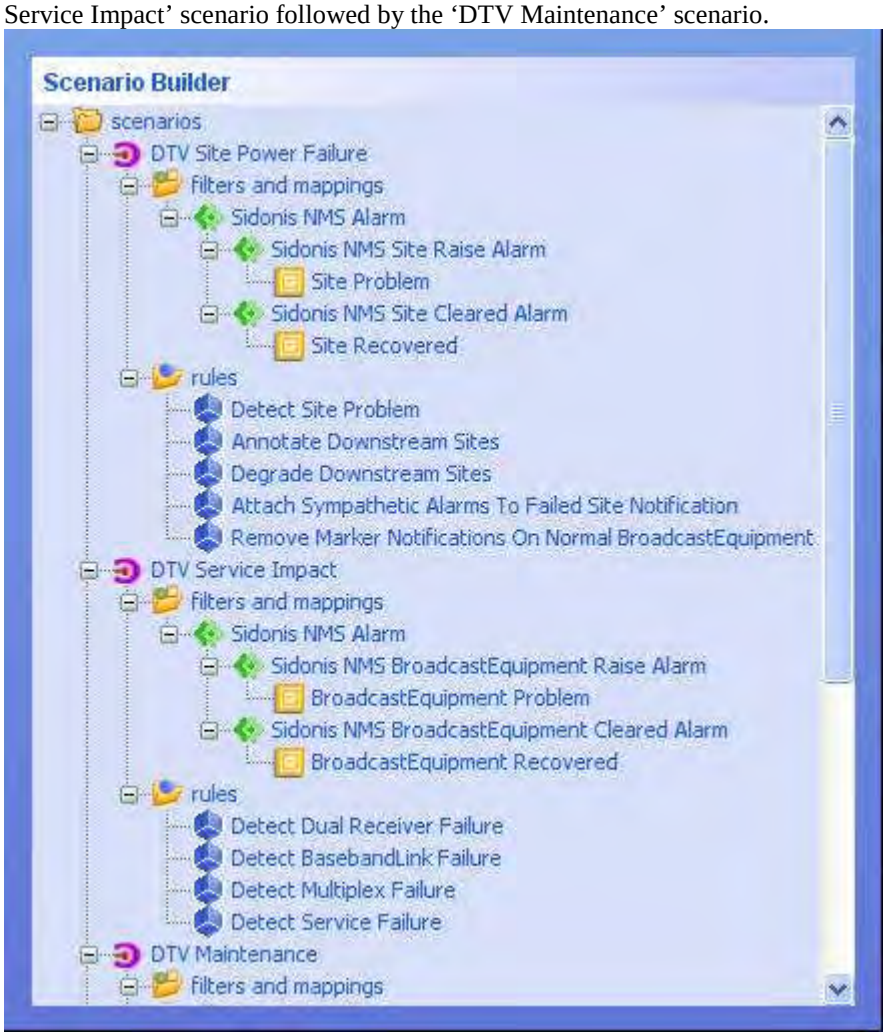
8.2.2 Arranging Filters

When dragging a filter to the Scenario Builder Tree, it may be placed underneath the 'filter and mappings' node or underneath an existing filter. A filter at the same 'level' as another filter is its 'sibling'; a filter below another filter is its 'child'. For example, in the screenshot below, filter2 is a child of filter 1; filter4 is a sibling of filter2; filter3 is a sibling of filter1.
When an event is being tested against the filters in the Scenario Builder Tree, the following order of processing takes place:

- The event is tested against the first filter of the first scenario.

- If the event passes the filter then the next child filter will be tested against. If there is no child filter, then a mapping must have been reached (see below).
- If an event fails to pass a filter, then the next sibling filter is examined. If there is no sibling filter, then the whole process is repeated for the next scenario, if there is one.
- If a mapping is reached then the event is allowed into the system ready to be mapped and the whole filtering process repeated for the next scenario, if there is one.
- The entire process ends when a mapping is reached or there are no more sibling filters to test against.

For example, as shown in the screenshot below, an incoming event would first be tested against the 'Sidonis NMS Alarm' filter. If the event passed the filter, it would be tested against the 'Sidonis NMS Site Raise Alarm' filter. If the event passed this filter it would be mapped using the 'Site Problem' mapping, otherwise it would be tested against the 'Sidonis NMS Site Cleared Alarm'. The whole process would then be repeated for the 'DTV Service Impact' scenario followed by the 'DTV Maintenance' scenario.



8.2.3 Using the Regular Expression Wizard with Filters

When adding a new filter statement during the filter definition process described above, some fields allow the 'matches' operator to be selected from the drop-down list. If 'matches' is selected, a regular expression value may be entered in the value field. Alternatively, the 'Wizard >>>' button may be selected, in which case the Regular Expression Wizard will be started. This wizard allows a user to automatically generate a regular expression without the need to know any regular expression syntax.
When the Regular Expression Wizard starts, the first page allows the user to define some sample text to apply the regular expression to and the second page is for defining the match conditions and viewing their effect on the sample text.
For example, suppose the additonalText field of an alarm contained the text

```
WO BATH/00X/00/XYZ123 AT-6 TIME 070202 1230 PAGE 1
*** ALARM 855 O1/APT "BATH/00X/0"U 070202 1230
DIGITAL PATH QUALITY SUPERVISION
SF
DIP DIPPART SFL QSV
BEURS 1 1 181
END
```

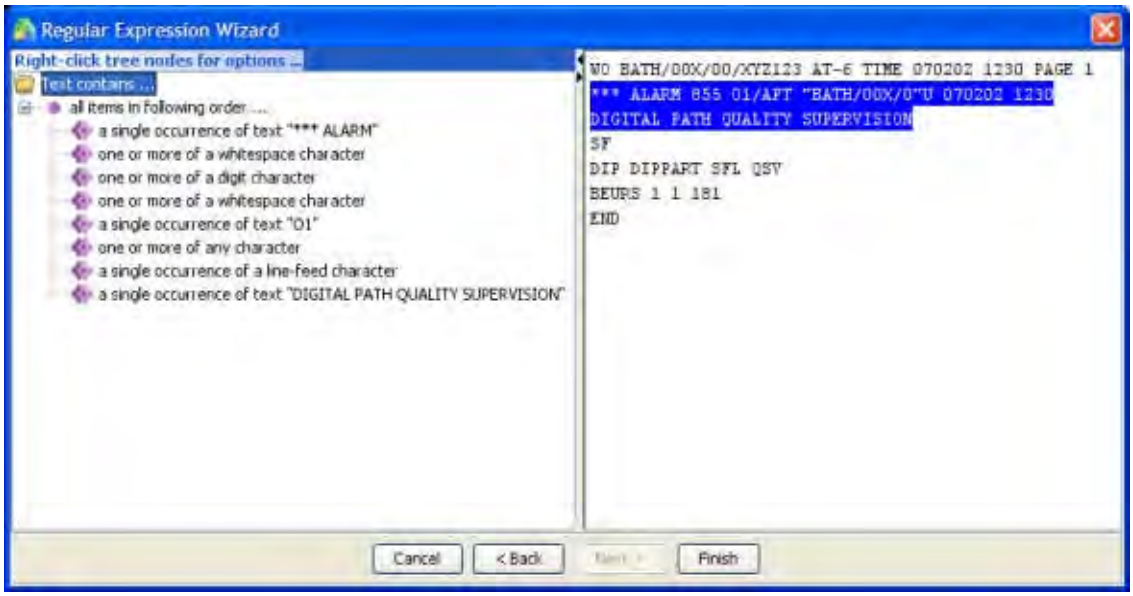
and you wish to filter alarms using a regular expression looking for the particular pattern of text:

"*** ALARM" followed by
one or more spaces followed by
one or more digit characters followed by
one or more spaces followed by
the text "O1" followed by
any text, excluding a line terminator followed by
the text "DIGITAL PATH QUALITY SUPERVISION"

then you would use the Wizard as follows:

- Enter the above text into the area on the right size of the window. The text may be typed into the text area. It may also be pasted from the current copy/paste buffer or read in from a local file, using the buttons on the left.

- The next stage is to define the match conditions. Clicking the **Next** button will display the screen that allows the conditions to be specified and their effect to be displayed, as shown below.
- To define the match conditions, right-click the 'Text contains …' root node in the tree on the left side and select the desired 'condition ►' submenu item, i.e. either 'all items in the following order …' or 'any of the following items …'. In this case, select 'all items in the following order …'.
- The tree node 'all items in the following order …' will be inserted under the root node. Right-click this node and select **insert new expression**.
- In the 'Add New Expression' dialog, select the appropriate drop-down menu items.
- Repeat this procedure using the right-click menu items to add, modify or delete nodes until all the expressions have been specified. Remember that match conditions may be nested under each other (similar to the logic expressions for Filters), if desired. As each expression is entered in the tree, the sample text on the right will be highlighted in blue to reflect the current matching. The screenshot below shows the whole tree of expressions for the example pattern of text.



- Finally, click the **Finish** button and the actual regular expression will be automatically generated and inserted into the filter statement value field. For the example, the regular expression would be:

```
\*\*\* ALARM\s+\d+\s+01.+\nDIGITAL PATH QUALITY SUPERVISION
```

For more details on the advanced use of regular expressions, see the Java documentation for the Pattern class at http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html

Advanced Options

There are two advanced options that may be set for the entire set of expressions in the tree. To select an advanced option, right-click the 'text contains …' tree root node and select the desired 'advanced options ▶' submenu item.

The effect of each advanced options menu item is:

enable multiline mode	→	By default, the expressions `"the beginning of the text"` and `"the end of the text"` ignore line terminators and only match at the beginning and the end, respectively, of the entire text sequence. If 'enable multiline mode' is set then `"the beginning of the text"` matches at the beginning of text and after any line terminator (except at the end of the text). When in multiline mode `"the end of the text"` matches just before a line terminator (and the end of the input text).
case insensitive (Unicode)	→	Enables Unicode-aware case-insensitive matching.

8.3 Mappings

Once an alarm has passed though the filter(s), it must be mapped. The purpose of mapping is threefold:

- Objects in the system are identified by their unique reference field. During data-load, all objects get stored within the UCA model database with their unique reference filled in. One of the functions of mapping is to relate the object the incoming event refers to with a corresponding Mesh Object. In the simplest case, there might be a one-to-one mapping of an event field with the corresponding object's unique reference. However, the situation may be far more complicated, involving extracting parts of many of the event fields and combining them to form a corresponding identifier to match to a unique reference. So the primary purpose of mapping is to extract a value

from the event that represents the unique reference of an object. UCA supports very flexible mapping of unique references from events. A unique reference may be mapped from an event directly from one of the event's fields, or it may be mapped from multiple parts of one or more fields, combining those parts in any order and with any prefix or suffix.

- It is not enough to just map the event's unique reference. The event also needs to be mapped to an appropriate class. The classes that an event can be mapped to are essentially those defined in the metamodel.
- Finally, the event must also be mapped to a status – normal, degraded or failed. Typically, a filter that passes a non-clear severity event will be followed by a mapping that maps to a status of failed or degraded; similarly, a filter that passes a clear severity event will be followed by a mapping that maps to a status of normal.

The result of mapping is to affect a corresponding Mesh Object, as described in sections 3.7 and 3.8.

UCA supports incoming events formatted as XML messages with a number of tags, each of which represents an event field – the following shows an example event (further details of the event format are provided in Chapter 10):

```
<Event>
      <eventRank>original</eventRank>
      <systemClass>sidonis_nms</systemClass>
      <systemInstance>V5</systemInstance>
      <eventId>1003</eventId>
      <dataType>X.733</dataType>
      <originatingTime>2005-06-10 12:16:32</originatingTime>
      <moClass>Site</moClass>
      <moInstance>10001</moInstance>
      <severity>critical</severity>
      <alarmType>EquipmentAlarm</alarmType>
      <probableCause>PowerProblem</probableCause>
      <additionalText>Site Power Failure</additionalText>
</Event>
```

An important field is the eventId, which uniquely defines the particular event. Typically, a non-clear severity event will be received with a particular eventId and the event will be mapped to failed or degraded status and to a particular base class and with its unique reference mapped from one or more fields. Subsequently, a clear severity event will be received with the same eventId as the original non-clear severity event and this will be mapped to normal status and the same base class and unique reference as the associated non-clear severity event. However, there are two special cases to be aware of:

1. If the eventId, mapped base class and mapped unique reference of a clear event do not match with a previously stored non-clear event, an alternative method is used to determine which Mesh Objects are effected:

   - In this case the Mesh Object(s) with the same Alarm Type, Probable Cause, Specific Problems and Additional Text will be cleared (i.e. their status set to 'normal').

2. If it is not possible for an external system to supply the clear event with enough information to allow the unique reference to be mapped, then the external system must send an 'event state change' message instead (see 10.3.3). This message contains a subset of the standard event fields, but it adds the 'updateState' field to indicate that this message essentially updates a previous one. An example of such as message is as follows:

   ```
   <Event>
        <eventRank>original</eventRank>
        <systemClass>sidonis_nms </systemClass>
        <systemInstance>V5</systemInstance>
        <eventId>1003</eventId>
        <dataType>X.733</dataType>
        <originatingTime>2005-06-10 12:16:34</originatingTime>
        <updateState>terminated</updateState>
   </Event>
   ```

   But the question remains, for such a message how would you specify the mapping to a Base Class and unique reference? The answer is that in the 'Add New Mapping' dialog (see below), the 'Lookup Unique Reference by matching Event Id' tick-box is selected. This

forces the system to lookup a raise event in the UCA event database with the same eventId and it uses that event's Base Class and unique reference.

To create a new mapping:

- Click on the button in the UCA Scenario Manager toolbar or select File → New → Mapping from the menu-bar.
- In the 'Add New Mapping' dialog, enter a description.
- Select the appropriate class to map to from the 'Map to class' drop-down list.
- Select the appropriate status to map to from the 'Map to status' drop-down list.
- If the message being mapped is an 'event state change' message (see description above), select the 'Lookup Unique Reference by matching Event Id' tick-box, otherwise the unique reference mapping details must be supplied.
- Finally, to complete the mapping definition, click on the **OK** button.

Mapping the Unique Reference

- In the 'Add New Mapping' dialog, right-click the tree root node ('Map UniqueReference using items in following order …') and select 'add new mapping statement'. The 'Add New Mapping Statement' dialog will be opened.

- In the drop-down list, select the desired event field to map from.
- If the selected field's contents are to be mapped in their entirety into the unique reference, select the 'directly' tick-box and click the **OK** button.
- Otherwise, click the **Wizard** >> button in order to create expressions that define the match and extraction criteria. Advanced users may select the 'advanced' tick-box and enter these criteria into the 'Match Expression' and 'Token Expression' boxes directly without using the wizard. See below for details of how to use the regular expression Wizard for mappings. Click the **OK** button.
- The new mapping statement will now be displayed under the root of the tree in the 'Add New Mapping' dialog, as in the following screenshot:

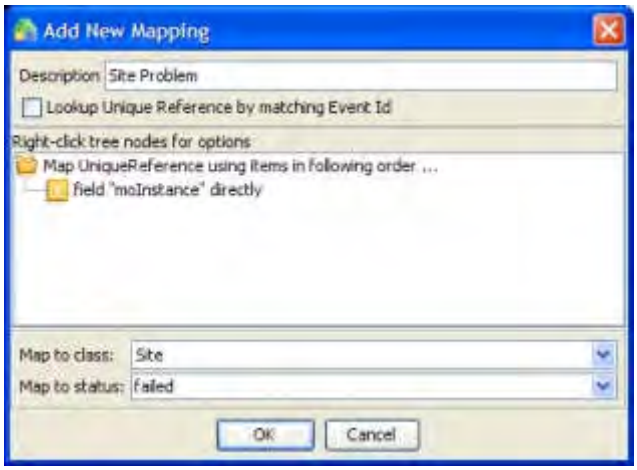
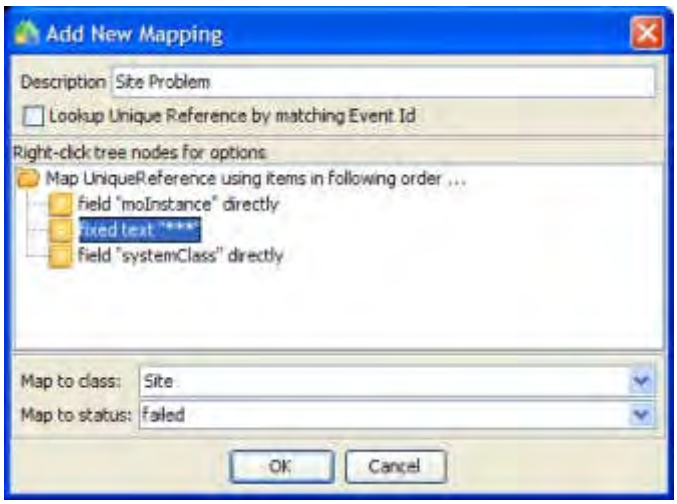


Figure 25 - The Add New Mapping Dialog

- If text needs to be extracted from a number of event fields in order to define the unique reference, then continue the process of adding new mapping statements. The value of the extracted unique reference will be the concatenation of the mapping statements. If fixed text delimiters need to be placed between any mapping statements, then right-click then tree root node and select 'add fixed text' and supply the desired text. For example, in the following mapping, if the event's moInstance field was "`10001`" and the systemClass was "`sidonis_nms`", then the mapped unique reference would be "`10001***sidonis_nms`".



- Mapping statements in the tree may be moved up or down, modified or deleted by right-clicking the node and selecting the appropriate pop-up menu item.

The new mapping will now be listed in the Scenarios, Filters, Mappings and Rules Summary List in the UCA Scenario Manager.

To view an existing mapping:

- Double-click the mapping in the Scenarios, Filters, Mappings and Rules Summary List in the UCA Scenario Manager, or right-click the mapping and select the **view / modify** pop-up menu item.

To modify an existing mapping:

- Double-click the mapping in the Scenarios, Filters, Mappings and Rules Summary List in the UCA Scenario Manager, or right-click the mapping and select the **view / modify** pop-up menu item.
- Make the necessary changes and click **OK**.

To include a mapping in a deployment:

- If the mapping is to be included in the set of scenarios, filters, mappings and rules for an active deployment, it must be dragged from the Scenarios, Filters, Mappings and Rules Summary List and dropped underneath an existing filter in the Scenario Builder Tree. Once this has been done, the new mapping will be shown in the tree. Note that multiple mappings may be dropped underneath the same filter.

To remove a mapping from a deployment:

- If the mapping is to be removed from the set of scenarios, filters, mappings and rules for an active deployment, right-click the mapping in the Scenario Builder Tree and select **delete from tree** in the pop-up menu.

8.3.1 Using the Regular Expression Wizard with Mappings

When adding a new mapping statement during the mapping definition process described above, the 'Wizard >>>' button may be selected, in which case the Regular Expression Wizard will be started. This wizard allows a user to:

- automatically generate a regular expression, without the need to know any regular expression syntax, that is used to match text against
- automatically construct a 'token expression' that determines how multiple matched items are joined together to form a complete piece of text

When the Regular Expression Wizard starts, the first page allows the user to define some sample text to apply the regular expression to. The second page is for defining the match conditions, viewing their effect on the sample text and defining which pieces of matched text should be extracted to form the unique reference. The third page is used to re-order the extracted items, if required, and set any desired fixed text prefixes or suffixes between the items.

As an example, suppose the additonalText field of an alarm contained the text

```
WO BATH/00X/00/XYZ123 AT-6 TIME 070202 1230 PAGE 1
*** ALARM 855 O1/APT "BATH/00X/0"U 070202 1230
DIGITAL PATH QUALITY SUPERVISION
SF
DIP DIPPART SFL QSV
BEURS 1 1 181
END
```

and you wish to map the unique reference so that is formed by trying to match the text highlighted in blue below:

```
WO BATH/00X/00/XYZ123 AT-6 TIME 070202 1230 PAGE 1
*** ALARM 855 O1/APT "BATH/00X/0"U 070202 1230
DIGITAL PATH QUALITY SUPERVISION
SF
DIP DIPPART SFL QSV
BEURS 1 1 181
END
```

and the actual text you wish to extract for the unique reference is as highlighted in red below:

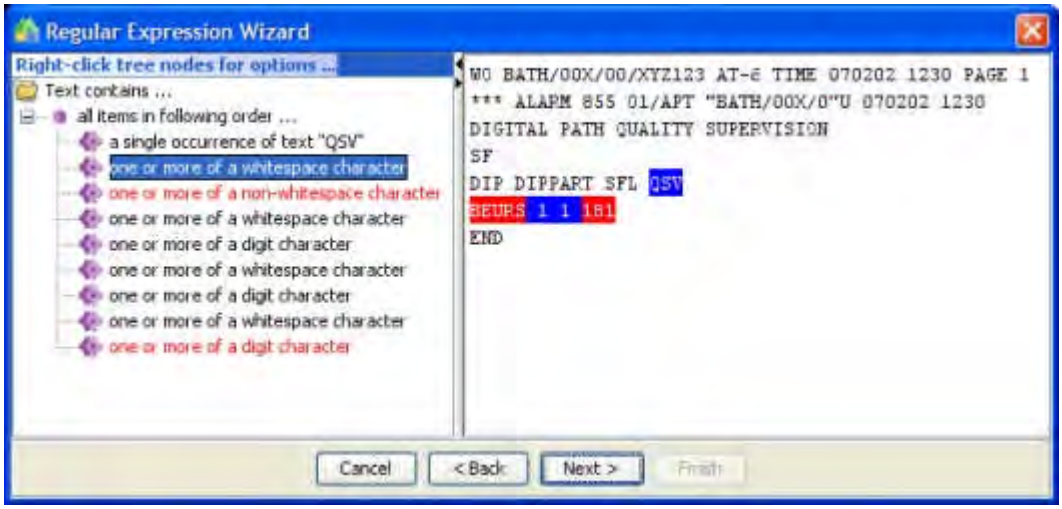
```
WO BATH/00X/00/XYZ123 AT-6 TIME 070202 1230 PAGE 1
*** ALARM 855 O1/APT "BATH/00X/0"U 070202 1230
DIGITAL PATH QUALITY SUPERVISION
SF
DIP DIPPART SFL QSV
BEURS 1 1 181
END
```

Furthermore, the text you wish to extract is not simply to be `"BEURS181",` but it should be `"181"` followed `"BEURS"`, and with `"BEURS"` prefixed with `"---"`. i.e. the mapped unique reference from the example would end up being `"181--BEURS181".`

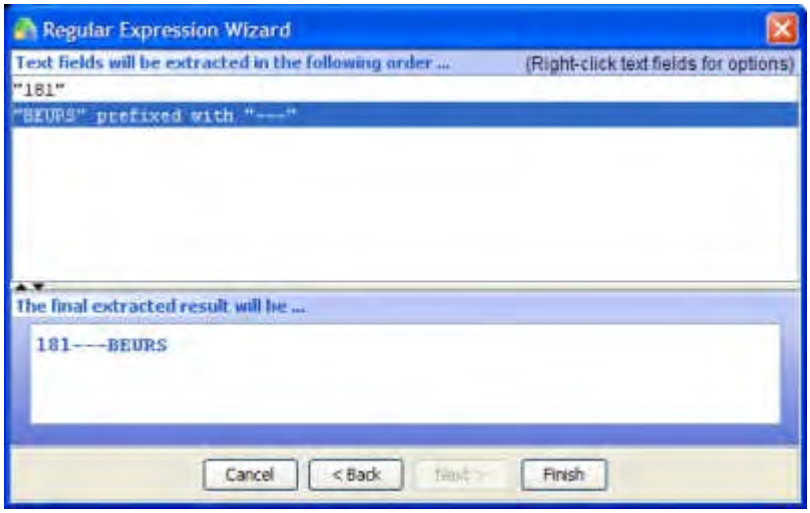
Then you would use the Wizard as follows:

- Enter the sample alarm text into the area on the right size of the window. The text may be typed into the text area. It may also be pasted from the current copy/paste buffer or read in from a local file, using the buttons on the left.
- Click the 'Next' button to display the page that allows the match and extraction conditions to be specified.
- To define the match conditions, right-click the 'Text contains …' root node in the tree on the left side and select the desired 'condition ►' submenu item, i.e. either 'all items in the following order …' or 'any of the following items …'. In this case, select 'all items in the following order …'.
- The tree node 'all items in the following order …' will be inserted under the root node. Right-click this node and select 'insert new expression'.
- In the 'Add New Expression' dialog, select the appropriate drop-down menu items.
- Repeat this procedure using the right-click menu items to add, modify, move up/down or delete nodes until all the expressions have been specified. Remember that match conditions may be nested under each other (similar to the logic expressions for Filters), if desired. As each expression is entered in the tree, the sample text on the right will be highlighted in blue to reflect the current matching.
- Next, you must identify which expressions relate to the text items you wish to extract. For example the tree node item 'one or more of a non-whitespace character' relates to the text `"BEURS"` and the final tree node 'one or more of a digit character' relates to the digits `"181"`. To identify the parts to be extracted, right-click the associated tree node and select 'extract' from the pop-up menu item.

When this is done, the associated sample text will be highlighted in red, as shown in the screenshot below:



- Click the **Next** button to display the page that allows you to re-order the extracted items, if required, and set any fixed text prefixes or suffixes.
- In the top half of the page, right-click `"181"` and select the 'move up' pop-up menu item. The bottom half of the window shows exactly what the final result of the whole matching and extraction would be.
- In the top half of the page, right-click `"BEURS"` and select the 'set prefix' pop-up menu item and enter `"---"` in the dialog. Again, the bottom half of the window shows exactly what the final result will be, in this case `"181---BEURS"`.



- Finally, click the **Finish** button and the actual match regular expression and 'token' regular expression will be automatically generated and inserted into the mapping 'Match Expression' and 'Token Expression' fields. For the example, these would be:

  `QSV\s+(\S+)\s+\d+\s+\d+\s+(\d+)`

  and

  `$2---$1`

For advanced users who wish to specify the 'Match Expression' and 'Token Expression' fields without using the wizard, the 'Match Expression' is simply the regular expression, with match groups enclosed in round brackets. The 'Token Expression' defines the match groups in order of extraction as $1, $2, $3 etc. and orders these groups as appropriate, with any required fixed text prefixes or suffixes.

For more details on the advanced use of regular expressions, see the Java documentation for the Pattern class at http://java.sun.com/javase/6/docs/api/java/util/regex/Pattern.html

Advanced Options

There are two advanced options that may be set for the entire set of expressions in the tree. To select an advanced option, right-click the 'text contains …' tree root node and select the desired 'advanced options ►' submenu item. The advanced options are the same as those when using the regular expression wizard for filters – see for 8.2.3 details.

8.4 Rules

Rules are central to the whole operation of UCA. Once events have passed through the filters and the mappings have been performed, the UCA rules engine operates on the basis of consequent state changes to mesh objects. There are four aspects to consider when defining a rule using the UCA Scenario Manager:

- The trigger conditions – these consist of rule statements that specify the conditions under which the trigger actions will be performed.
- The trigger actions – these are the actions (e.g. raise a root cause alarm) that are performed when the rule triggers i.e. the trigger conditions are satisfied.
- The teardown conditions – these consist or rule statements that specify the conditions under which the teardown actions will be performed.
- The teardown actions – these are the actions (e.g. clear a root cause alarm) that are performed when the rule tears down i.e. the teardown conditions are satisfied.

This section provides a basic overview of how to create rules and actions, whereas Chapter 9 provides extensive details, supplemented with examples and many screenshots of how to configure them.

To create a new rule:

- Click on the  button in the UCA Scenario Manager toolbar or select File → New → Rule from the menu-bar.
- In the 'Add New Rule' dialog, enter a description and a priority. Priority may be from 0 to 100, with 0 being lowest priority and 100 highest, and represents the order in which satisfied rules are processed by the rules engine.
- In the Trigger Conditions tab, right-click the tree root node and select the 'insert object existence condition' pop-up menu item.
- In the 'Add New Rule Object Condition' dialog, select the object type (e.g. 'a Notification', 'an Associate Group' etc.) and condition ('exists' or 'does not exist') from the drop down menus and click on OK.
- The new 'object existence condition' will be automatically added under the tree root node.
- Right-click the 'object existence condition' that was added to the tree and select 'insert attribute conditions' from the pop-up menu.
- In the 'Add New Rule Attribute Condition' dialog, select the appropriate items from the drop-down lists (or enter the values, depending on the attribute and condition selected), as appropriate for the rule trigger condition.
- Click on the **OK** button to add the new rule attribute condition.
- Continue adding new rule attribute conditions as above.
- Rule attribute conditions may be modified, deleted, moved up or moved down by right-clicking the associated tree node and selecting the appropriate pop-up menu item.
- Continue adding new 'object existence conditions' together with their associated 'rule attribute conditions', as above. An example set of rule trigger conditions is shown in the screenshot below.
- 'Object existence conditions' in the tree may be deleted, moved up or moved down by right-clicking the associated tree node and selecting the appropriate pop-up menu item. ***Note that deleting an' object existence condition' from the tree will also delete all its child nodes, i.e. all its associated 'rule attribute conditions'.***

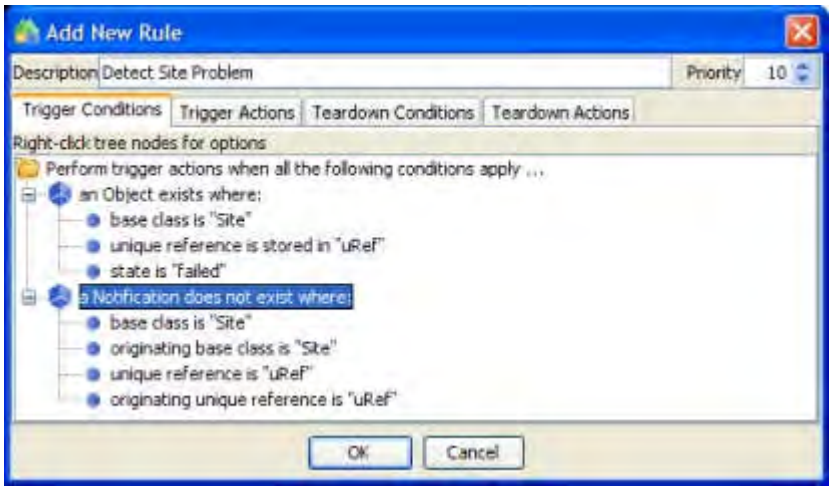
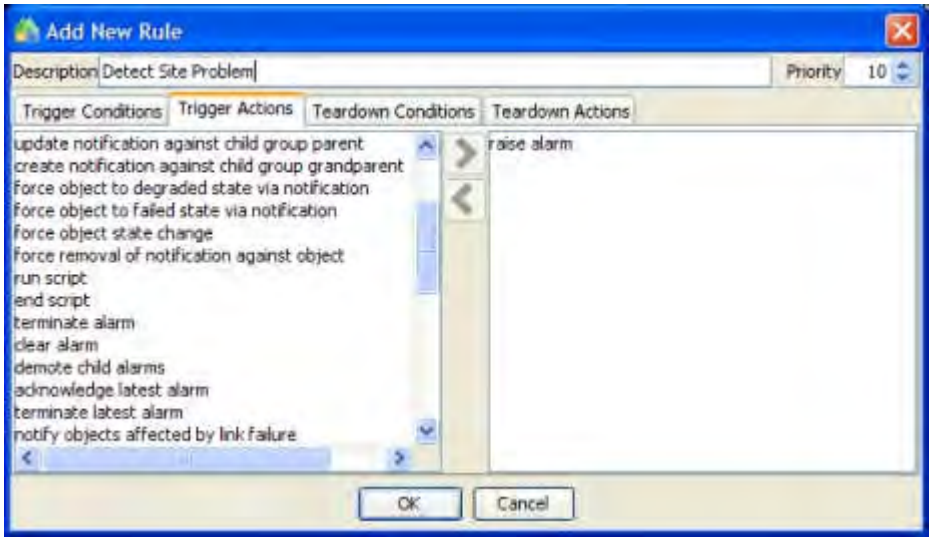


Figure 26 - The Add New Rule Dialog

- Select the Trigger Actions tab to define the action(s) to be associated with the trigger conditions.
- Select the required action in the left side of the screen and click the button. In the resulting 'Add Trigger Action' dialog, enter the appropriate values and click the **OK** button. The action will be removed from the left hand list and will appear on the right hand list.
- Repeat this for all actions to be added.
- To modify an action in the right hand list, double-click it or right-click and select 'modify' from the pop-up menu item.
- Action will be performed in the order that they are shown in the right hand list – top to bottom.
- To re-position an action in the right hand list, right-click the action and select **move up** or **move down** from the pop-up menu item.
- To remove an action in the right hand list' select it and click on the button. The action will then re-appear in the left hand list.



- Repeat the entire above procedure to specify the teardown conditions and teardown actions in a similar way, but using the 'Teardown Conditions' and 'Teardown Actions' tabs.
- Finally, to complete the rule definition, click on the **OK** button.

The new rule will now be listed in the Scenarios, Filters, Mappings and Rules Summary List in the UCA Scenario Manager.

To view an existing rule:

- Double-click the rule in the Scenarios, Filters, Mappings and Rules Summary List in the UCA Scenario Manager, or right-click the rule and select the **view / modify** pop-up menu item.

To modify an existing rule:

- Double-click the rule in the Scenarios, Filters, Mappings and Rules Summary List in the UCA Scenario Manager, or right-click the rule and select the **view / modify** pop-up menu item.
- Make the necessary changes and click OK.

To include a rule in a deployment:

- If the rule is to be included in the set of scenarios, filters, mappings and rules for an active deployment, it must be dragged from the Scenarios, Filters, Mappings and Rules Summary List and dropped onto the 'rules' node in the Scenario Builder Tree. Once this has been done, the rule will be shown in the tree. The example below shows five rules in a particular scenario.

To remove a rule from a deployment:

- If the rule is to be removed from the set of scenarios, filters, mappings and rules for an active deployment, right-click the rule in the Scenario Builder Tree and select 'delete from tree' in the pop-up menu.

8.4.1 Rules and user-defined event fields

User-defined event fields are also accessible for use in rules. Each user-defined event field is accessible via the 'last event' raised on a Mesh Object.
An example rule utilising a user-defined event field is shown below:

8.5 Rule templates

8.5.1 Templated Rules

Rules can be created as described in the above section, but they can also be created using ‘templates’. A ‘templated rule’ acts like a pattern for generating actual rules later on. In a template rule, the rule conditions and actions are defined as usual but the actual values used in the conditions and action fields are not supplied when the template is defined, but instead ‘variable names’ are used in their place and the actual values for the variables are supplied later. Collections of ‘templated rules’ are very useful for addressing general purpose situations, for example commonly seen state propagation rules. A collection of templated rules can be used again and again; each time the actual rules are generated from the templates, different values for the ‘variables’ may be used.

The key points to be aware of when templating a rule are:

- Any rule can be ‘templated’
- You can template a rule’s trigger / teardown attribute conditions
- You can template a rule’s trigger / teardown actions

To template a rule’s trigger / teardown attribute condition, select the ‘use template attribute?’ checkbox when adding or modifying a rule’s attribute condition. When this is done, instead of supplying a value in the attribute condition, you will be prompted to enter a template attribute name and description. The template attribute name acts like a ‘variable’ for which you will later supply a value. The description is useful to help clarify the meaning of the template attribute name.
To template a rule’s trigger / teardown action, right click a desired action field and in the resulting ‘Action Template Item’ dialog, select the ‘use template attribute?’ checkbox and enter a value for the template attribute name and description, as above.
You can use the same value for the template attribute name for many conditions and actions in many template rules.

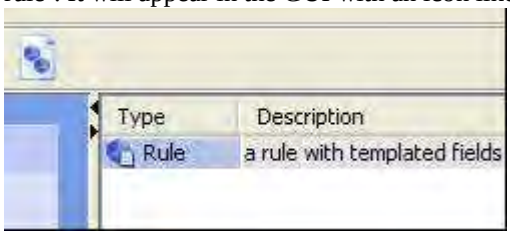
- Any rule can be template
- You can template:
  - trigger / teardown rule attribute conditions
  - actions

right-click a rule action field to get the template dialog

e.g. “baseClass” might be a sensible template attribute name here

Attribute names act like ‘variables’ for which you later supply a value
For example, wherever you’ve used “baseClass”, the actual value for this (eg. NetworkElement”) will later be substituted as the value in the expression wherever “baseClass” occurs.

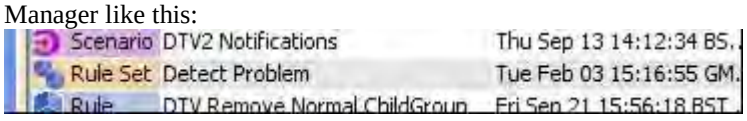
Any rule that uses template attribute names, either in its rule attribute conditions or action fields, is a ‘templated rule’. It will appear in the GUI with an icon like this:



8.5.2 Rulesets

A Ruleset is simply a container for rules that have been templated.

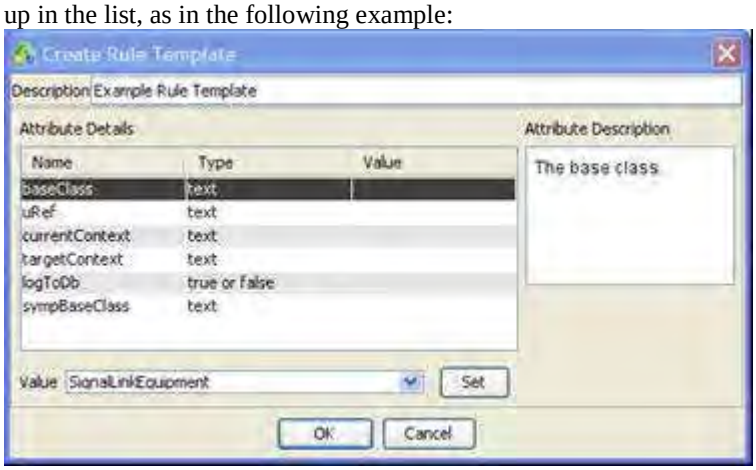
To create a ruleset, select the  icon in the Scenario Manager toolbar. Once you have clicked this icon you can drag any templated rule into the resulting RuleSet dialog.
When a RuleSet has been created, it will show up in the Summary List table on the right side of the Scenario Manager like this:



A Ruleset acts as the vehicle for generating the actual rules from the template set of rules.

8.5.3 Using a ruleset

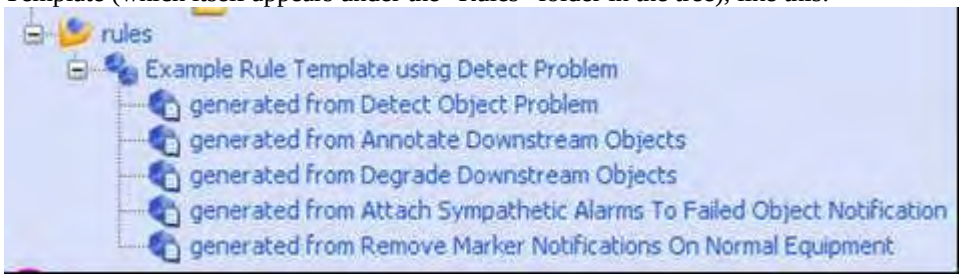
To use a RuleSet, drag it to the Rules folder of a Scenario in the Scenario Builder tree on the left of the Scenario Manager.
When the RuleSet is dropped on to the tree, a “Create Rule Template” dialog box will appear
All templated rule attribute conditions and action fields from the template rules in the dragged RuleSet will show up in the list, as in the following example:



This dialog is used to supply the actual values to be substituted in place of the template names in the templated rules within the Ruleset.
You must supply a value for each name in this dialog e.g. for “baseClass” the actual value you might enter could be “NetworkElement”.

8.5.4 Generating the rules from the rule template

When all values have been supplied in the Rule Template Dialog box and “OK” clicked, a set of new rules will be automatically generated. These new rules will appear in the Scenario Manager GUI under the new Rule Template (which itself appears under the “Rules” folder in the tree), like this:



You can right-click on the Rule Template in the tree and view or modify the set of values to re-generate new rules based on the new values.
You can also right-click on any of the auto-generated rules and view (read-only) its details.

8.6 Deploying Scenarios, Filters, Mappings and Rules

Once the Scenario Builder Tree has been set up with all the scenarios, filters, mappings and rules, it may be deployed into active use. However, the deployment must first be validated.
To validate a deployment:

- click the button on the Scenario Manager toolbar.
- The validation will check that the Scenario Builder Tree is valid (e.g. that mappings exist under filters etc.) and that the rules engine considers the rules to be valid.
- If the deployment validates correctly, the status bar will show “Status: the validation was successful”. Otherwise the ‘Validation Errors’ dialog will open, showing the translated rules code with details of the errors, including the line and column numbers where the error(s) occurred.

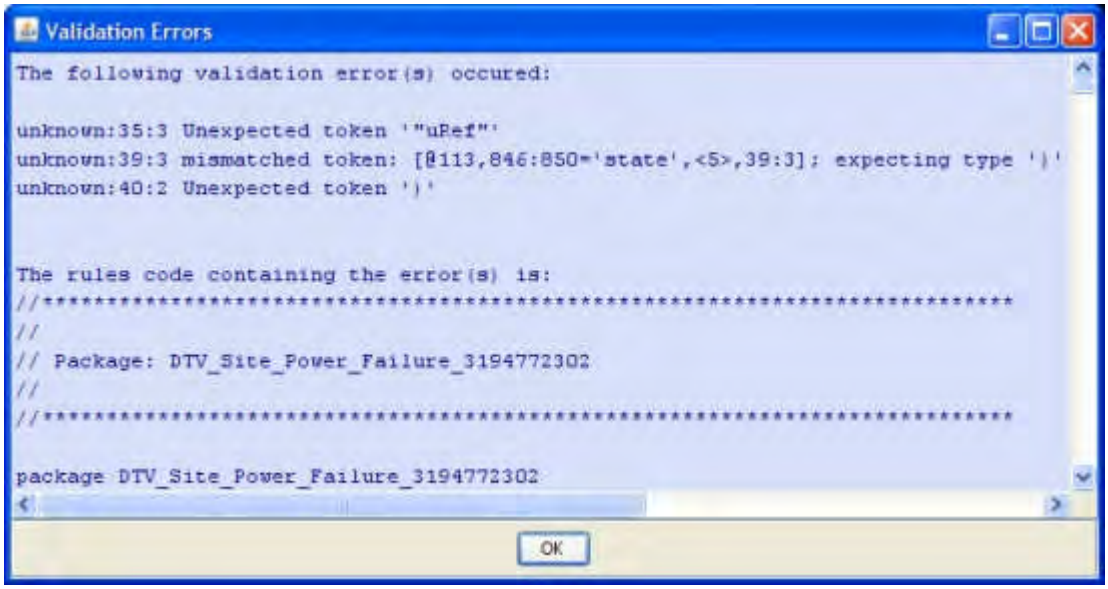


Figure 27 - The Validation Errors Dialog

- The translated rules code In the 'Validation Errors' dialog is annotated with comments that match the text in the Trigger / Teardown Conditions trees of the Add or View / Modify Rule dialogs. This allows precise location of which part of which rule has not validated.

The set of scenarios, filters, mappings and rules can only be deployed after they have been successfully validated. After validation, to deploy the scenarios, filters, mappings and rules present in the Scenario Builder Tree:

- click the button on the Scenario Manager toolbar.
- In the 'Deployment Details' dialog, enter a description and additional information that describes this deployment and click **OK**

A warning may be raised when using value packs (see section Chapter 14) that include rules. This will only happen if the new deployment does not include rules for every value pack that incorporates rules.

To show details of previous deployments, or to open a previous deployment into the Scenario Manager, click the button on the tool-bar or select 'Show Deployments' from the Server menu.
To show details of scenarios added to the 'Scenario Library', or to merge a scenario form the 'Scenario Library' into the existing set of scenarios, click the button on the toolbar or select 'Show Library' from the Server menu.
Details of using the 'Show Deployments', 'Scenario Library', exporting a scenario to the library and importing a scenario from the library are provided in section 7.1.

Chapter 9 Configuring Rules and Actions

Rule conditions and corresponding actions are defined together to form a block. These rule condition-action blocks (commonly referred to as Rules with a capital R) are further divided into two sections; a trigger section that reacts to the departure of one or more objects from the normal state i.e. 'on the way in', and a teardown or recovery section that reacts to the return of one or more objects to the normal state i.e. 'on the way out'. In practice either the trigger or teardown section may be left undefined if they are not required.

9.1 Format

Rule condition-action blocks (Rules) as a whole are assigned a priority in the range 0 to 100. Satisfied Rules with higher priority i.e. more positive, are executed ahead of others on the inference engine agenda with lower priority. This allows the user to force one Rule to execute ahead of another. A useful technique to adopt when designing correlations that rely on this feature is to construct a state diagram for the object(s) involved. Priorities may then be used to force a particular path in the state diagram ahead of another if there is an equal choice.

9.1.1 Structure

Each section of a Rule (trigger or teardown) consists conceptually of a set of object existence conditions sub-divided into clauses (each potentially referring to a different Object, Associate or Child Group, Notification or Script and containing one or more attribute conditions) that are evaluated by the inference engine and one or more consequential actions to carry out when all of those object existence conditions are satisfied. The general format of each section of a Rule is:

(Clause 1) When an Associate Group|Child Group|Object|Notification|Script exists|does not exist with:

Attribute1	**Comparison Operator**	**Expression** (is true)
Attribute2	**Assignment Operator**	**VariableX**
	…	

(Clause 2) And (optionally)
When an Associate Group|Child Group|Object|Notification|Script exists|does not exist with:

Attribute1	**Comparison Operator**	**Expression|VariableX** (is true)
Attribute2	**Assignment Operator**	**VariableY**
	…	
	…	
	…	

(Clause N) And (optionally) …
Then

Action1 (Argument List)
Action2 (Argument List)
…

When object existence conditions are evaluated, the inference engine begins evaluating the first clause and proceeds until it encounters an attribute condition that is not yet satisfied or all of the clauses are satisfied. If an attribute condition is invalid and is subsequently satisfied, evaluation continues from that point onwards (previously satisfied attribute conditions are not re-evaluated unless the object is removed and re-inserted into a working memory). When all of the clauses are satisfied, the associated action(s) are executed. If an object is removed from a working memory before all of the object existence conditions are satisfied, then knowledge of all previously satisfied attribute conditions is discarded.

A general principle for object existence conditions that is a direct consequence of the use of generalised objects in the state mesh is that for a given object type, it is usually necessary to:

- Identify the specific class of generalised object (or that of its parent and/or the objects it contains)
- Evaluate one or more conditions relating to the identified object.

It is also important to realise that unless specifically made so, object existence conditions are non-specific and operate at the class level so that they will operate for any and all matching instances that are encountered in working memory.
A limitation of the underlying use of the JBoss Rules 3 inference engine is that variables initialised in a clause may not be evaluated in the same clause. This limitation may be lifted in later releases of UCA.
Where non-existence in working memory of an Associate or Child Group, Object, Notification or Script with particular attributes is tested in a clause, it is important to note that a reference to that (non-existent) item cannot be used in an action (because by definition it does not exist and therefore it has a null object reference). Further, it is also not possible to store the value of an attribute in a non-existent item in a local variable (again because the system is testing that it does not exist and therefore would not have an attribute value to store in the variable). However, attribute evaluation conditions for a non-existent item may be evaluated against local variables (provided they were initialised in a previous clause for an object that exists).

9.1.2 Rule Conditions

9.1.2.1 Object Types

UCA supports the evaluation of the following object types in object existence condition clauses:

- Objects (both static mesh object components of the state mesh & dynamically created alarm collectors)
- Child Group
- Associate Group
- Notification
- Script

The Object type is a generic name to describe static long-lived mesh object components of the state mesh and dynamically created short-lived alarm collectors (designed to hold a set of transient events from one or more stream sources). In practice, Objects are implemented using the same type of Java object but in addition to lifetime considerations, the former also differ in that they have relationships to surrounding objects defined, whereas alarm collectors exist in isolation from other components of the state mesh.

9.1.2.2 Attributes

Each supported object type has a number of attributes that may be evaluated by attribute comparison operators in a clause. Each attribute has a type and some object types support a common subset of attributes. Section 12.1 lists the supported attributes, their types and a brief description of their purpose for each object type.

9.1.2.3 Operators and Expressions

Each attribute type (String, Integer, Boolean, Enum, Object, Child Group, Associate Group) may be evaluated using an operator against an expression. The following table lists the supported operators for each attribute type and the required expression type.

Operator	String	Integer	Boolean	Enum	Child Group	Associate Group	Expression Type
Is (equal to)	✓						String
Is not (equal to)	✓						String
Contains	✓						String
Does not contain	✓						String
Starts with	✓						String
Ends with	✓						String
Is (equal to)		✓					Integer
Is not (equal to)		✓					Integer
Is greater than		✓					Integer
Is greater than or equal to		✓					Integer
Is less than		✓					Integer
Is less than or equal to		✓					Integer
Is greater than value in (variable)		✓					Integer
Is greater than or equal to value in (variable)		✓					Integer
Is less than or equal to value in (variable)		✓					Integer
Is less than value in (variable)		✓					Integer
Is (equal to)			✓				Boolean
Is not (equal to)			✓				Boolean
Is (equal to)				✓			Enumeration
Is not (equal to)				✓			Enumeration
(Group) contains					✓	✓	Mesh Object
(Group) does not contain					✓	✓	Mesh Object
Is stored in [assignment operator]	✓	✓	✓	✓	✓	✓	Assigning Object Type

Figure 28 - Operators and Expressions

The ‘stored in’ operator is the only assignment operator (all others are conditional) and may be utilised to store the current value of an attribute or expression into a local variable (whose type is automatically determined from that of the assigning object’s type). The scope of a local variable is the remainder of the Rule section (trigger or teardown) in which it is declared, beginning with the next rule clause (if one exists) or the following action(s).

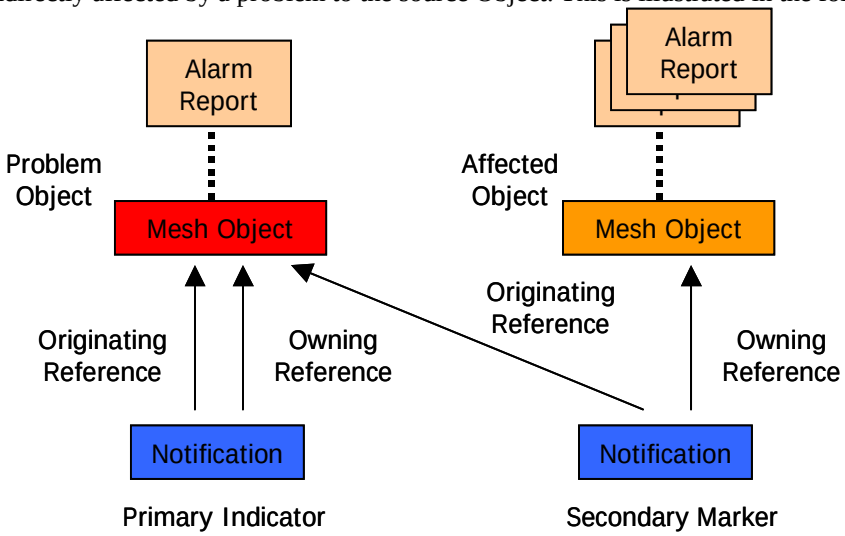
9.1.3 Actions

Once the object existence conditions of a Rule are satisfied, one or more consequential actions may be executed. The system supplies a comprehensive set of pre-defined actions to choose from and additional user-defined actions may be created as required.

(Mesh) Objects and Child & Associate Groups can exist outside working memory (regardless of their current state, they are still part of the state mesh). (Alarm collector) Objects, Notifications and Scripts only exist inside working memory – they are transient objects that exist for the purpose of collecting alarm streams, reporting a correlation notification or managing a script execution respectively.
Notifications are of interest in that they carry two sets of references to Objects. The 'originating' reference (base class and unique reference) is normally used to identify the Object whose 'problem' is the reason for its existence e.g. a non-NORMAL state. The 'owning' reference (base class and unique reference) is used to identify the object that it is currently associated with. This allows Notification objects to work in two ways:

- As a primary indicator of a problem – both originating and owning references refer directly to a 'problem' Object.
- As a secondary marker on another Object affected by the 'problem' Object – the originating reference refers to the 'problem' Object, while the 'owning' reference refers to the 'affected' Object.

Given these features, Notifications can be used for the purpose of constructing correlations where it is necessary to link Objects indirectly affected by a problem to the source Object. This is illustrated in the following diagram:



The arrangement shown in the diagram above illustrates how UCA may be used to gather sympathetic alarms from Objects affected by a failure elsewhere in the state mesh. Assuming that an action has created the primary Notification object against the 'problem' Object, then another action (usually produced specifically for that purpose) can identify potentially affected Objects and attach secondary marker Notification objects to them, in turn referring back to the 'problem' Object. Once this link is constructed, then any sympathetic alarm reports attached to the 'affected' Object may be tied to the original problem.
One of the purposes of building notifications is to report useful information back to the user via notification reports on the Notification Viewer GUI. Notification objects by themselves do not achieve this purpose. To make a notification report visible on the GUI, a notification record needs to be created in the UCA notification database. The separation of these two functions is necessary to allow flexibility in the use of Notifications – often the problem they represent does not need to be visible to users via the GUI, particularly where they are used as an intermediate step in a correlation that may involve several levels of the model.
When a notification report is displayed on the Notification Viewer GUI, it will often be accompanied by a list one or more alarm reports. Typically, an action that sets out to create a notification report will carry out the following operations:

- Build an event list of existing contributory alarm report records in the notification database associated with the problem Object (recall that all alarm reports that pass the input filters are stored in the event database).
- Build a Notification record in the notification database and attach the contributory event list. This will result in an automatic display of a notification report and accompanying contributory alarm reports from the event list on the GUI.
- Build an equivalent primary indicator Notification object in working memory from the notification record to support further processing.

If the correlation requires the attachment of affected Objects and their sympathetic alarm reports to a notification report that is already displayed on the Notification Viewer GUI, then a slightly different approach is adopted in relation to the notification database. The following operations will be necessary for each affected Object:

- Build (or append to) an event list of existing sympathetic alarm report records in the notification database using the affected Objects as the source of the alarm reports. The sympathetic event list is attached to the notification record on the problem Object.
- Build (or append to) a list of affected Object records in the notification database using the affected Objects themselves. Again, the affected Object list is attached to the notification record on the problem Object.
- Build an equivalent secondary marker Notification object in working memory from the affected Object (for the 'owning' Object reference) and the problem Object (for the 'originating' Object reference).

An additional action may add late arriving alarm reports to the sympathetic event list as required, allowing the notification report to gather further alarm reports over an extended period.
The types of action available depend on the rule-action block section in which they are initiated. Section 12.2 describes in detail the currently supported set of actions available to each rule-action block section (Trigger and Teardown).
Actions often require configuration parameters to be supplied from the objects associated with rule clauses. Objects that participate in rule clauses are automatically assigned names depending on their type and position in the set of clauses. The following naming convention is adopted:

- Objects; name = objNN
- Associate Groups; name = assocNN
- Child Groups; name = childNN
- Notifications; name = notifNN
- Scripts; name = scriptNN

Where NN is an integer, beginning at 0 and incrementing independently for each type, so if a rule contained two Object clauses and a Notification clause, then these objects would be automatically assigned the names; obj0, obj1 and notif0.
During action configuration, the user may also be given the option to provide message text (literals enclosed in " " or rule condition variable names) or other additional values. Generally, the user is also given the option to record action execution in the notification database (which results in the data being presented on the Fired Rules GUI). The only exception to this is the situation where an action creates a Notification object and it must be recorded in the notification database – in this instance, the user is not given the option. For efficiency, it is recommended that once initial testing has been completed that the absolute minimum number of action execution logs are created, consistent with user audit trail maintenance requirements.

9.2 Example Rules and Actions

The DTV example included with the UCA installation contains a set of Rules designed to implement the following correlation scenarios:

- DTV Site Power Failure – Creates a DTV Site Power Failure primary Notification, identifies the downstream DTV Sites and Receivers affected by an upstream DTV Site Power failure, attaches marker Notifications to the downstream DTV Sites and Receivers and gathers any sympathetic alarms under the primary Notification.
- DTV Service Impact - Identifies the DTV Services affected by localised Receiver problems.
- DTV Maintenance – Handles the retraction of normal Objects & Groups from the DTV context.
- DTV2 Notifications - Creates an additional Notification (in response to the creation of DTV Site Power Failure Notifications) that 'straddles' two working memory contexts, DTV & DTV2. It also handles retraction of normal Objects & Groups from the DTV2 context. The primary purpose is to illustrate the technique for linking correlations in separate contexts.

The following sections describe in some detail the Rules that implement the first of these scenarios and illustrate some of the important features of the remainder.

9.2.1 Correlation Scenario - DTV Site Power Failure

The starting point for definition of a correlation scenario is often the identification of a problem in the monitored network that would benefit from automated correlation analysis. Typically that network problem is characterised by a set of contributory events that are symptomatic of the problem. In addition, there may be an additional set of sympathetic events that occur at other locations in the network as an indirect result of the problem. It is also

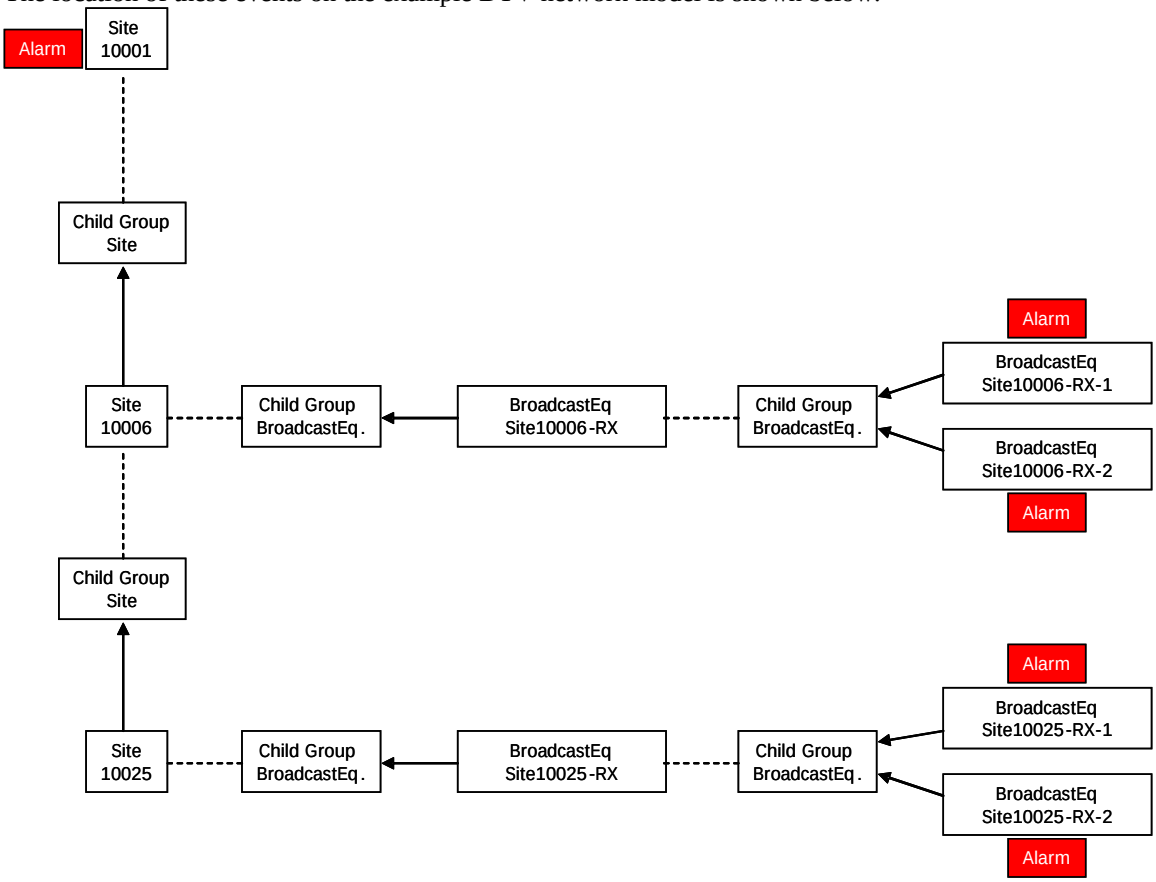
necessary to establish the target requirements for the correlation itself i.e. what is the desired outcome of the correlation.
In the DTV Site Power Failure scenario, the target correlation requirements are:

- Detect a DTV Site that has undergone a power failure and report a Notification.
- Gather any sympathetic events from downstream DTV Sites & Receivers under the Notification.

Definition of the Rules to perform such a correlation scenario usually begins with injection of an example set of contributory and sympathetic events into UCA using the UCA Event Injector tool. This in turn is driven by one or more files containing XML representations of the contributory and sympathetic events. These event files may be hand crafted or created using some automated translation process from existing event histories. A much more convenient alternative is to enable pre-filter logging in UCA and to either instruct the event source system(s) to replay the required events from their own histories or to simply wait for the problem to re-occur. The resulting log files may then be used directly with the UCA Event Injector tool. A major advantage of using the Event Injector in this way is that the captured problem events can be replayed repeatedly during initial testing.
It should also be noted that use of an example event set in this way is just the first step in developing a robust correlation. Any production quality correlation will need to be tested with several other examples of problem events (particularly where they occur in a different order) and ultimately be connected to a live system over a suitable period to ensure that actual problem occurrences are reliably correlated.
The DTV example includes a set of events that are characteristic of a DTV Site power failure and when injected, cause UCA to report the following problems in the UCA Mesh Viewer:

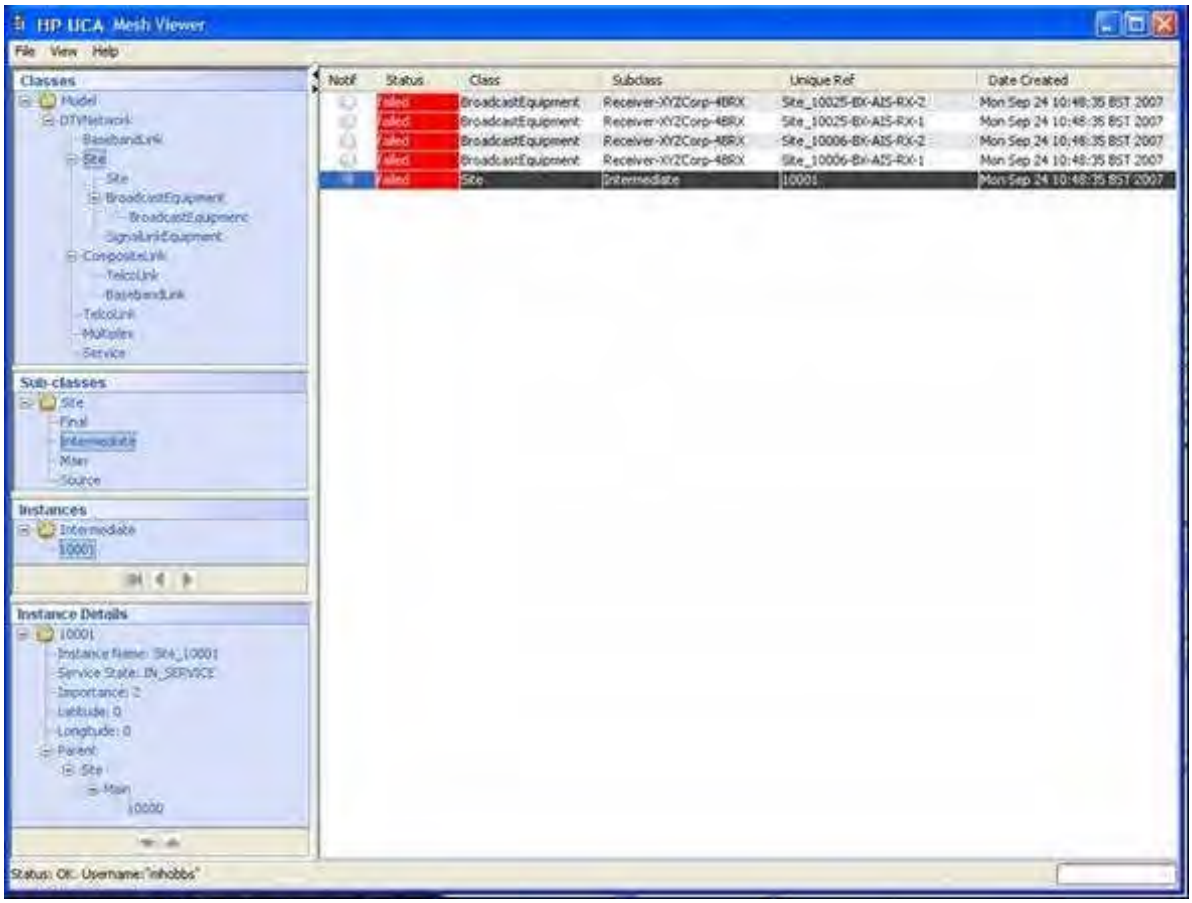
- Site 10001 has undergone a power failure.
- As a result of the power failure of Site 10001 (and consequential loss of transmission capability), component Receivers at downstream Sites 10006 & 10025 have detected a loss of signal from their respective upstream transmitters.

The location of these events on the example DTV network model is shown below.



Assuming that a minimal scenario (without Rules) to handle DTV Site power failures has been deployed in UCA (the preceding chapter describes in detail how to achieve this and the reader is encouraged to examine the scenarios, filters and maps in the supplied example for the actual configurations required), then the presence of

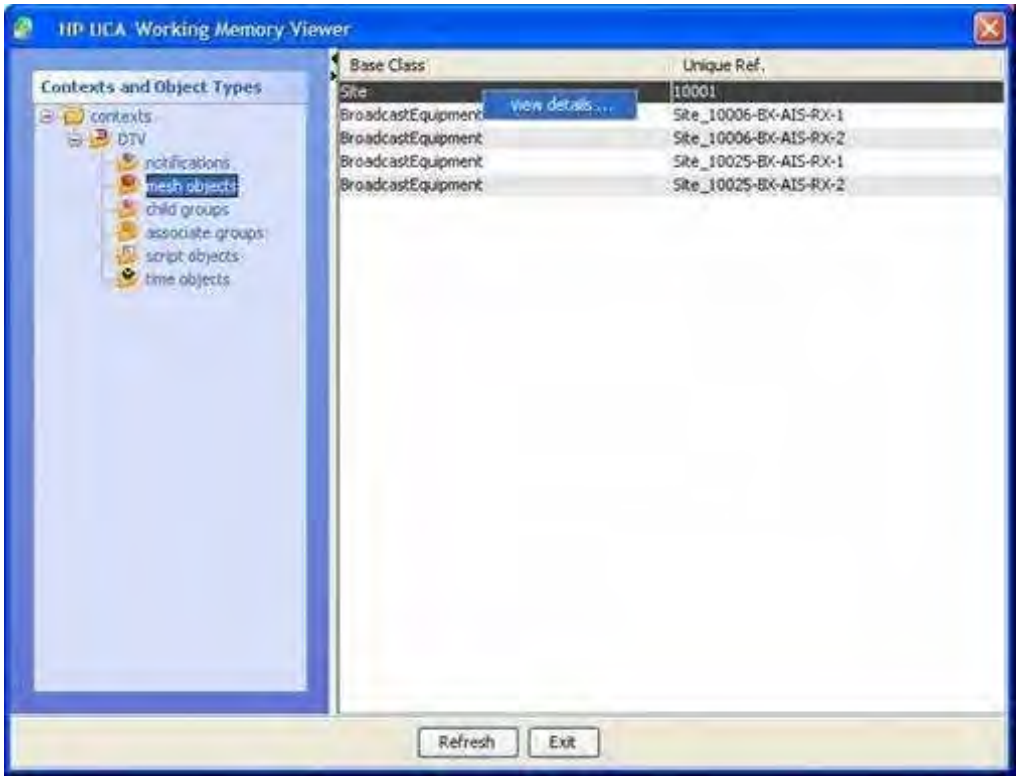
these events causes the equivalent mesh objects to change state, resulting in the following display on the UCA Mesh Viewer.



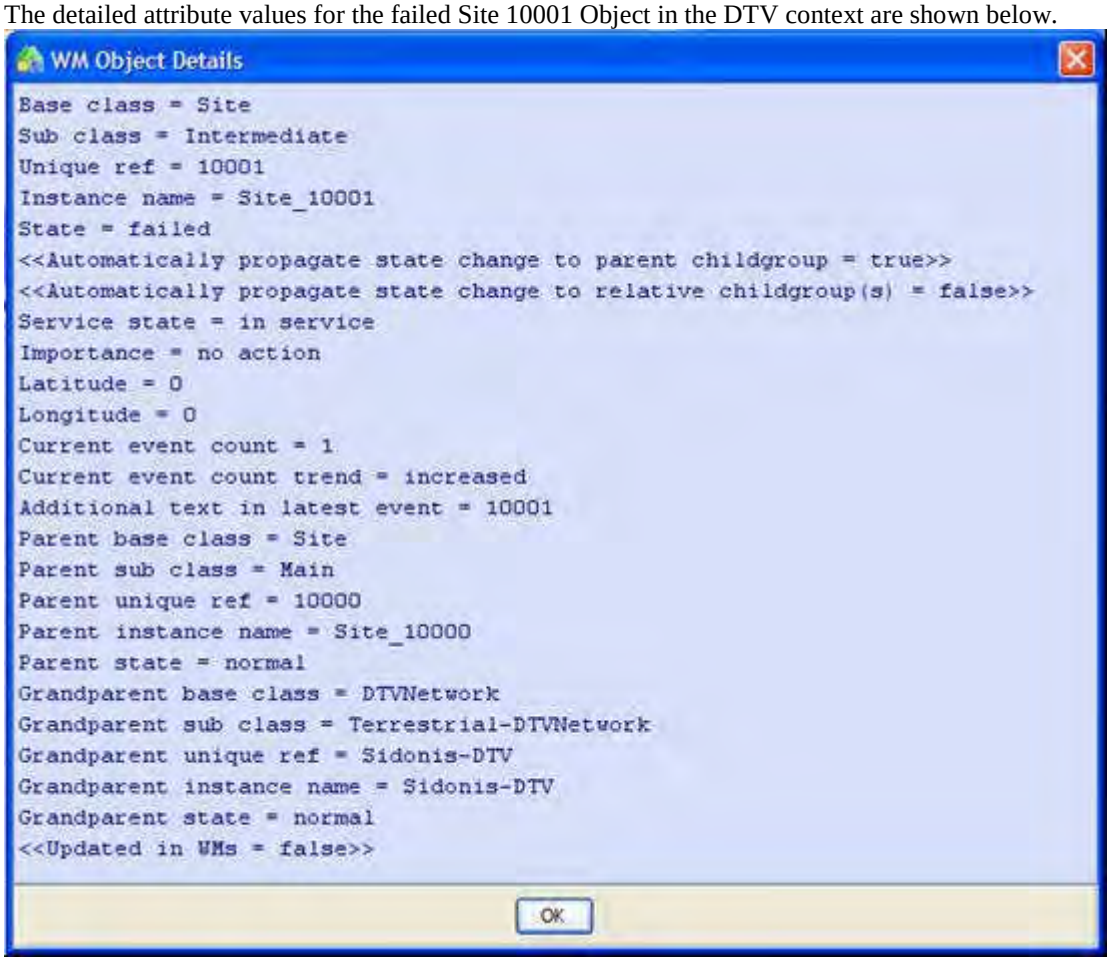
At this point with such a minimal scenario (no Rules have been defined) UCA will not attempt to carry out any type of correlation.

The first requirement for the correlation scenario is the detection of a failed DTV Site object and the creation of a primary Notification reporting the failure.

To satisfy this requirement, a Rule needs to be defined to locate failed Site objects with the correct attribute values. A simple way to evaluate the necessary object existence conditions is to examine the failed Site object in the UCA Working Memory Viewer – see below (recall that the Site 10001 object will be automatically be inserted into the DTV working memory context when the incoming event causes it to adopt the failed state).



The detailed attribute values for the failed Site 10001 Object in the DTV context are shown below.

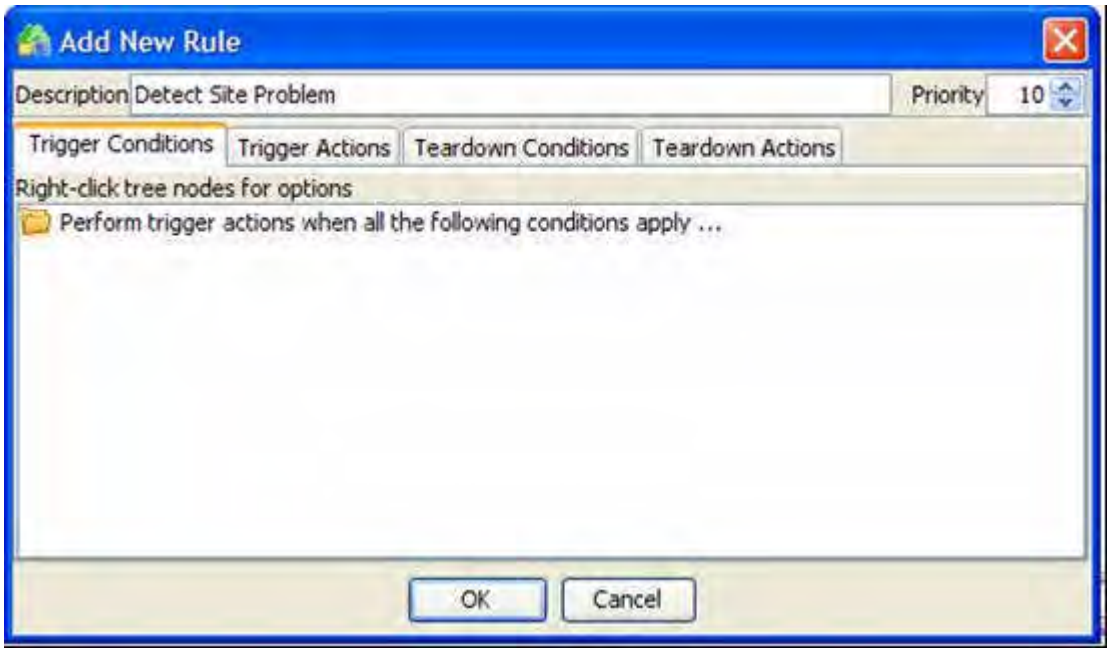


The user should consider that except in special circumstances, Rules are normally intended to operate at multiple locations throughout the network, rather than at specific positions. The choice of attribute conditions to test for in object existence condition clauses (i.e. the constraints) should then be made specific enough to identify the

correct type of Object, Associate or Child Group, Notification or Script in the required state, without un-necessarily limiting the scope of the search (for example by NOT testing for a particular unique reference which limits the Rule to operate at a single location). To this end, a single object existence condition clause (to locate an Object) with the following (naïve) set of attribute conditions should be sufficient to locate failed Site Objects:

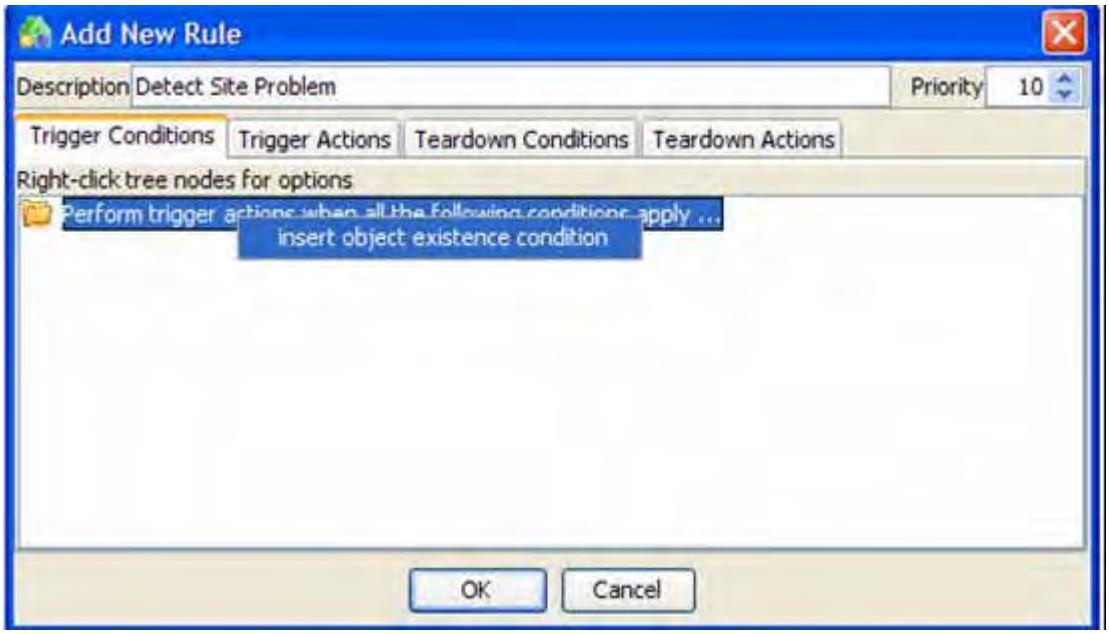
- Base class (i.e. type) is Site.
- State is failed.

In practice, an additional object existence condition clause will be needed to exclude those situations where a primary Notification has already been created on the failed Site object. This additional restriction will prevent a new Notification being created each time any attribute of the failed Site Object is updated (causing a naïve Rule to be re-evaluated).
To begin definition of a suitable Rule (using the UCA Scenario Manager), the option to create a new Rule is selected (as described in the previous chapter) and an empty Rule is created. In this case, a Rule has been created with a name of 'Detect Site Problem' and a priority of 10 (a useful starting point – it can be adjusted later if required). This is illustrated below:



As described previously, a Rule in fact provides for both trigger and teardown object existence conditions and corresponding actions.
The first step in defining a new Rule is normally to define the trigger conditions. In practice this is achieved by selecting the Trigger Conditions tab and entering one or more object existence condition clauses - recall that each such clause constrains the Rule to test for the existence or otherwise of an Object, Associate or Child Group, Notification or script in a working memory). In this example, the first clause will be required to locate failed Site Objects, so the object existence condition must be set to check for the existence of an Object. This is achieved as follows:

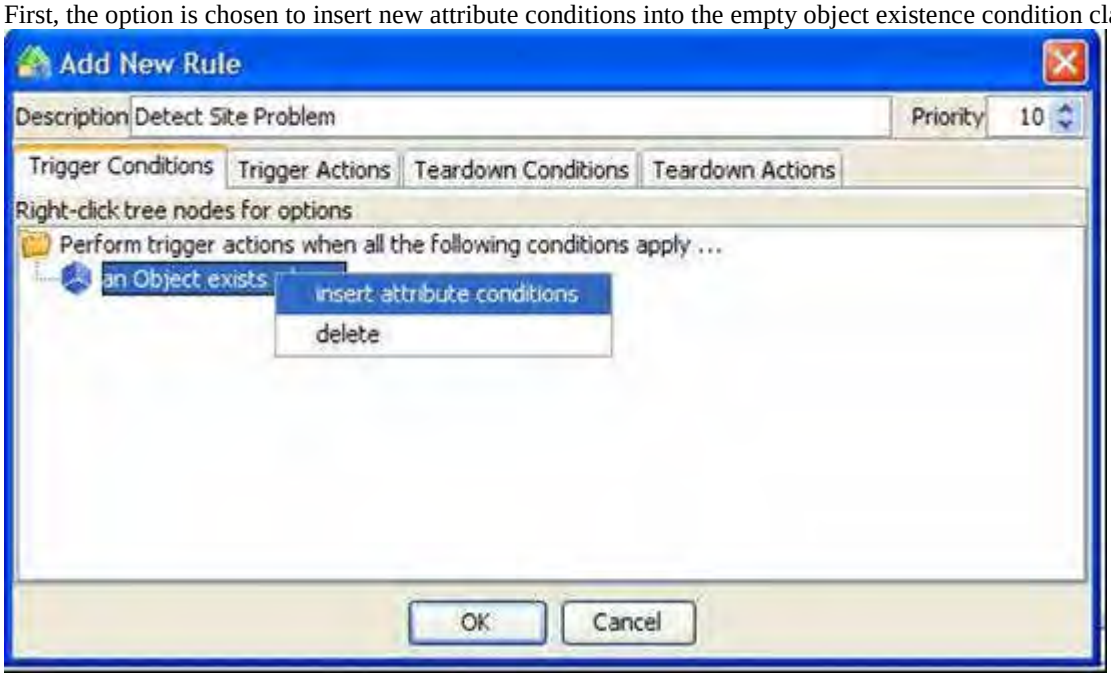
First, the option is chosen to insert an empty object instance condition clause into the trigger conditions of the empty Rule:



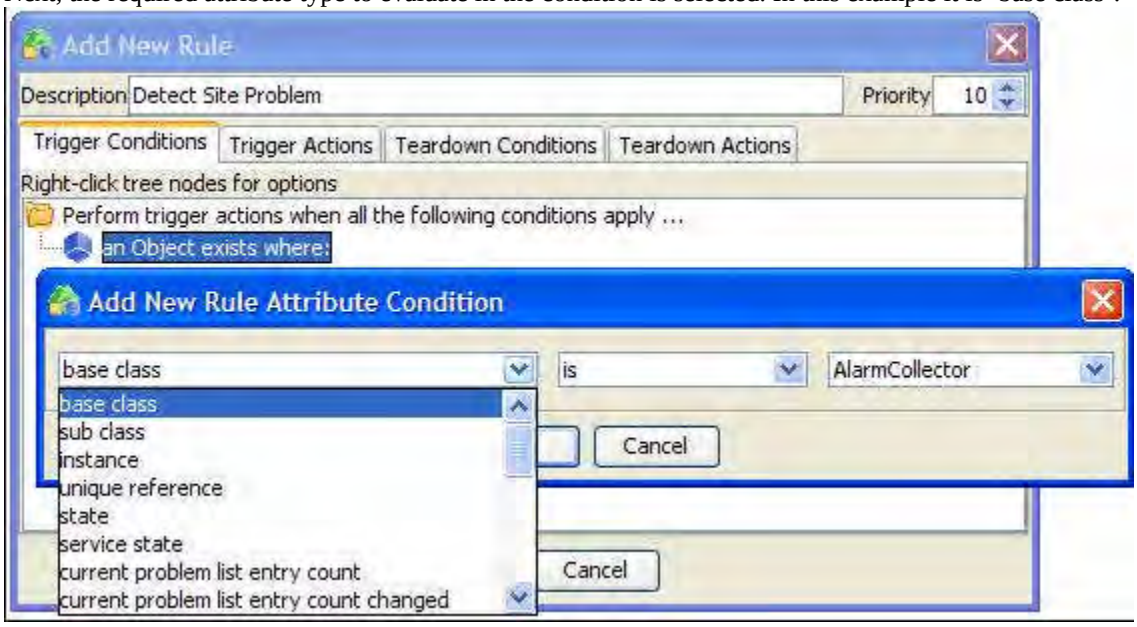
Then, the 'Object exists' condition is added:

Once an empty object existence condition clause for an Object is created, then the individual attribute conditions can be applied as follows:

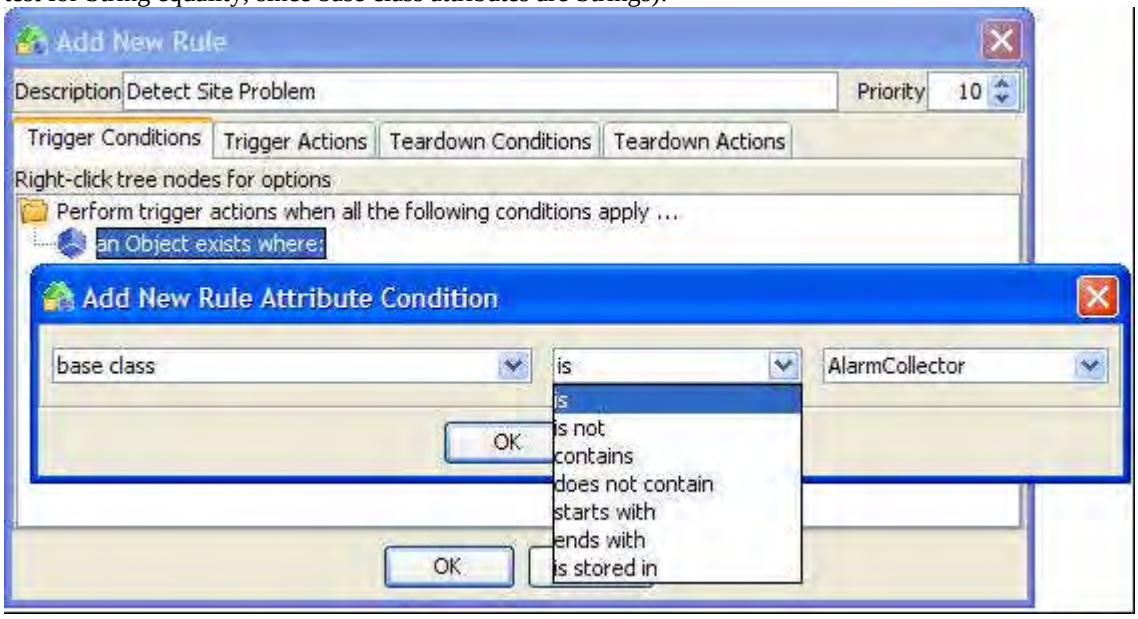
First, the option is chosen to insert new attribute conditions into the empty object existence condition clause:



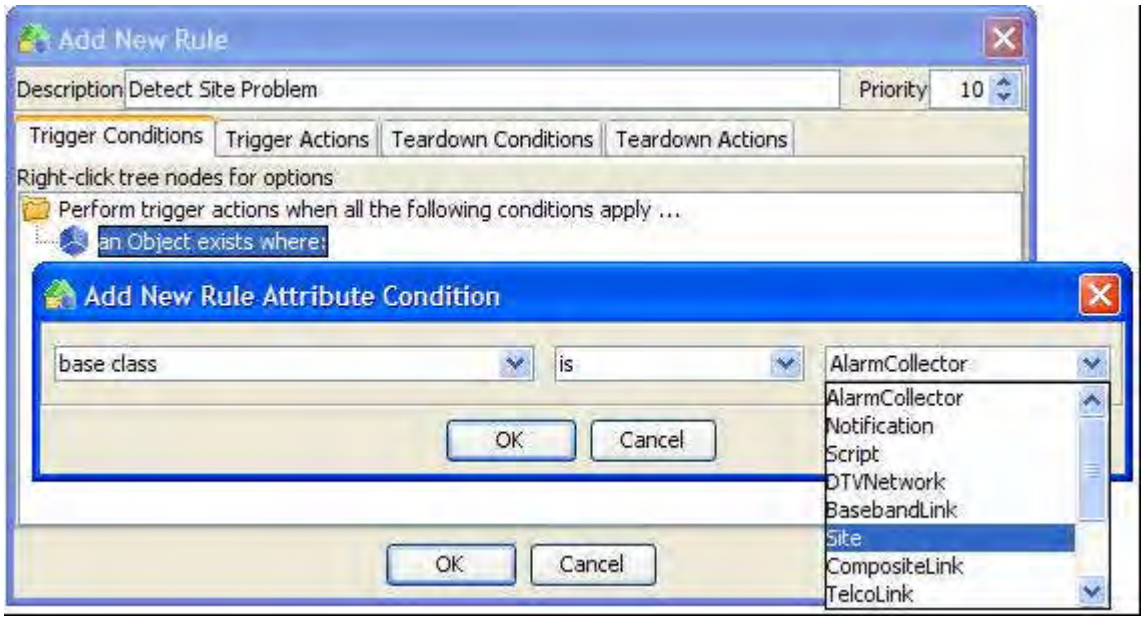
Next, the required attribute type to evaluate in the condition is selected. In this example it is 'base class':



Then, the evaluation operator to apply to the attribute is selected. In this example the ‘is’ operator is used (i.e. to test for String equality, since base class attributes are Strings):

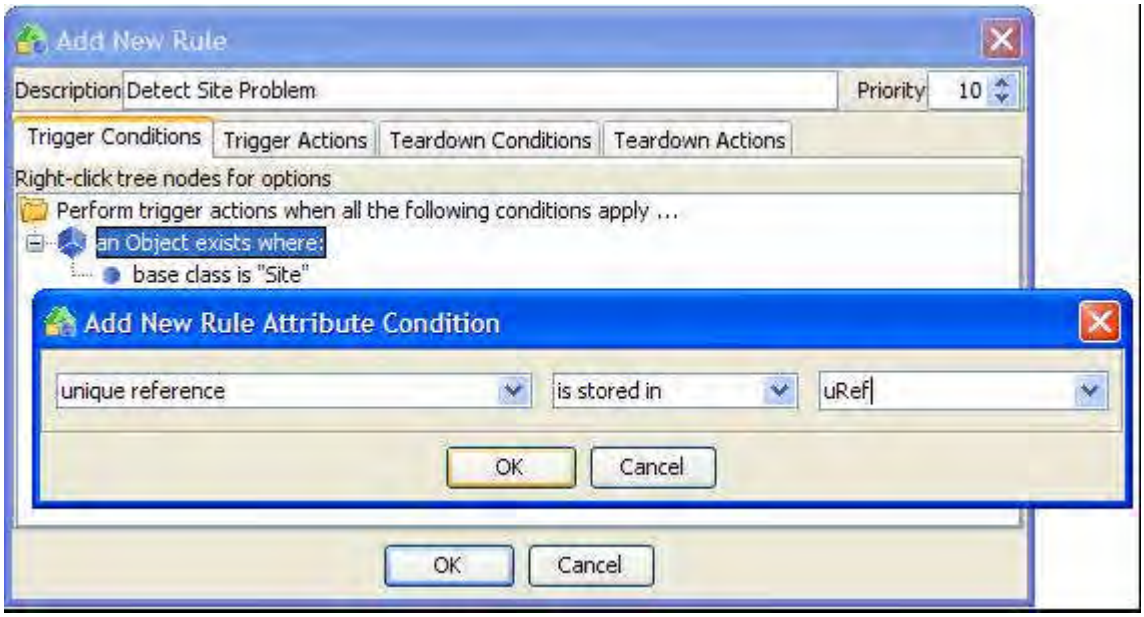


Finally, the required base class name is selected from the available choices. In this example, the ‘Site’ name is used.



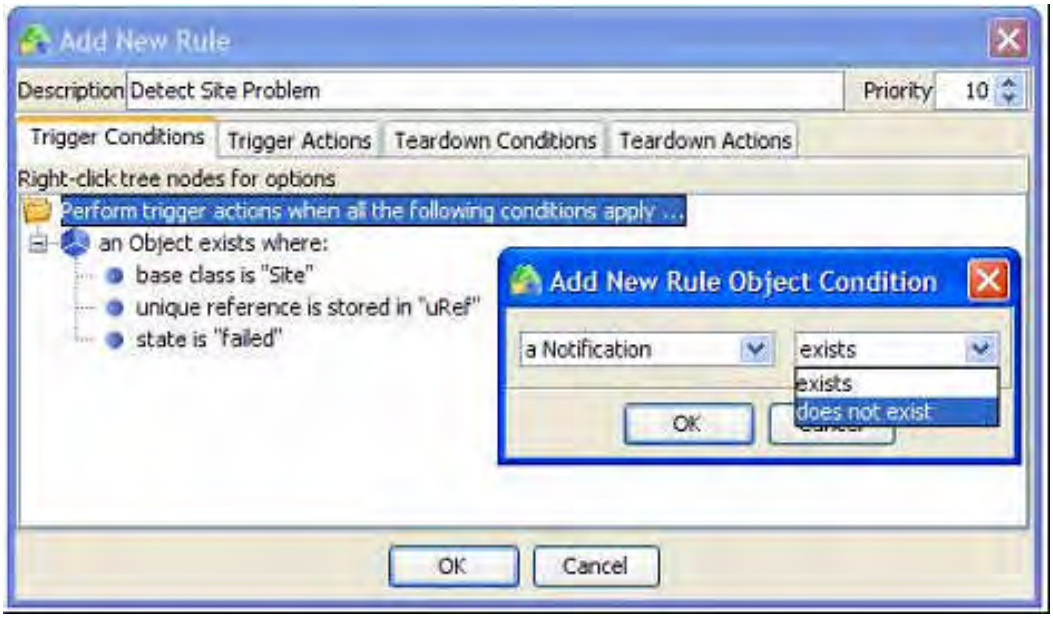
The end result is an object existence condition clause that the inference engine will use to search for all Objects whose base class is Site.

Because it will be necessary to exclude Site Objects that already have Notifications on them, the next attribute condition records the unique reference of the located Site Object in a local variable called ‘uRef’ for use in the next clause.

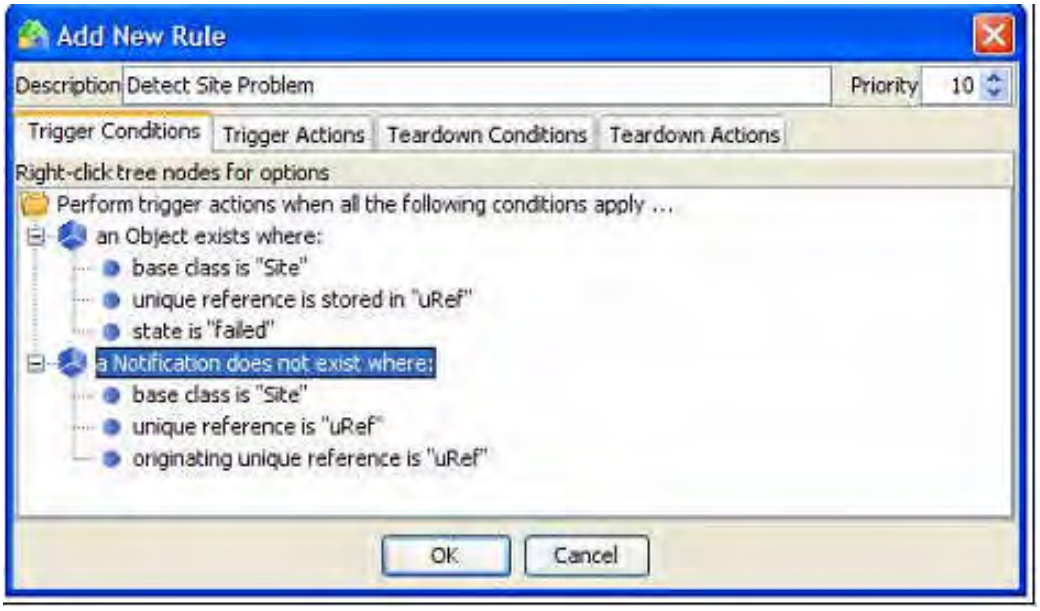


The final attribute condition for this object existence condition clause forces the inference engine to only consider those Sites that are in the 'failed' state.

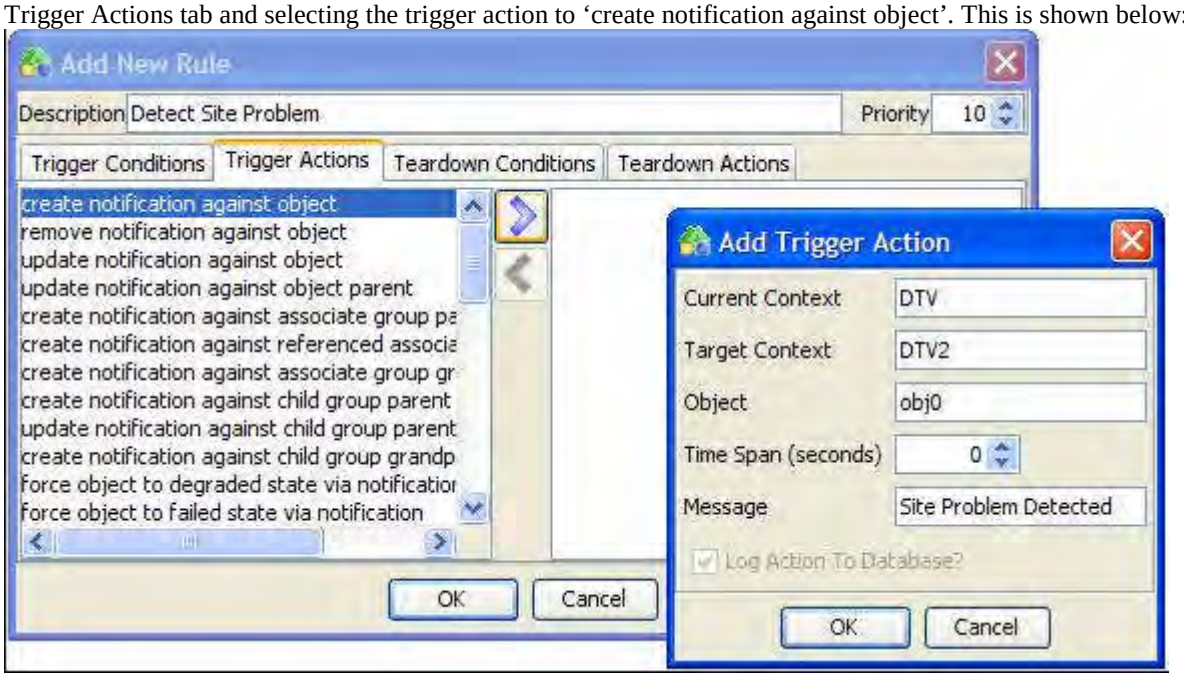
The next object existence condition clause in the trigger conditions is responsible for ensuring that the inference engine only locates failed Sites that do not already have a primary Notification on them. This is achieved by adding a clause that checks for the non-existence of an attached primary Notification as follows:



The attribute conditions for this clause are chosen such that they would identify an existing primary Notification on the failed Site Object identified in the first clause (and remember that the clause for this Notification is checking that it DOES NOT exist, so the rule WILL NOT fire if a matching Notification is found). The unique reference of the previously located failed Site object (stored in the local variable 'uRef' in the first clause) is used in this clause to ensure that a primary Notification on the same failed Site Object does not exist.
The completed trigger conditions are shown below:



Note that the originating unique reference of the Notification is also evaluated against the same Site Object unique reference to ensure that only those failed Sites with existing primary Notifications are excluded (recall that primary notifications have identical 'originating' and 'owning' unique references whereas marker Notifications have different unique references). This is done to allow another instance of the same correlation that originates further up the broadcast chain (and which may had previously created a marker Notification on the now failed Site) to co-exist with a new correlation on the failed Site.
So far, the Rule trigger conditions will only detect a failed Site without an existing primary Notification. The correlation requirement is such that a primary Notification is to be created when this set of trigger conditions is satisfied and this is achieved by executing a corresponding action. The action may be defined by selecting the Trigger Actions tab and selecting the trigger action to 'create notification against object'. This is shown below:



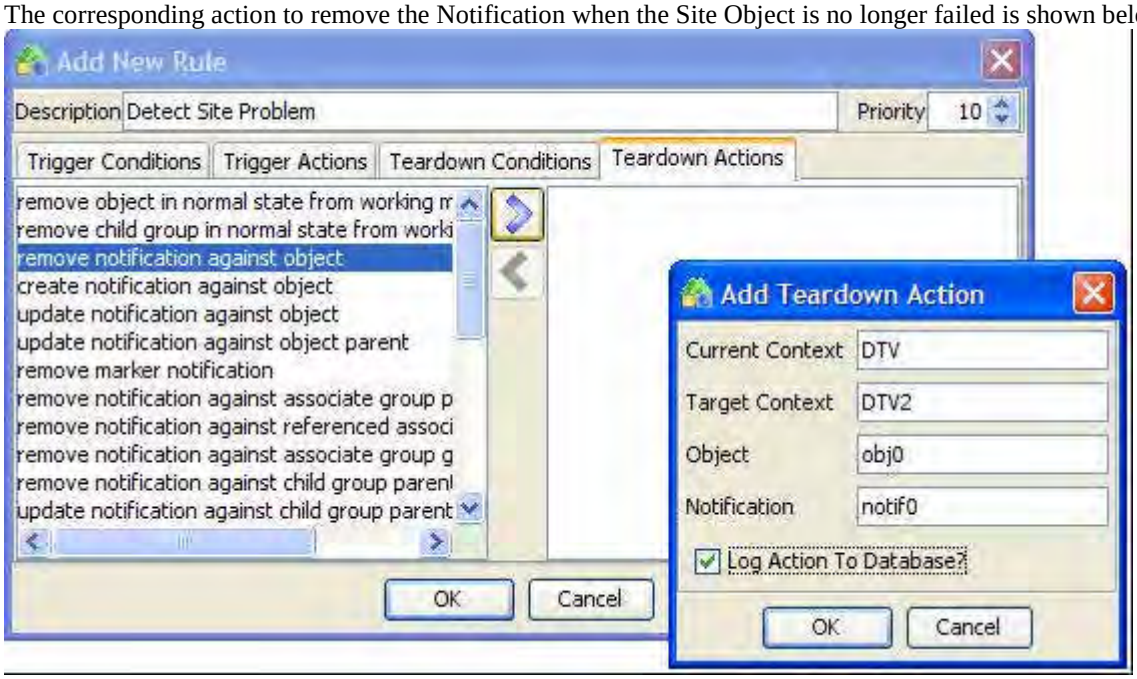
The Add Trigger Action dialogue allows the action to be configured in a number of ways:

- Current Context i.e. the working memory in which the Rule will search for objects that match its trigger object existence conditions and also in which it will insert the corresponding Notification object.
- Target context i.e. an additional working memory in which the Notification will also be inserted. This may be the same as the Current Context in which case it has no effect.
- Object refers to the Site Object identified in the trigger conditions. As there is only one such Object identified in this example, its name will be 'obj0' according to the previously described naming conventions.
- Time Span (Seconds) allows the user to specify a maximum age (relative to the time at which the Rule triggers) of contributory events attached to the identified Object that should be added to the contributory events list of the Notification. In this example, a value of 0 signifies that all non-Normal events attached to the identified Object should be attached.
- Message is the text message that will appear in the equivalent notification entry in the UCA Notification Viewer.

Note that the 'Log Action to Database?' checkbox is greyed out. This means that the Rule trigger and associated Notification creation will always be recorded in the UCA notification database.
Once configured, this action will appear in the list of trigger actions, as shown below:

As described previously, an equivalent set of teardown conditions and associated teardown actions are usually defined to allow the correlation to correctly handle network recovery. Based on the original correlation requirements, the scenario is expected to close a primary Notification if the affected Site Object no longer exists in the failed state (i.e. it could be degraded or normal). Based on this description, the corresponding teardown conditions are shown below:

The object existence condition clauses are subtly different from the trigger case. In particular, the first clause is designed to locate an existing primary Notification and to remember the unique reference of the Site to which it is attached.
The second object existence clause searches for a Site Object that is no longer in the failed state i.e. has become degraded or normal as a result of network recovery, and uses the same Site Object unique reference as that of the Notification located by the first clause. In this situation, the priority of the Detect Site Problem Rule becomes important because the DTV Maintenance Rules responsible for removing normal Objects and Groups from the DTV context execute by default at priority 0. If this Rule also had a priority of 0, then an unpredictable race-condition could exist in which the time order of placing satisfied Rules onto the inference engine agenda would become important, leading to unpredictable correlation recovery behaviour. By setting the priority of this Rule to 10, it is guaranteed to execute before the appropriate Maintenance Rule with consequent predictable behaviour.
The corresponding action to remove the Notification when the Site Object is no longer failed is shown below:

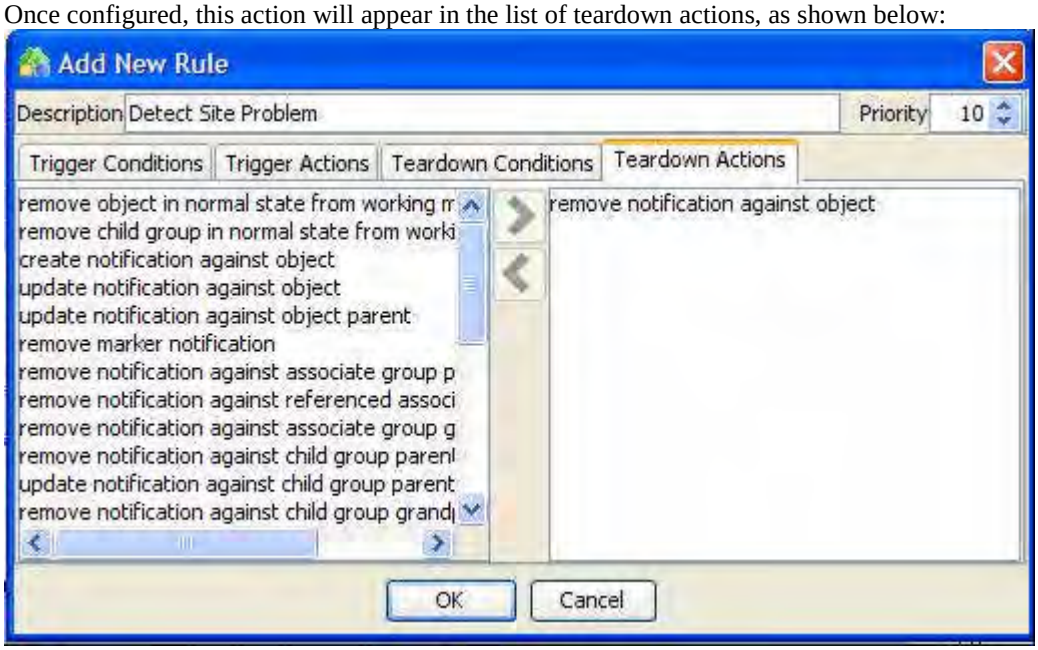


The Add Teardown Action dialogue allows the action to be configured in a number of ways:

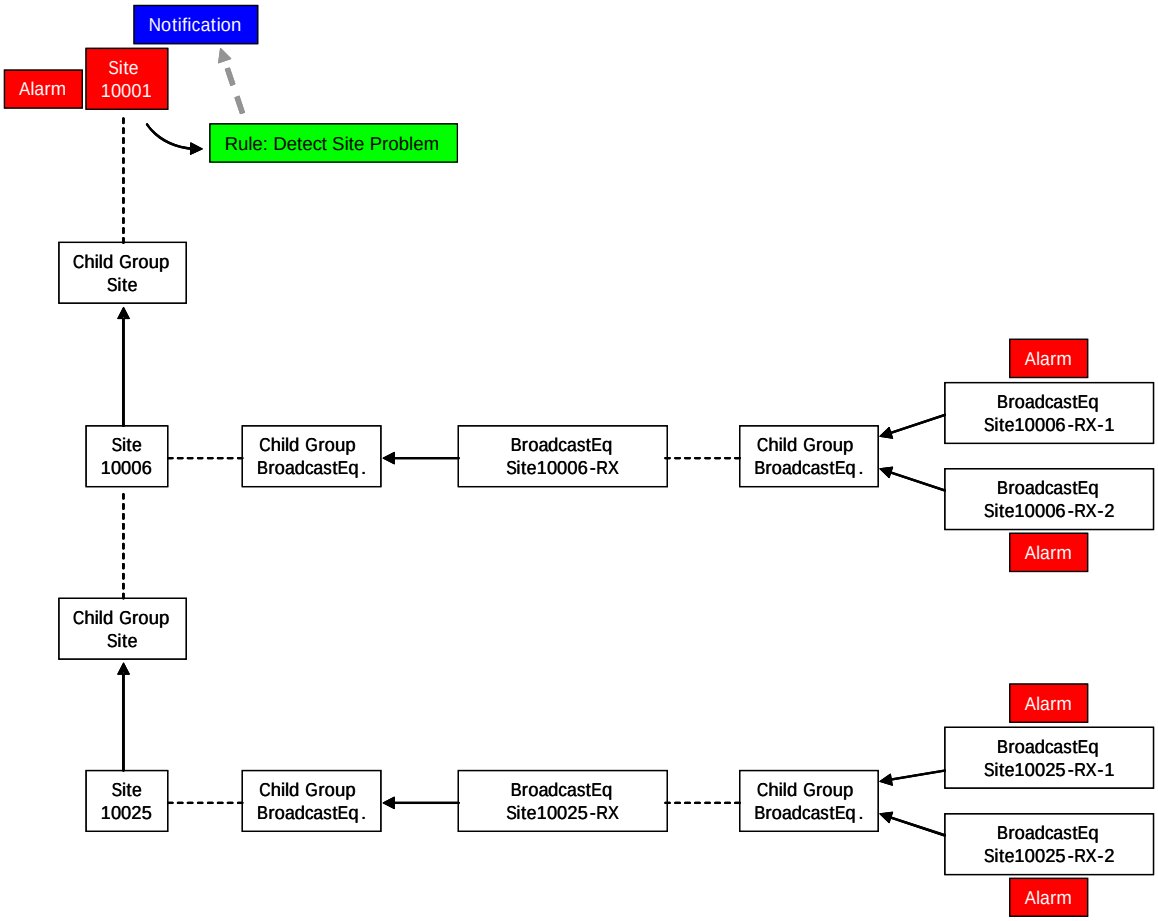
- Current Context i.e. the working memory in which the Rule will search for objects that match its teardown object existence conditions and also from which it will remove the corresponding Notification object.
- Target context i.e. an additional working memory from which the Notification will also be removed. This may be the same as the Current Context in which case it has no effect.

- Object refers to the Site Object identified in the teardown conditions. As there is only one such Object identified in this example, its name will be ‘obj0’ according to the previously described naming conventions.
- Notification refers to the Site Notification identified in the teardown conditions. As there is only one such Notification identified in this example, its name will be ‘notif0’ according to the previously described naming conventions.

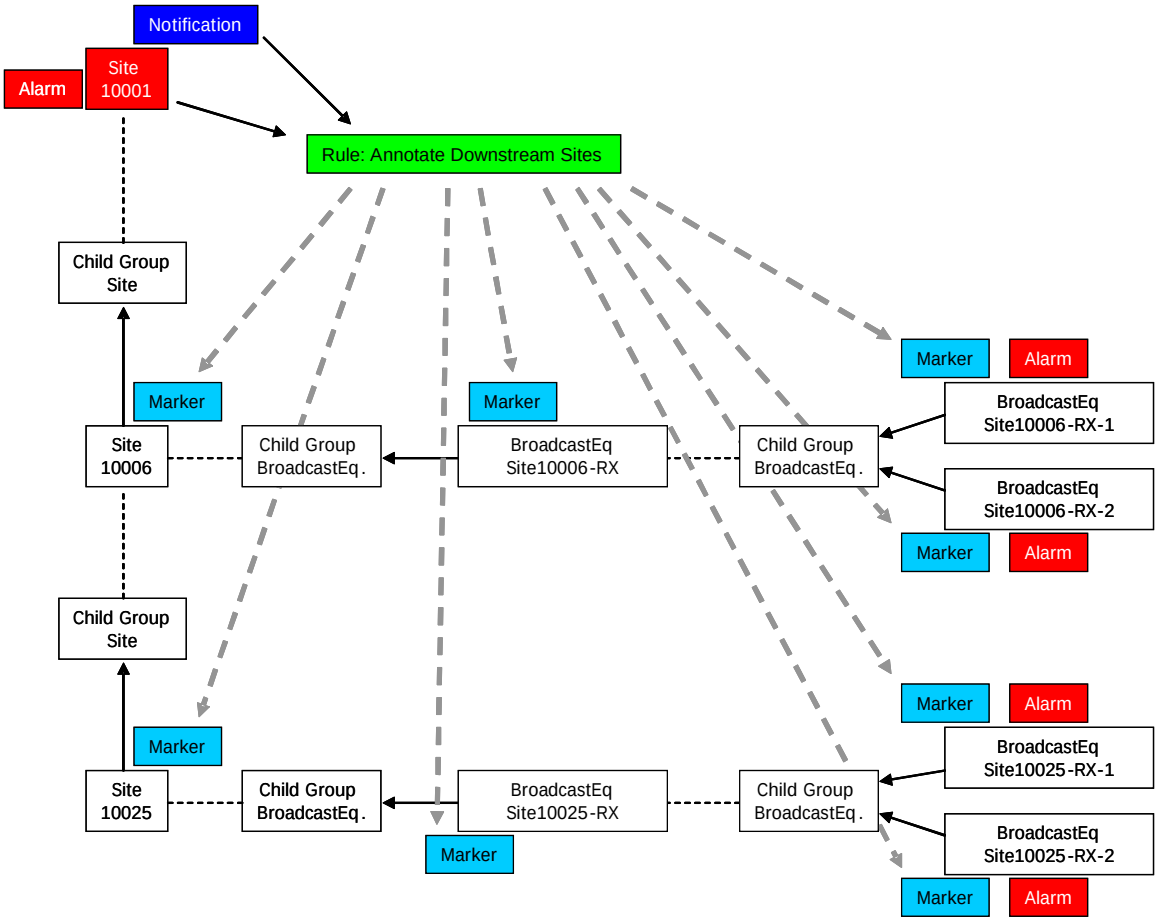
Note that the ‘Log Action to Database?’ checkbox is active. This means that the Rule teardown and associated Notification closure may be recorded in the UCA notification database if required. Given that the corresponding trigger action was recorded in the UCA notification database, it is normally prudent for the purposes of maintaining a consistent audit trail to record the clearance as well.
Once configured, this action will appear in the list of teardown actions, as shown below:



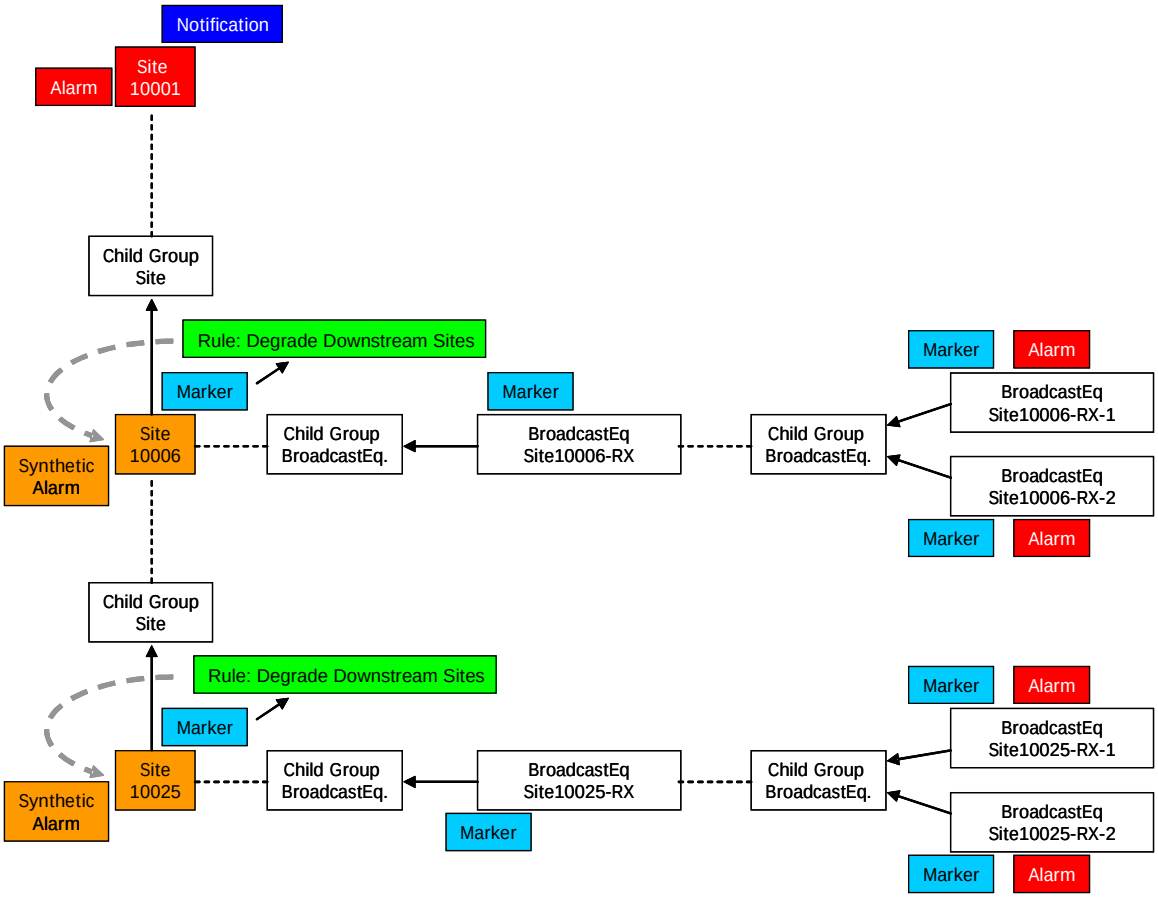
The effect of the Detect Site Problem Rule on the DTV Network example model is to attach a primary Notification to a failed Site, as illustrated in the following diagram:



To satisfy the remaining requirements for this correlation scenario, a number of additional Rules have been provided in the supplied example. These additional Rules are effectively chained together and their execution is triggered by the creation of the primary Notification.
The first of these additional Rules (Annotate Downstream Sites) attaches marker Notifications to downstream Site and Receiver Objects (Composite & Component) in anticipation of the arrival of sympathetic events, so that they may later be gathered under the primary Notification. The effect of this Rule on the DTV Network example model is shown below:

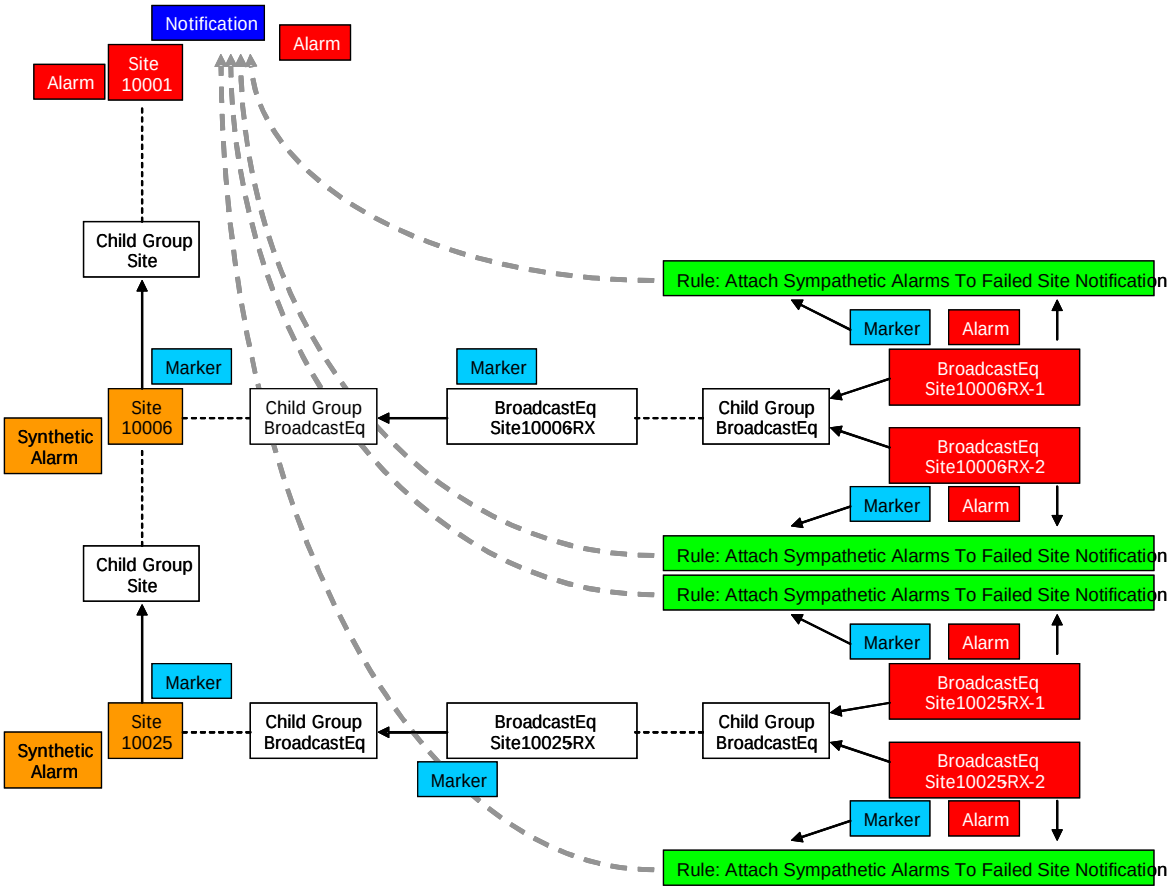


The action used to create and attach the marker Notifications onto the model is an example of a user-supplied action that has been created specifically to locate potentially affected downstream Objects. A bi-product of this discovery is that the Affected Objects list for the primary Notification is populated and this information appears in the UCA Notification Viewer when the primary Notification details are examined.
By way of a convenience to users, the next Rule (Degrade Downstream Sites) forces downstream Sites to the degraded state, so that they appear as degraded objects in the UCA Mesh Viewer. The effect of this Rule on the DTV Network example model is shown in the following diagram:

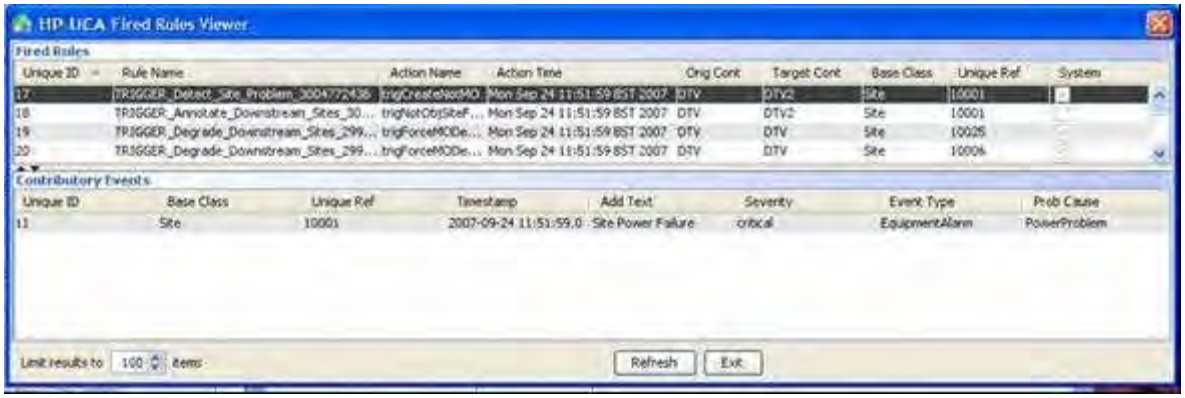


Of particular interest is the fact that the Rule is designed to operate at a single affected Site. The inference engine however will automatically identify all Sites where it is valid and the result for this example is that it will be triggered twice – once at Site 10006 and again at Site 10025.

The final Rule in this correlation scenario (Attach Sympathetic Alarms to Failed Site Notification) identifies any locations in the DTV Network example model having marker Notifications where sympathetic alarms have appeared. The associated action attaches these sympathetic events to the primary Notification. Again this is a Rule that is written to operate at a single location and in this example the inference engine automatically identifies the four Receivers on which sympathetic events are attached. Again this is summarised by the following DTV Network example model:

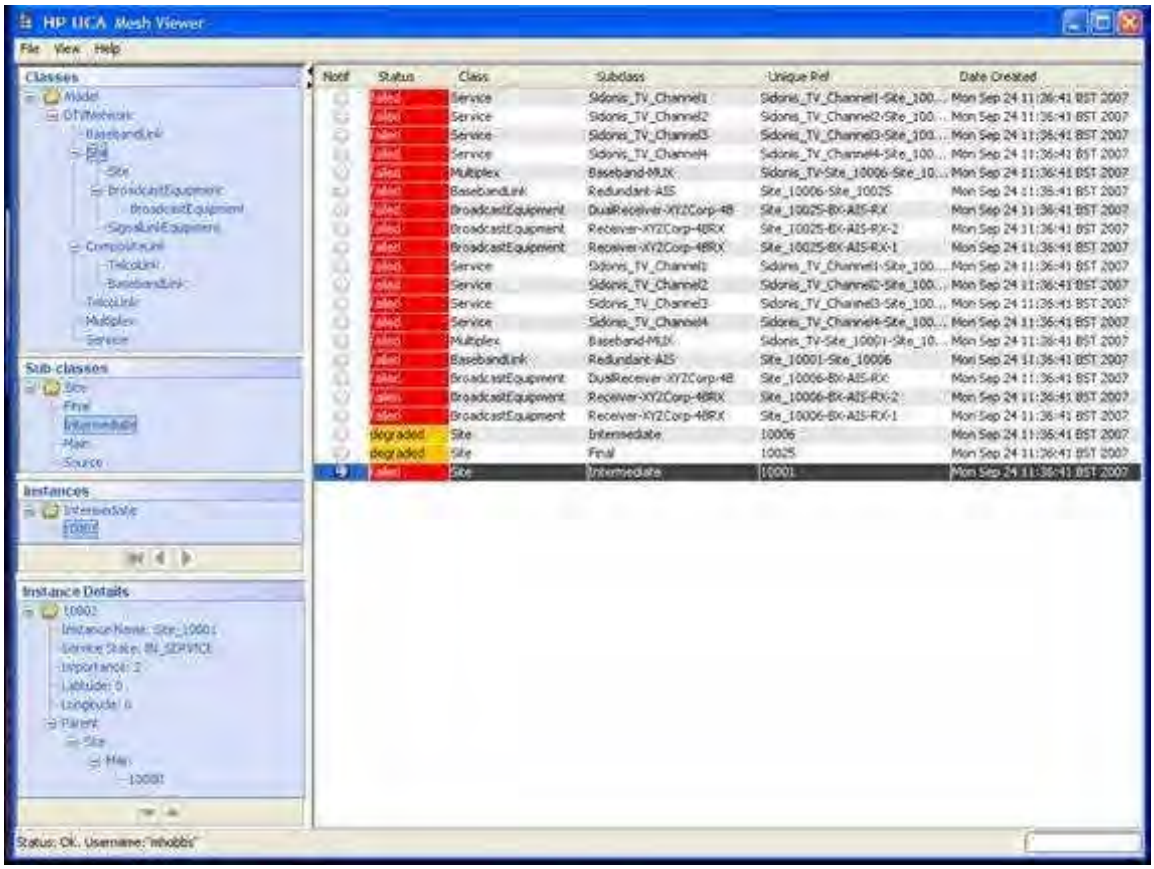


The results of this correlation scenario are visible on various UCA user interfaces.
Of particular interest to a scenario developer is the UCA Fired Rules Viewer. As long as logging to the UCA notification database has been enabled for the actions executed, the time-ordered sequence of individual Rule actions is available, as shown below for this example.

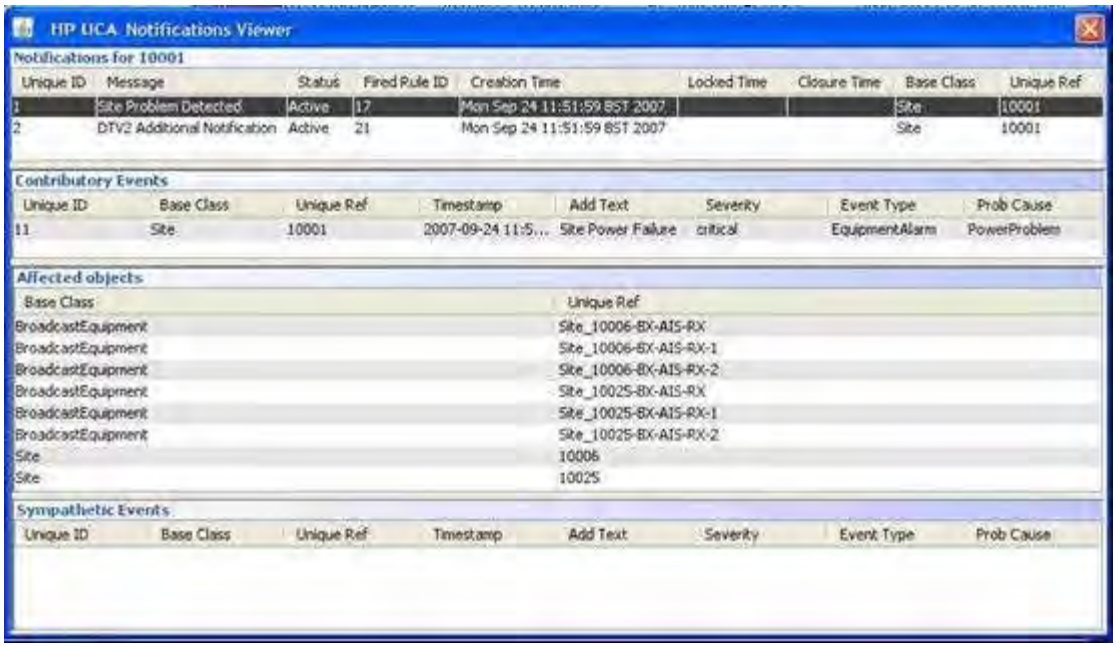


While this feature provides an in-depth view of the actions execution sequence, it incurs a processing overhead that may in certain circumstances prove onerous. The recommended use of this feature is to enable action logging as required only during the correlation development phase. Once deployed into a production environment, action logging should be scaled back to a level where it provides sufficient information to satisfy auditing requirements.

The current state of Objects affected by received events or modified by Rule actions is shown in the UCA Mesh Viewer. This information is likely to be of interest to both a Rule developer and a network operator as it gives a near real-time view of the state of the monitored network, augmented by forced state changes provided by correlation scenarios. For the DTV Network example provided with UCA, the Mesh Viewer output is shown below:



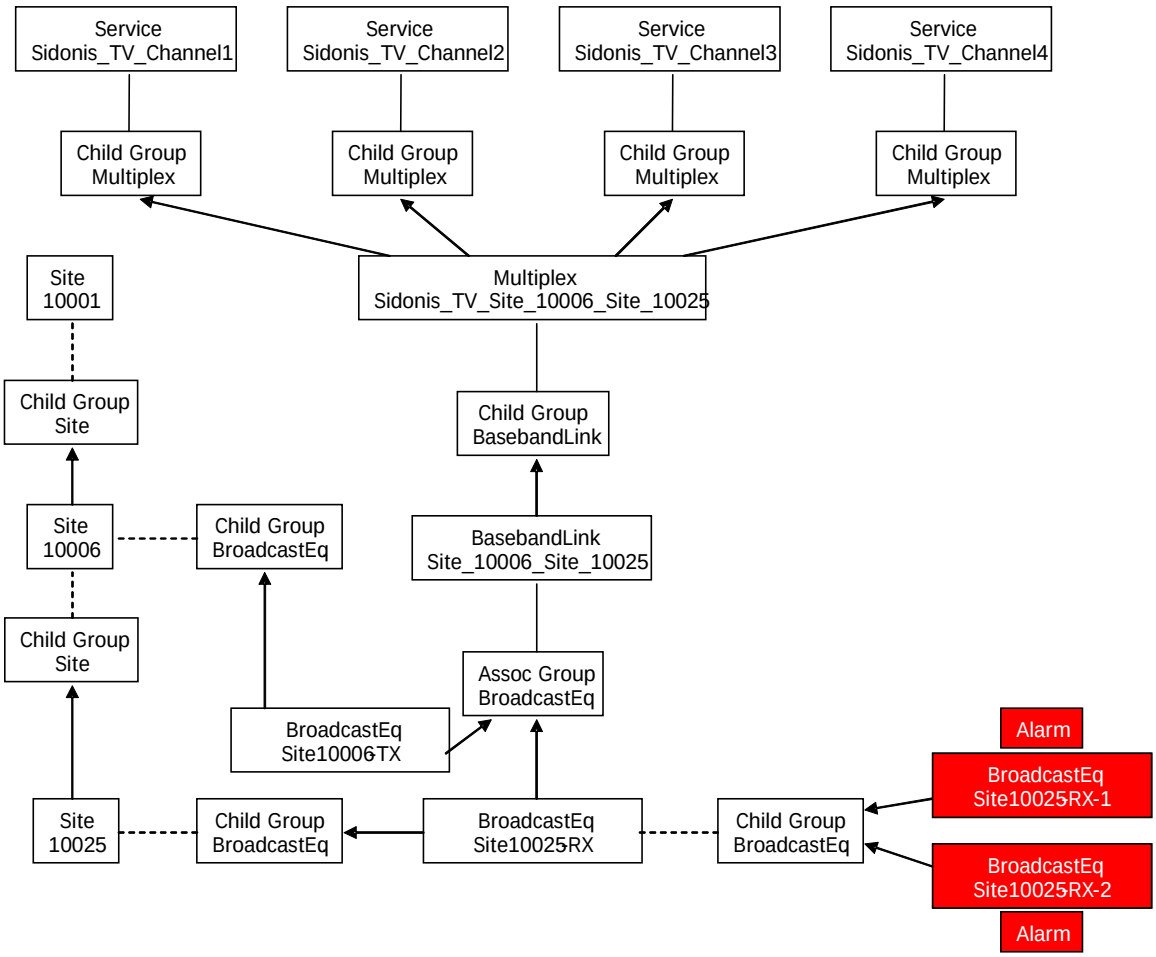
In this example, the states of Site 10001 and the individual Receivers at Sites 10006 & 10025 have been affected by the received events (recall their mappings were configured to cause the target object to adopt the failed state). In response to the 'Degrade Downstream Site' Rule described above, the states of Site 10006 & Site 10025 have been modified to degraded. This reflects the fact that both of these downstream Sites are effectively 'off-line' because Site 10001 has failed, but they have suffered no actual failure themselves.
The blue arrow icon next to Site 10001 in the above display reports that one or more notifications are present against this object. Using the UCA Notification Viewer, these notifications can be examined, as shown below:



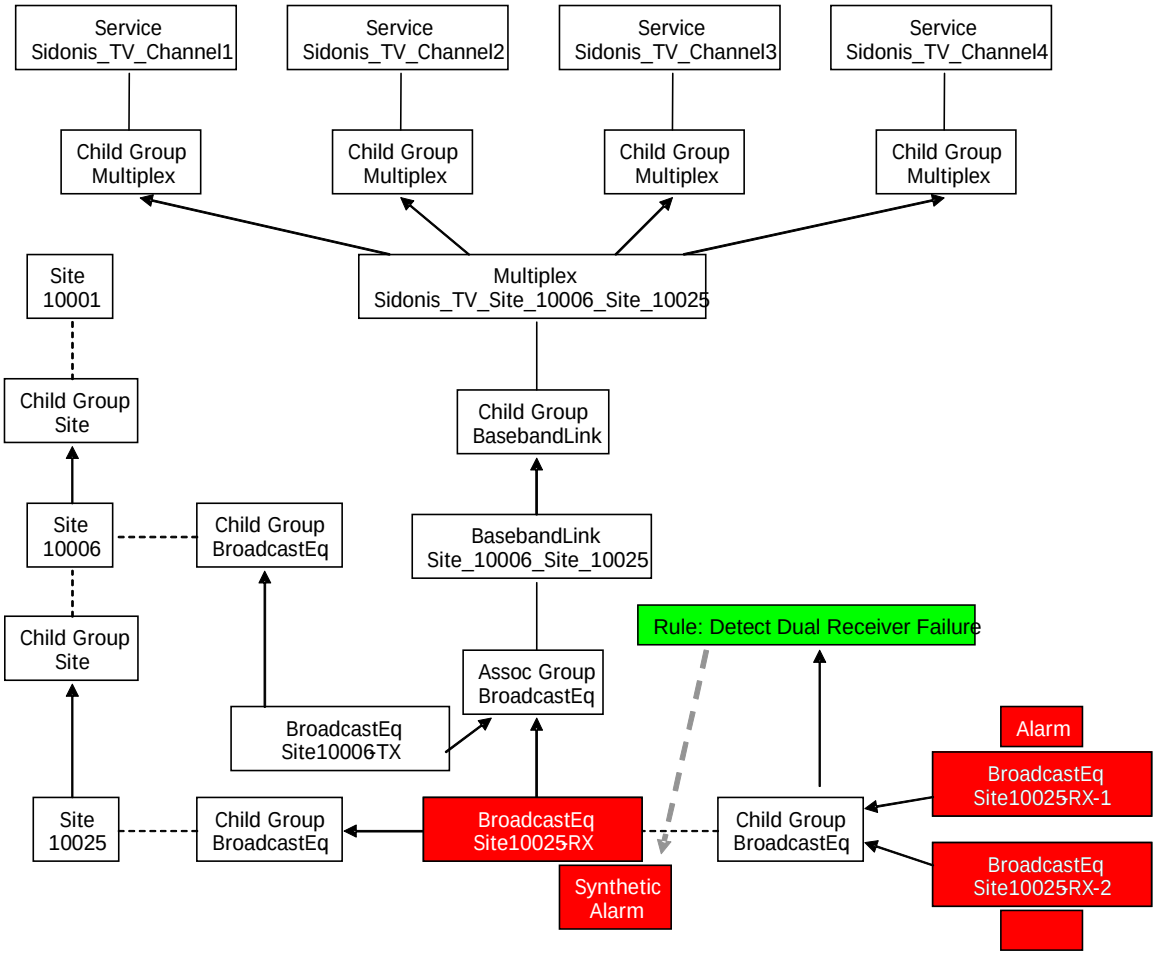
The notification created by the Rules in the correlation scenario described above has been selected in this screenshot and as a result, the contributory events and affected objects are also displayed.

9.2.2 Correlation Scenario - DTV Service Impact

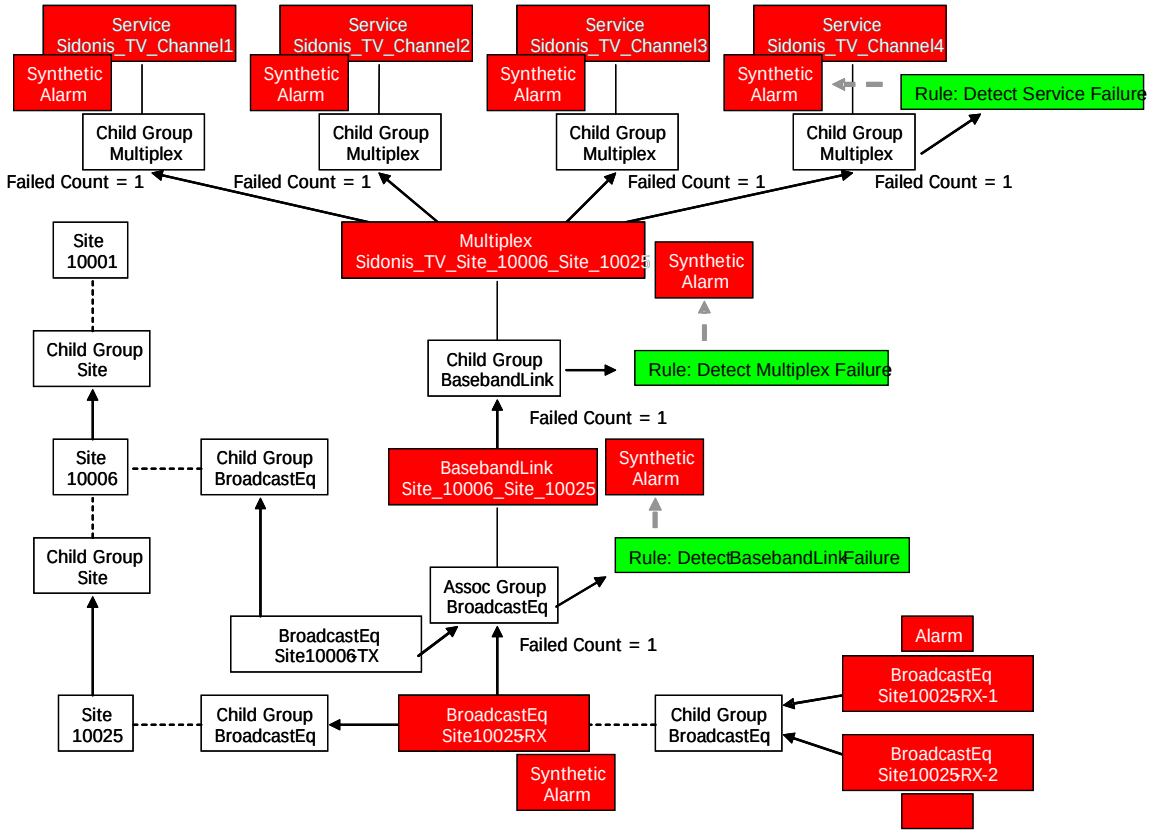
The DTV Network example provided with UCA includes a DTV Service Impact correlation scenario that operates concurrently with (but independently from) the DTV Site Failure correlation scenario.
The DTV Service Impact correlation scenario is required to detect when the Broadcast Equipment at either end of a Baseband Link between two Sites has failed, thereby affecting the state of DTV services broadcast from the subtending Site. In order to provide a more realistic example, the DTV Network model allows for redundancy in transmitting and receiving equipment at each end of the Baseband Link by modelling its endpoints as a redundant entity e.g. a Composite Receiver is built from one or more child Receivers. The result is that failure of a Broadcast Equipment endpoint only occurs when all of the child components have failed.
In the included example, the DTV Service Impact correlation scenario is triggered by the same individual Receiver failure events that are regarded as sympathetic events by the DTV Site Failure correlation scenario, however for the former they are regarded as contributory events. This illustrates the fact that carefully designed concurrent scenarios can utilise the same events for different purposes without conflict. Further, the DTV Service Impact correlation scenario implementation is implemented in a location independent manner so that it can operate equally well for transmitter and receiver failures.
The DTV Network example model before any correlation Rules have triggered is shown in the following diagram, including the events attached to the Receiver objects.



The first Rule to trigger in this correlation scenario detects failure of the composite Broadcast Equipment at one end of the Baseband Link. Because 100% of the child Receivers has failed at Site 10025, the action forces the Composite Receiver to fail by associating a synthetic failure event. This is illustrated in the following DTV Network model diagram.



Consideration of this scenario in fact shows to be an example of the Physical-Logical Vee design pattern described earlier. Physical equipment failures, in this case Receivers and in turn their containing Composite Receiver; cause the associate Baseband Link to fail. This is in turn propagated up through the logical branch of the DTV Network Model to the DTV Services. This is illustrated in the following diagram.



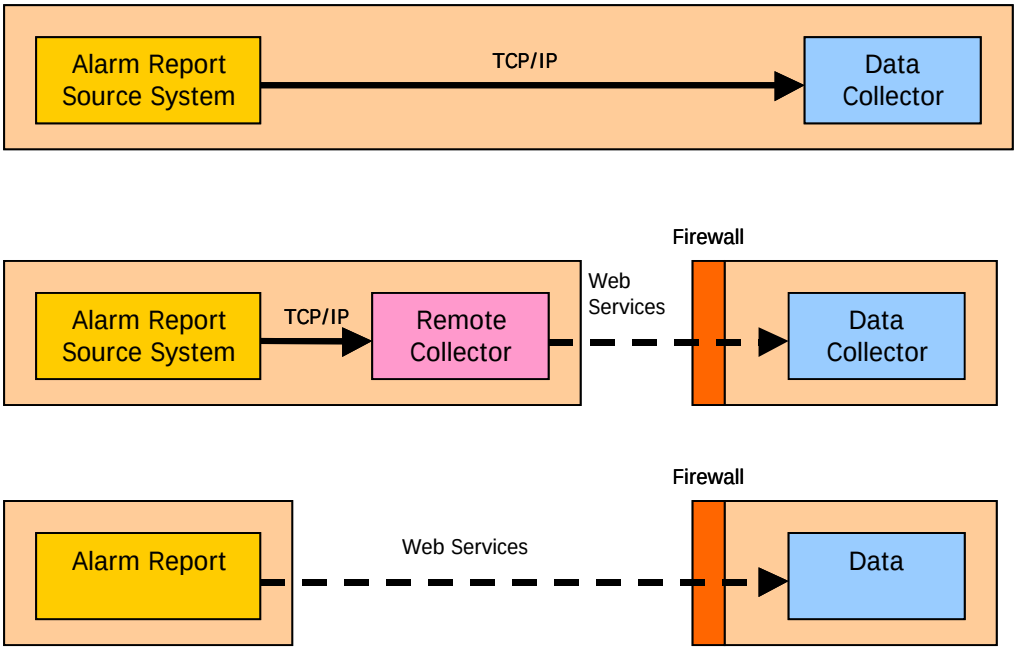
9.2.3 Correlation Scenario - DTV Maintenance

This scenario differs from the previous examples for a number of reasons:

- The purpose of each Rule is to retract components of the state mesh in the normal state from the working memory associated with the DTV context.
- Each Rule possesses only teardown conditions and actions and operates at priority 0, allowing higher priority Rules to evaluate normal state mesh components before they are retracted.
- For syntactic reasons (a scenario must have at least one filter and map), this scenario includes a ‘default’ filter and map. In practice, the conditions chosen for each are unlikely to occur in practice and are simply chosen to provide a ‘placeholder’ filter and map chain. No alarm reports are intended to pass the default filter and map chain.

Chapter 10 Alarm Interfaces

UCA offers a number of options to gather alarm reports, illustrated in the following diagram.



If the alarm report source system is able to obtain TCP/IP connectivity to the platform on which the UCA Data Collector executes, then alarm reports may be delivered directly via a socket interface.
If the alarm report source system is remotely located from the UCA platform or a firewall exists between the two systems, then the Remote Collector in combination with the Data Collector may be used. The Remote Collector connects to the Data Collector using an XMLRPC Web Services connection. The remote source then connects to the Remote Collector via a TCP/IP socket as normal.
Alternatively, a direct XMLRPC Web Services connection may be opened by the source system to the Data Collector.

10.1 Local Socket Interface

The Data Collector supports a TCP/IP socket interface and listens for incoming connections from alarm report sources on a pre-defined port (by default 6666, but this may be configured in the **uca.properties** file).
The Data Collector functions as a socket server and the remote system must be configured to connect as a socket client. The remote system is responsible for establishing and maintaining the connection with the Data Collector.

10.2 Web Service Interface

The Data Collector also supports a web service interface. One advantage of establishing a Web Services connection is that it may more easily traverse a firewall. It also provides for the possibility of gathering alarm report information across an intranet or even the Internet.
To maintain compatibility with an existing socket interface implementation, UCA provides a Remote Collector that implements a TCP/IP socket to Web Service proxy adapter. If the Remote Collector is executed on a platform accessible to the source system, it automatically establishes a Web Services connection to the Data Collector. The source system then connects to the TCP/IP socket interface provided by the Remote Collector as described in the previous section. Details of configuring and launching the Remote Collector are provided in the API Related documentation.

10.3 Supported Event Messages

Many network management systems raise alarm reports with a given severity (eg. critical, major, minor). When the alarm condition ceases, the network management system then raises an identical alarm report but with severity ‘cleared’ to indicate that the problem condition has finished. However, some systems do not produce clear alarm reports in this way – they raise a ‘state change’ type of alarm report that simply contains the id of the original alarm report to be cleared. UCA accommodates both types of alarm clearance mechanism by supporting two forms of input message, relating to:

- Alarm creation reports (for all alarm severities including ‘cleared’).
- Alarm state change reports (where the new state is ‘terminated’).

For both cases, the input data received by UCA is in the form of an XML message stream. The stream consists of a series of messages enclosed in XML <Event> …. </Event> tags. The transmitted XML data stream must not contain any XML header information and since it is streamed, it is not dynamically associated with any schema or DTD document. The tags within an alarm report are based on the alarm fields defined in the ITU-T X.733 specification. User-defined tags, also called user-defined alarm fields, are also supported and are described in the subsequent section.

10.3.1 User-defined event fields

User-defined event fields are defined in the file `filterfield.properties` and must have a “user.” prefix. For example, the user-defined field, resourceText, is defined as follows:

```
user.resourceText : String,conditionkey.string,valuekey.default,true
```

The property value in this case defines the type, condition key, value key and editable flag for the user-defined type ‘resourceText’.

An example event message would contain the configured event field, thus:

```
<Event>
      ...
     <resourceText>Further operational information.</resourceText>
     ...
</Event>
```

10.3.2 Event Message

Each event message consists of a stream of XML data formatted as follows. The order of the tags within the <Event>…</Event> tags is unimportant:

```
<Event>
      <eventRank></eventRank>
      <systemClass></systemClass>
      <systemInstance></systemInstance>
      <eventId></eventId>
      <dataType></dataType>
      <originatingTime></originatingTime>
      <moClass></moClass>
      <moInstance></moInstance>
      <severity></severity>
      <alarmType></alarmType>
      <probableCause></probableCause>
      <specificProblems></specificProblems>
      <additionalText></additionalText>
      <additionalTextTag1></additionalTextTag1>
      <additionalTextTag2></additionalTextTag2>
      <additionalTextTag3></additionalTextTag3>
      <additionalTextTag4></additionalTextTag4>
      <additionalTextTag5></additionalTextTag5>
      <additionalTextTag6></additionalTextTag6>
</Event>
```

NOTE: All tags are case-sensitive.

The tags have the following meaning:

Tag Name	Description of Tag Value	Mandatory
eventRank	If this is a new alarm report from an external source system, then set to “**original**”. If the alarm report has resulted from an Action that UCA executed e.g. raising a root cause alarm, then the value is “**master**”. In all normal circumstances, an external alarm system should	yes

Tag Name	Description of Tag Value	Mandatory
	use "**original**".	
systemClass	The generic type of the alarm source system, e.g. "sidonis_nms" etc.	yes
systemInstance	A string that uniquely identifies the identity of the alarm source system, e.g. "v1.0.1-02".	yes
eventId	A string that uniquely identifies the alarm report ID eg "2311"	yes
dataType	This should be set to "**X.733**"	yes
originatingTime	For alarm reports that are not of 'cleared' severity, this is the time the alarm report was raised as reported by the source system. For 'cleared' severity alarm reports, the time that the alarm report was cleared on the source system. The format is "**YYYY-MM-DD hh:mm:ss**" where **DD** = day in month (1-31) **MM** = month in year (1-12) **YYYY** = year eg. 2006 **hh** = hour in day (0- 23) **mm** = minute in hour (0-59) **ss** = second in minute (0-59).	yes
moClass	The value of the managed object class associated with the alarm report e.g. "Site, or "BroadCastEquipment"	yes
moInstance	The value of the managed object instance associated with the alarm report e.g. "10006" or "Site_10006-BX-AIS-RX-2"	yes
severity	One of the ITU-T X.733 severity enumerations, namely: critical, major, minor, warning, indeterminate or cleared	yes
alarmType	One of the ITU-T X.733 alarmType enumerations, namely: communicationsAlarm, equipmentAlarm, processingAlarm, qualityOfServiceAlarm or environmentalAlarm	yes
probableCause	One of the ITU-T X.733 probableCause enumerations, namely: adapterError, applicationSubsystemFailure, bandwidthReduced, callEstablishmentError, communicationsProtocolError, communicationsSubsystemFailure, configurationOrCustomizationError, congestion, corruptData, cpuCyclesLimitExceeded, dataSetOrModemError, degradedSignal, dTE-DCEInterfaceError, enclosureDoorOpen, equipmentMalfunction, excessiveVibration, fileError, fireDetected, floodDetected, framingError, heatingOrVentilationOrCoolingSystemProblem, humidityUnacceptable, inputOutputDeviceError, inputDeviceError, lANError, leakDetected, localNodeTransmissionError, lossOfFrame, lossOfSignal, materialSupplyExhausted, multiplexerProblem, outOfMemory, outputDeviceError, performanceDegraded, powerProblem, pressureUnacceptable, processorProblem, pumpFailure, queueSizeExceeded, receiveFailure, receiverFailure, remoteNodeTransmissionError, resourceAtOrNearingCapacity, responseTimeExcessive, retransmissionRateExcessive, softwareError, softwareProgramAbnormallyTerminated, softwareProgramError, storageCapacityProblem, temperatureUnacceptable, thresholdCrossed, timingProblem, toxicLeakDetected, transmitFailure, transmitterFailure, underlyingResourceUnavailable or	yes

Tag Name	Description of Tag Value	Mandatory
	versionMismatch	
specificProblems	A text string that further qualifies the alarm problem.	no
additionalText	A text string that provides additional useful information related to the alarm. All white space and linefeed characters will be maintained. This field normally contains the 'main body' or raw text of the original alarm report raised by the alarm source system.	yes
additionalTextTag1 - 6	If used, these may be used to add any extra information to qualify the alarm report.	no

Note:

- ***If any field contains an XML meta-character such as > or < then the character or the whole field should be surrounded by <![CDATA[and]]>***
- ***No field should contain a value with single quotes i.e. a ' character.***

The following is an example section of a data stream over the UCA input interface:

```
<Event>
        .
        .
</Event>
<Event>
     <eventRank>original</eventRank>
     <systemClass>HP_nms</systemClass>
     <systemInstance>V5</systemInstance>
     <eventId>1003</eventId>
     <dataType>X.733</dataType>
     <originatingTime>2005-06-10 12:16:32</originatingTime>
     <moClass>Site</moClass>
     <moInstance>10001</moInstance>
     <severity>critical</severity>
     <alarmType>EquipmentAlarm</alarmType>
     <probableCause>PowerProblem</probableCause>
     <additionalText>Site Power Failure</additionalText>
</Event><Event>
        .
        .
</Event>
```

10.3.3 Event State Change Messages

The system supports two different kinds of event state change message: terminate and attributeValueChanged (AVC).
For a state change event, each XML message in the stream of data is formatted as follows. The order of the tags within the <Event>…</Event> tags is unimportant:

```
<Event>
      <eventRank></eventRank>
      <systemClass></systemClass>
      <systemInstance></systemInstance>
      <eventId></eventId>
      <dataType></dataType>
      <originatingTime></originatingTime>
     <updateState></updateState>
</Event>
```

NOTE: All tags are case-sensitive.
The tags have the following meaning:

Tag Name	Description of Tag Value	Mandatory
eventRank	If this is a new alarm report from an external source system, then set to "**original**". If the alarm report has	yes

Tag Name	Description of Tag Value	Mandatory
	resulted from an Action that UCA executed e.g. raising a root cause alarm, then the value is "**master**". In all normal circumstances, an external alarm system should use "**original**".	
systemClass	The generic type of the alarm source system, e.g. "sidonis_nms" etc.	yes
systemInstance	A string that uniquely identifies the identity of the alarm source system, e.g. "v1.0.1-02".	yes
eventId	A string that uniquely identifies the alarm report ID eg "2311"	yes
dataType	This should be set to "**X.733**"	yes
originatingTime	For alarm reports that are not of 'cleared' severity, this is the time the alarm report was raised as reported by the source system. For 'cleared' severity alarm reports, the time that the alarm report was cleared on the source system. The format is "**YYYY-MM-DD hh:mm:ss**" where **DD** = day in month (1-31) **MM** = month in year (1-12) **YYYY** = year eg. 2006 **hh** = hour in day (0- 23) **mm** = minute in hour (0-59) **ss** = second in minute (0-59).	yes
updateState	Either 'terminated' or 'attributeValueChanged'	yes

The following is an example section of a data stream over the UCA input interface for a terminate event:

```
<Event>
        .
        .
</Event>
<Event>
      <eventRank>original</eventRank>
      <systemClass> HP_nms </systemClass>
     <systemInstance>V5</systemInstance>
     <eventId>1003</eventId>
      <dataType>X.733</dataType>
      <originatingTime>2004-01-27 14:50:54</originatingTime>
      <updateState>terminated</updateState>
</Event>
<Event>
        .
        .
</Event>
```

The following is an example section of a data stream over the UCA input interface for an attributeValueChanged (AVC) event:

```
<Event>
        .
        .
</Event>
<Event>
      <eventRank>original</eventRank>
      <systemClass> sidonis_nms </systemClass>
     <systemInstance>V5</systemInstance>
     <eventId>1003</eventId>
      <dataType>X.733</dataType>
      <originatingTime>2004-01-27 14:50:54</originatingTime>
      <updateState>attributeValueChanged</updateState>
      <severity>major</severity>
</Event>
```

```
<Event>
        .
        .
</Event>
```

10.3.3.1 Terminate messages

The eventId field is used to locate the existing event in the database and the terminate event is reported to the associated Mesh Object or Notification.

10.3.3.2 AVC (Attribute Value Changed) messages

The eventId field is used to locate an existing event in the database and an update event is reported to the associated Mesh Object or Notification.
The following fields are available for update: severity, probableCause, specificProblems, additionalText, additionalText1, additionalText2, additionalText3, additionalText4, additionalText5, additionalText6 and any custom fields.
The original field values for these fields are also retained in the database.

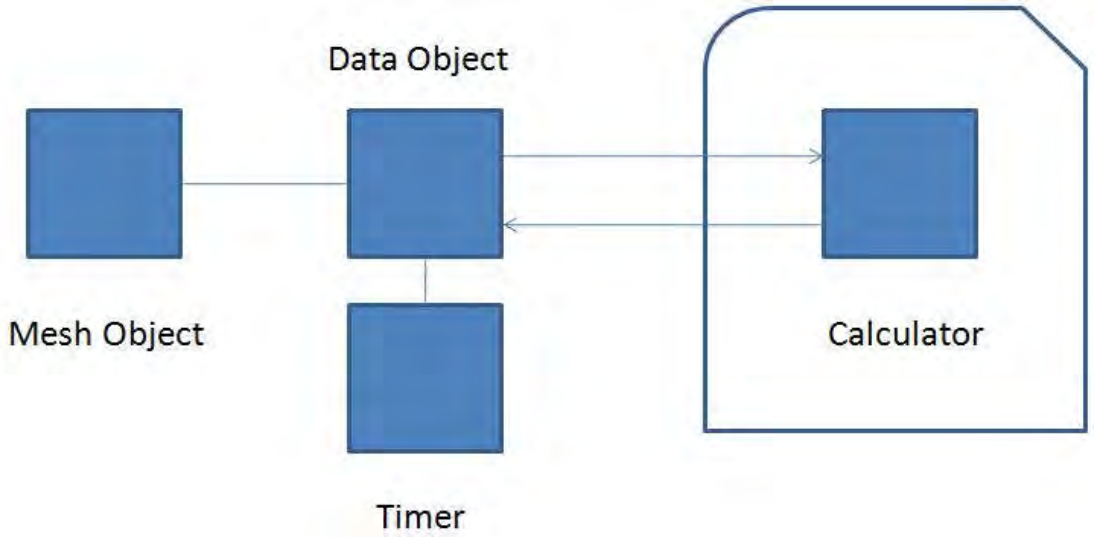
10.3.3.3 Auto-bypass filters and mappings

It is possible to configure the system such that the event state change messages bypass the filters and mapping. This means that no filter or map is required to enable state change messages. This is useful when there are few event change state messages entering the system. For high-volume scenarios, the bypass should be disabled so that unnecessary events can be filtered. The default state is disabled.
To enable the filter and mapping bypass, please set the following property in `uca.properties`:

```
automatic.update.handling       :     true
```

Chapter 11 Data and calculator objects

A data object is typically used to interrogate an external database and hold the returned raw data in a list of pre-configured key/value pairs for further processing within the system. The key/value pair will also have a type. For example, for a smart metering application we may want to store meter readings using the key 'meterReading', the value read from the database and with type 'long' i.e. a 64-bit signed integer.
One data object is instantiated per affected object. The data object is created by a custom rule trigger action.
The data object utilizes a calculator object (one per context) to perform processing on the raw data.
Through configuration it is possible to expose derived fields to the rules engine so that rules can interrogate the derived values and perform further actions.
A basic schematic is shown below:



11.1 Data Object Attributes

A data object can be viewed in the working memory by double clicking on the data object instance, as identified by its base class and unique reference. The data object attributes will be listed in the dialog box and brief descriptions of each are listed below.

11.1.1 Raw Data

This attribute consists of a list of key/value pairs which represent the raw data as populated from the external database via a RemoteHandler call and call-back mechanism. The data keys are defined in the Data Object configuration file.

11.1.2 Derived Data

This attribute consists of a list of key/value pairs which represent the derived data as populated by calculations performed on the raw data. The derived data keys are defined in the Data Object configuration file.

11.1.3 Last change reason

This is an enumeration of one of the following values: initialising, data-available, derived-data-available.
'Initialising' means that the data object has been instantiated but does not yet have any raw data.
'Data-available' means that the object has been filled with raw data.
'Derived-data-available' means that calculations have been performed on the raw data.

11.1.4 Base class

This field represents the base class of the data object.

11.1.5 Unique reference

This field represents the unique reference of the data object.

11.1.6 Timer state

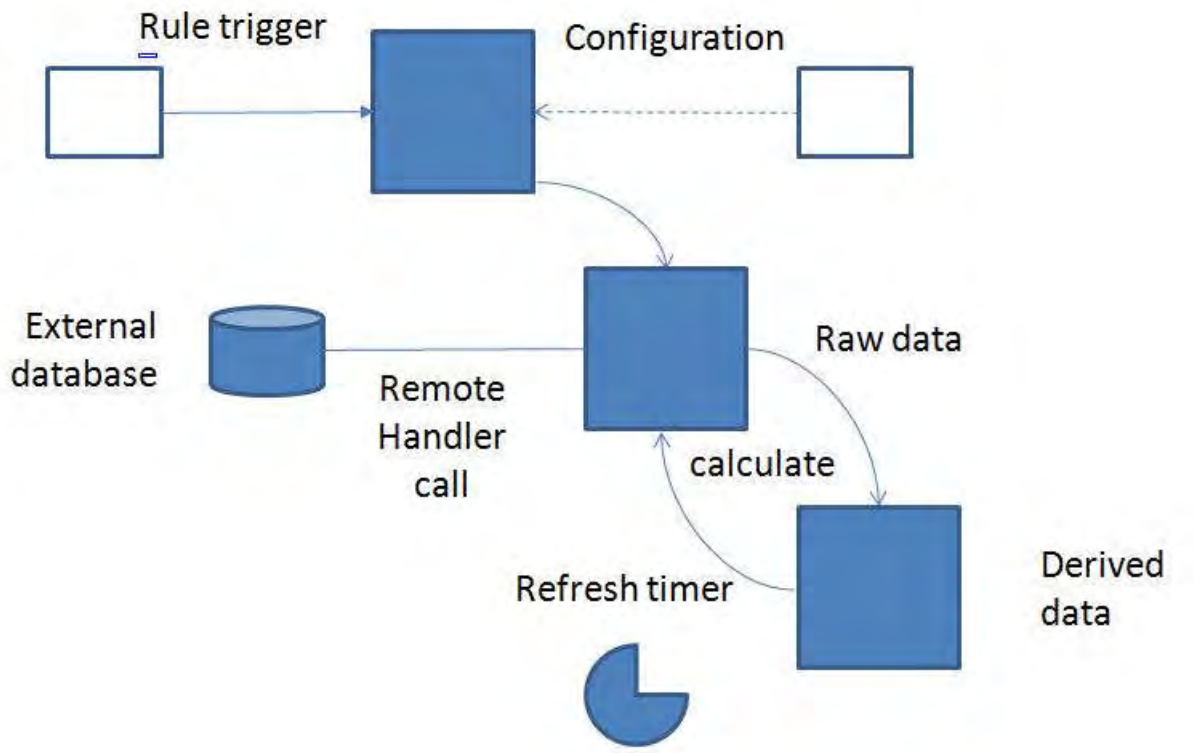
The associated timer state: an enumeration of undefined, initialised, running, suspended, expired, completed.
A refresh rule will detect the ‘expired’ state i.e. the refresh countdown has reached zero.

11.1.7 Timer state changed

A flag indicating that the timer has changed state.

11.2 Data Object Lifecycle

A schematic of the data object lifecycle is shown below:



A data object has very distinct parts to its lifecycle: initialisation followed by a cycle of data retrieval and derived data calculation/storage.

11.2.1 Initialise Data Object

11.2.1.1 Rule trigger action

The action ‘create data object’ must be inserted as a trigger action on a rule.
When the rule is actually constructed in the GUI, the type of the data object is specified. When the rule is fired, the data type will be created for the associated Mesh Object.
If a data type already exists for the Mesh Object, the action will be ignored.

11.2.1.2 Data Object Configuration

The following file snippet shows an example data object configuration (for a fictional smart meter data object):

```
<metaDataObject type="smartMeter">
     <dataMappings>
            <dataMapping from="meterValue" to="currentMeterValue"/>
      </dataMappings>

     <dataTuples>
            <tuple name="meterValue" type="long" />
            <tuple name="previousMeterValue" type="long" />
            <tuple name="timestamp" type="long"/>
            <tuple name="previousTimestamp" type="long"/>
      </dataTuples>

      <outputTuples>
            <tuple name="usageChangePercent" type="double" />
      </outputTuples>

            <dataSource name="smartMetering" user="meterUser"
      pass="meterPassword" connections="10" dbms="postgresql">
                        <driverClass>
                              org.postgresql.Driver
                        </driverClass>
                        <connectionUrl>
                              jdbc:postgresql://localhost/smartMetering
                        </connectionUrl>
            </dataSource>
</metaDataObject>
```

dataMappings element
It is possible for a database field name to a stored under a different key name using the mappings as defined in this XML section.

dataTuples element
The raw data keys as taken from the database are defined in this section of XML. Supported types are: boolean, int, long, float, double, string.

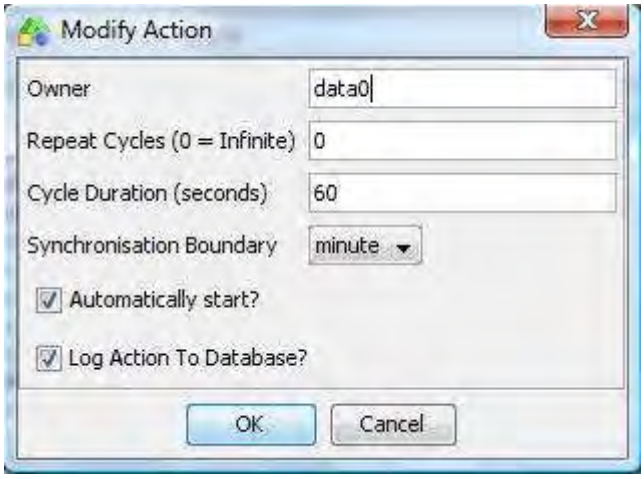
outputTuples element
The derived data keys as populated by calculations are defined in this section of XML. Supported types are: boolean, int, long, float, double, string.

dataSource element
The data source for the external database is defined in this section of XML.

Note - The configuration file can be found in `properties/data-config.xml`

11.2.1.3 Create Associated Timer

The final part of the data object initialisation is the creation of an associated timer to perform the countdown for a refresh of the raw data. This is achieved using a rule to detect when the timer state is expired. A trigger action is included in the rule to create a countdown timer to repeat infinitely i.e. until the data object is removed.
The action details are as follows:



11.2.2 Populate raw data

A rule must be created which contains the 'refresh data object' trigger action. This action will detect an expired countdown timer and make a call to the remote handler to interrogate the external database. Please refer to the Remote Handler Specification for more information on this call.
The call-back mechanism from the `RemoteHandler` will result in the sending of a `DataRefreshEvent` to the event manager, which will refresh the raw data stored in the key/value pairs.

11.2.3 Populate derived data

The derived data is populated by an action called from a rule. The action in question is the 'perform calculation' action which specifies the data object for which the calculation is to take place, and also the desired calculation name.
Multiple calculation actions can exist per rule and calculation actions can be split across many rules with different priorities. The only proviso is that the final calculation action must be preceded by a 'finish calculations' action. This action informs the data object that it can validate the derived data and be updated in the working memory.

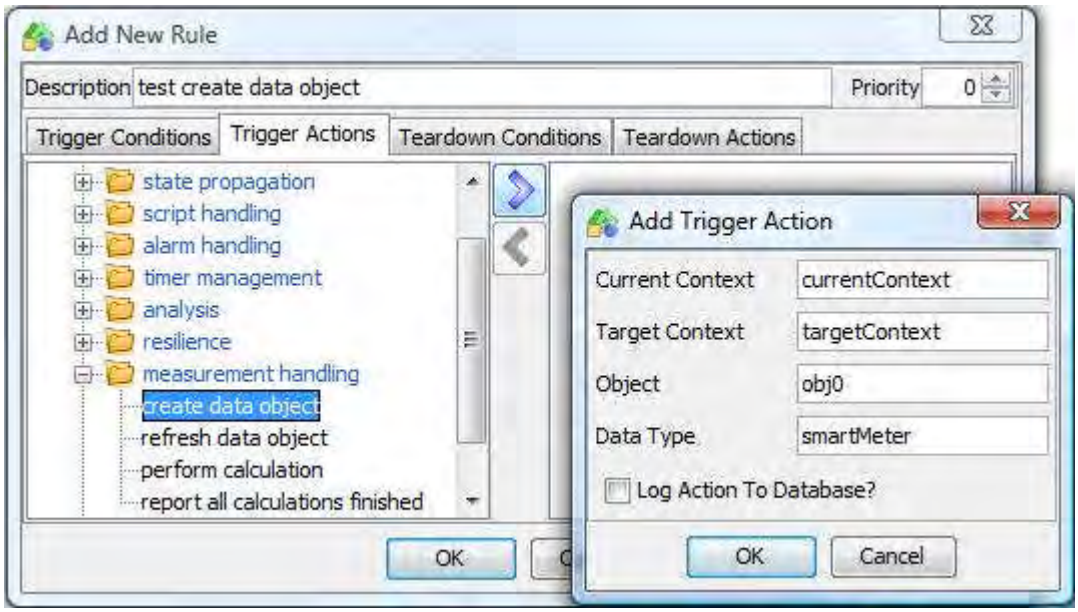
11.2.4 Data object actions

The following actions are available from the rule action dropdown list, under the category 'measurement handling':

create data object	This action is used to create a data object of the specified type, for a given Mesh Object.
refresh data object	This action is used to refresh a data object of the specified type, for a given Mesh Object.
remove data object	This action is used to remove a data object of the specified type, for a given Mesh Object.

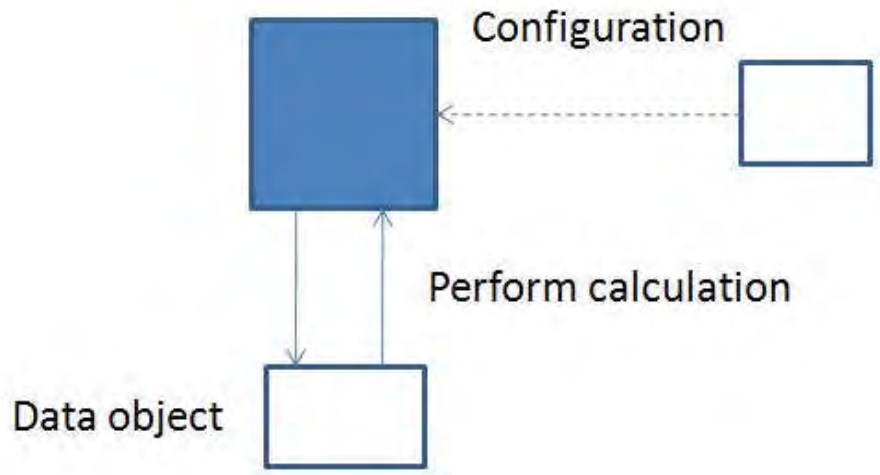
11.2.4.1 Example data object action

The screenshot below shows the create data object trigger action:



11.3 Calculator object lifecycle

A schematic of the calculator object lifecycle is shown below:



The calculator uses an expression evaluator (called 'Janino') to compile the configured expressions into byte-code for evaluation at runtime. The expressions must conform to the correct syntax to prevent compilation errors, which would be reported to the exception log at run-time.
A calculator object will perform calculations with the raw data supplied from data objects. The derived data is then stored in the data object.

11.3.1 Calculator Configuration

When the system first starts-up the calculator expressions are compiled and then held in memory for use by the calculator object in each working memory.
The following file snippet shows an example calculator expression for calculating the percentage change between two values:

```
<expression>
     <name>Calculate Usage Change</name>
      <expressionValue>
                  (meterValue/previousMeterValue)*100
      </expressionValue>
      <inputs>
            <input>
```

```
                  <name>meterValue</name>
                  <type>long</type>
            </input>
            <input>
                  <name>previousMeterValue</name>
                  <type>long</type>
            </input>
      </inputs>
      <output>
            <name>usageChangePercent</name>
            <type>double</type>
      </output>
</expression>
```

Multiple expressions can be configured in this manner in the same configuration file

11.3.1.1 name element

This XML element is the **unique** name of the calculation, which is used in the rule dialog for action ‘perform calculation’.

11.3.1.2 expressionValue element

This XML element is the actual (mathematical) expression to evaluate.

11.3.1.3 inputs element

This XML element describes the input value key-names to the expression.

11.3.1.4 output element

This XML element describes the output value key-name from the expression.

The configuration file can be found in `properties/calculator-functions.xml`

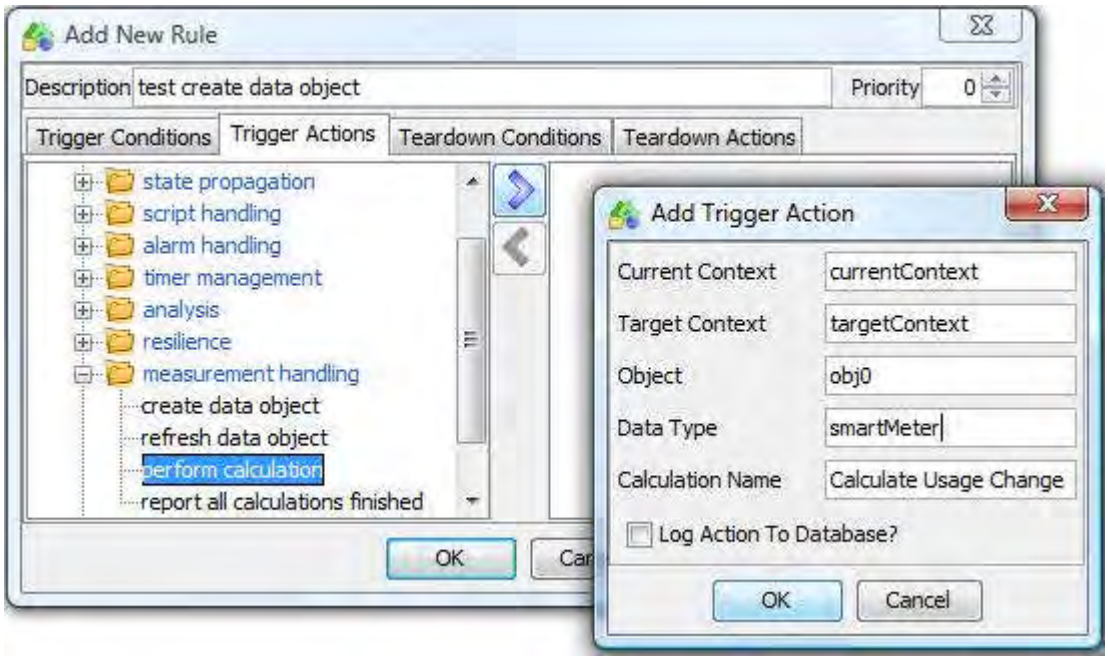
11.3.2 Calculator Actions

The following actions are available from the rule action dropdown list under the category ‘measurement handling’:

perform calculation	This action is used to perform a specified calculation on the data object for a given Mesh Object.
report all calculations finished	This action is used to report that all the calculations have finished on the data object for a given Mesh Object. Each calculation requires a trigger action to actually perform the calculation, followed by a ‘finish calculations’ action to inform the data object that all the calculations have been completed. At this point, the data object will change its state to indicate that the derived data is available for further processing.

11.3.2.1 Example calculation action

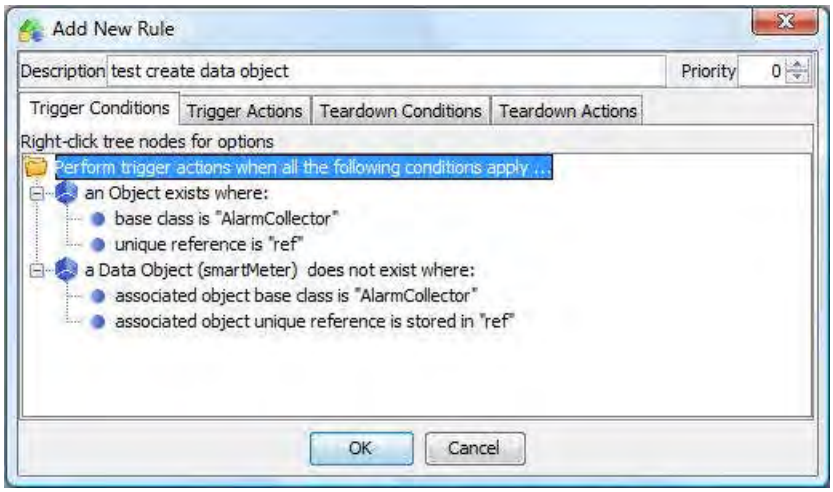
The action ‘perform calculation’ is shown below for the data type ‘smartMeter’ and calculation name ‘Calculate Usage Change’:



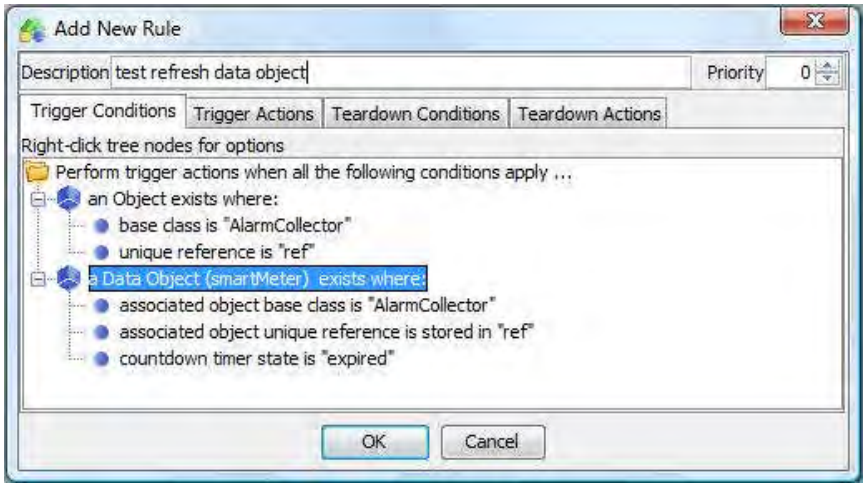
11.4 Example data object scenario

The series of screenshots shown below show an example scenario in which a data object is created (rule conditions 1), refreshed (rule conditions 2) and for which a calculation is performed (rule conditions 3).

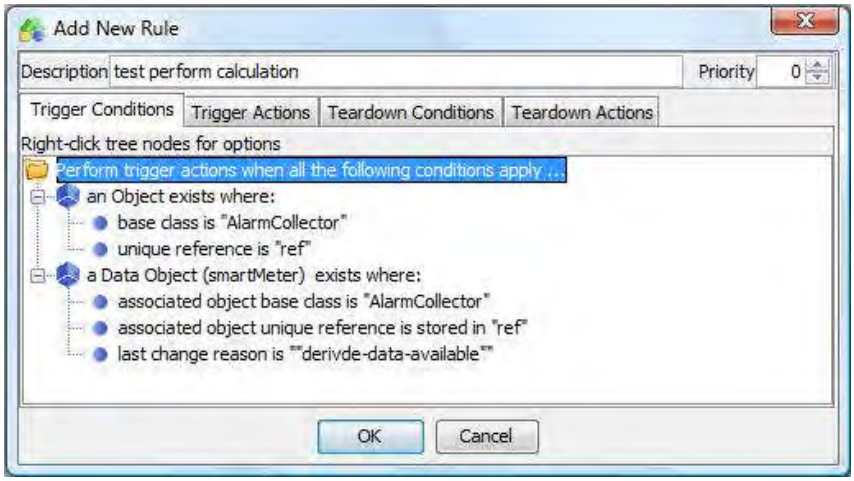
11.4.1 Example Rule Conditions for ‘create data object’



11.4.2 Example Rule Conditions for ‘refresh data object’



11.4.3 Example Rule Conditions for ‘perform calculation’



Chapter 12 Time Dependent Event Correlation

UCA offers the following capabilities and features to enable the construction of time dependent correlations:

- Time bounded event processing actions, offering comprehensive support for time dependent correlations on event streams.
- Relative and absolute time comparison operators for evaluating the time attributes model, alarm and correlation objects
- Independently controllable, countdown Timer objects (one per model or correlation object)

In addition, UCA includes sophisticated time compression algorithms for providing rapid resynchronization with event sources while maintaining the accuracy of both existing and historical time-dependent correlations.
For correct operation of a resilient UCA configuration, it is important that the system time clocks of both servers are closely aligned. For this reason, it is essential to make use of an operating system time synchronization protocol e.g. NTP.
The following sections describe each of the time dependent correlation features.

12.1 Relative and absolute time comparison operators

UCA provides a comprehensive set of comparison operators to evaluate absolute date and/or time (Date attributes) of model, alarm and correlation objects against the current UCA 'clock' time (itself a Date) or relative to another date and/or time. Each use of a time comparison operator is re-evaluated once a second until the object is retracted or the condition is satisfied.
UCA 'clock' time is not the system hardware clock. In fact it is implemented as an event driven software clock with a granularity of one second and is advanced by internal 'tick' messages generated by the system hardware clock. This implies that under circumstances, the 'clock' time may lag behind actual time as measured by the system clock, in particular where event buffering occurs. This does not affect the accuracy of the time dependent correlations because they are driven by the UCA 'clock' and eventually each 'tick' message will be processed allowing apparent and actual time to be re-aligned. At any time, the 'clock' time (referred to as 'apparent' time) and the actual time may be examined using the Time object in the Working Memory Viewer.
It should also be noted that during resynchronization processes involving event replay in 'compressed time', the current UCA 'clock' time will be adjusted to an earlier time and then continuously advanced by the system to establish historically accurate time dependent correlations for the resynchronizing event source. During this process, all other time dependent correlations for other event sources will be 'frozen' (to preserve their accuracy as the UCA 'clock' is adjusted).
The following table lists the time comparison operators and illustrates their use with the "Creation Time" attribute of a Notification although they may be used with any attribute of the Date type. Where <Variable> is specified, this implies that a previous 'stored in' assignment operation has been carried out to initialise the variable with another Date value or an integer offset value in seconds.

Operator	Use
is before	[Creation Time] is before <Absolute Time>
is after	[Creation Time] is after <Absolute Time>
plus offset is older than current time	[Creation Time] plus offset <x seconds> is older than current time
plus offset is younger than current time	[Creation Time] plus offset <x seconds> is younger than current time
minus offset is older than current time	[Creation Time] minus offset <x seconds> is older than current time
minus offset is younger than current time	[Creation Time] minus offset <x seconds> is younger than current time
is older than value in	[Creation Time] is older than value in <Variable>

is younger than value in	[Creation Time] is younger than value in <Variable>
plus offset (in variable) is older than current time	[Creation Time] plus offset in <Variable> is older than current time
plus offset (in variable) is younger than current time	[Creation Time] plus offset in <Variable> is younger than current time
minus offset (in variable) is older than current time	[Creation Time] minus offset in <Variable> is older than current time
minus offset (in variable) is younger than current time	[Creation Time] minus offset in <Variable> is younger than current time

12.2 Countdown Timers

UCA supports the concept of a countdown Timer object that may be dynamically created and attached to objects using rule actions. Each global (System), model (Mesh Object & Child/AssociateGroups) and correlation (Notification, Script, Data & Calculation) object may have a single Timer object attached to them. Note however that the System object timer is reserved for use with the Resilience package and is therefore not normally available for user-defined correlations.
Each Timer object operates with a granularity of one second and is driven by the UCA 'clock' with the implications described in the previous section.
Each model or correlation object is provided with two attributes that allow an associated Timer to be used in conjunction with it:

- An enumerated current timer state (undefined means that the Timer has not been created)
- A boolean timer update flag reporting if the last update applied to the object was a timer state change.

A typical use is to construct a rule that waits for the Timer associated with an object to adopt a particular state, although this must always be guarded with an additional test on the timer update flag to prevent unwanted rule firings. The timer update flag is necessary because any update to an object in a Working Memory context effectively refreshes all of the values of that object. Correct use of the update flag allows a user to distinguish between a timer state change and any other attribute change on that object.
Timers are created, maintained and destroyed by rule actions and their existence and current state can be examined via the list maintained by the global time object visible in the Working Memory Viewer.
Timers consume system resources and should be used only when necessary.
Timers have the following properties:

- They are driven by the UCA 'clock' with a granularity of one second and as a result their first cycle may last between N-1 and N seconds (where N is the timer period). Subsequent cycles will last N seconds.
- They are capable of operating in 'one-shot', counted (i.e. they time-out N times) or infinitely repeating modes.
- They may be created and then started automatically or manually
- They may be suspended, resumed, stopped and re-initialised
- They can exist in each of the following states:
    - Undefined – a Timer has not been defined for the owning object
    - Initialised – a Timer has been defined but has not yet been started or has been re-initialised
    - Running- a defined Timer has been started
    - Suspended – a previously running Timer has been temporarily suspended
    - Expired – a running Timer has reached the end of its current cycle and timed-out or has been stopped
    - Completed – a one-shot or counted Timer has exhausted the number of operating cycles or has been stopped
- Their start times may be aligned to the following time boundaries:
    - Unaligned – in fact aligned to the one second boundaries defined by the UCA 'clock'

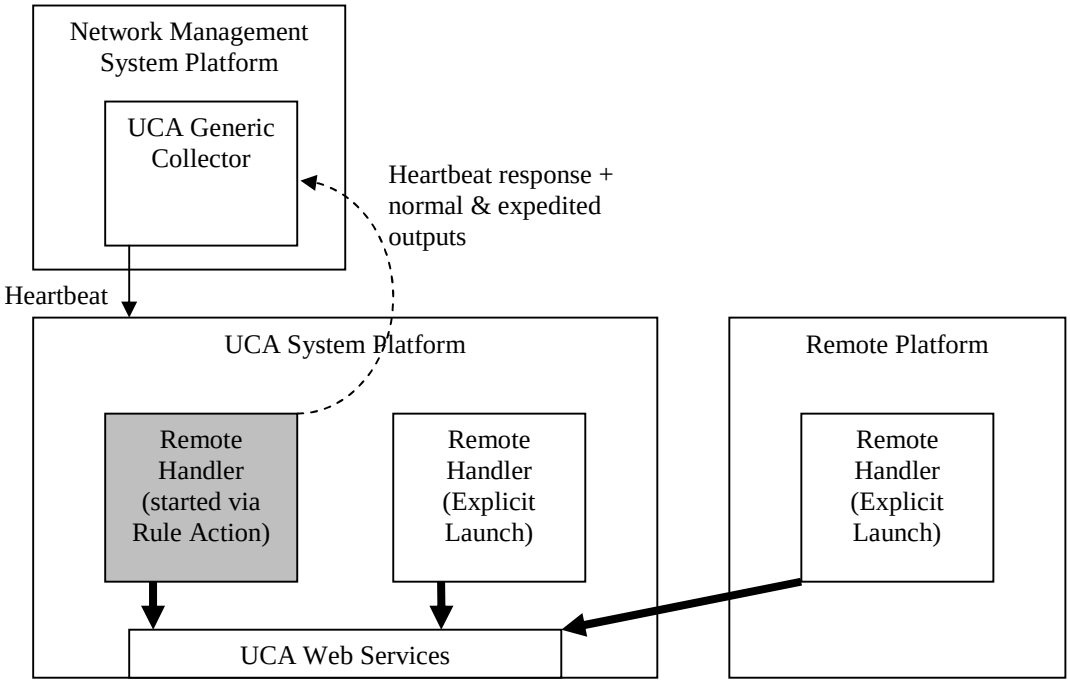
- Minute – aligned to minute boundaries, implying that the first cycle will be truncated to incur a time-out at the next minute boundary
- Hour – aligned to hour boundaries, implying that the first cycle will be truncated to incur a time-out at the next hour boundary
- Day - aligned to day boundaries, implying that the first cycle will be truncated to incur a time-out at the next day boundary

A comprehensive description of the facilities offered by Timers is contained in the section describing Time related actions later in this guide.

12.3 System Operating Modes

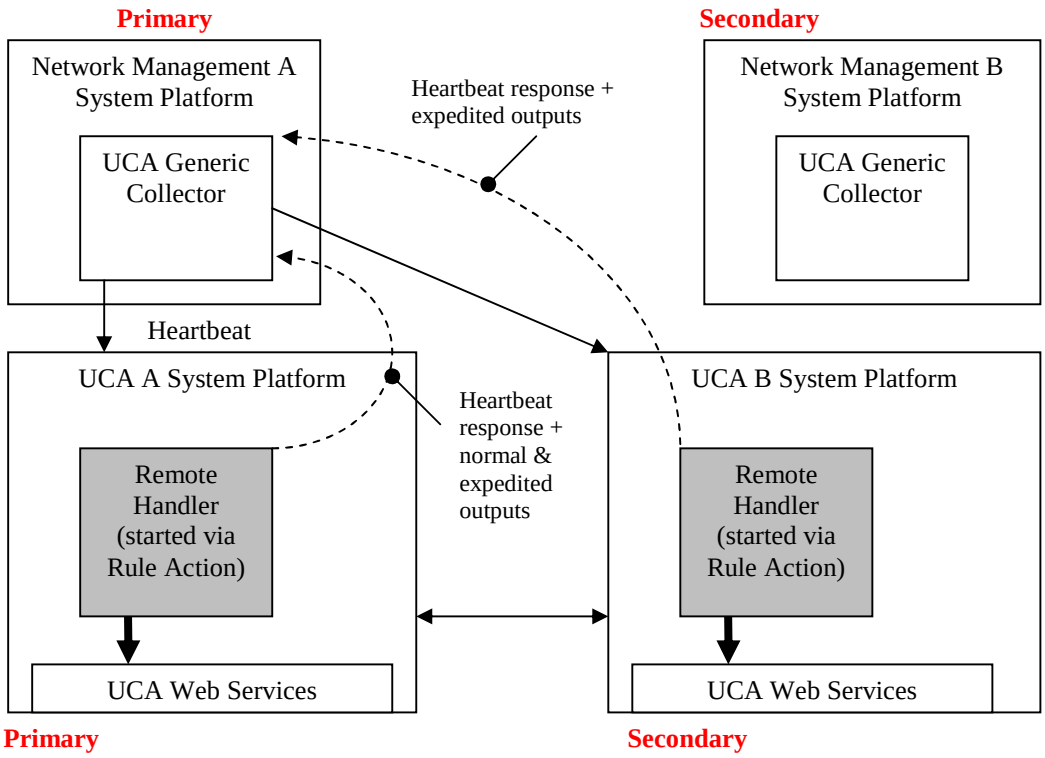
12.3.1 Standalone Mode

The following diagram illustrates UCA operating in a standalone configuration. Note that UCA may be operated in standalone configuration with or without the resilience heartbeat generated by the UCA Generic Collector. The current operating mode is set using the system.mode property in the uca.properties file. Detailed descriptions of the Remote Handler and Generic Collector are provided in the UCA Remote Handler Interface and Generic Collector Interface specifications respectively.

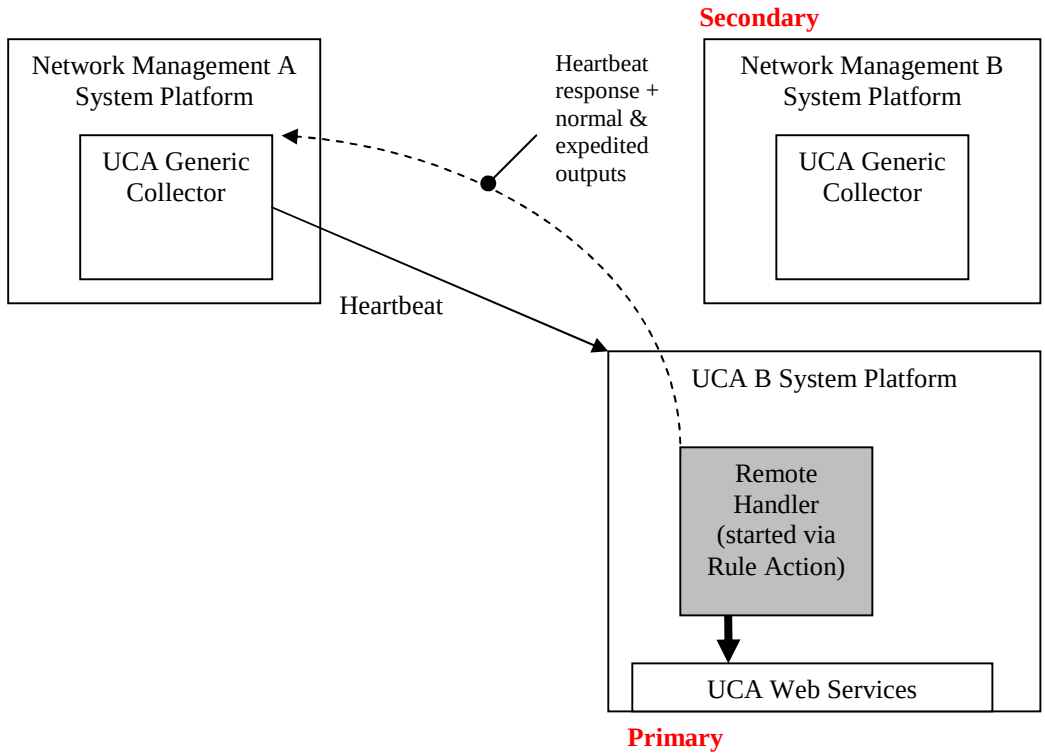


12.3.2 Resilient Mode

The following diagram illustrates UCA operating in a resilient configuration. In this example, NMS platform A is the primary and UCA platforms A & B form a resilient primary/secondary 'hot standby pair'. Remote Handlers used in a resilient configuration are normally run via Resilience Package rule actions.
The Remote Handler running on the primary UCA machine is normally operated with outputs enabled (allowing communication with the primary NMS), while that on the secondary is normally operated with outputs disabled (although expedited alarms reporting for example local platform problems may still be sent to the primary NMS). Remote Handlers running on both primary and secondary UCA machines will normally be connected to the UCA Generic Collector on the primary NMS platform, allowing each system to report an individual heartbeat response.



If a UCA failover occurs, the above configuration is modified to enable outputs from the Remote Handler on the new primary UCA platform, as shown below.



If an NMS failover occurs, each UCA instance expects the new NMS primary system to start a new instance of the UCA Generic Collector. Rules in the Resilience package automatically detect the new heartbeat source and will issue instructions to the Remote Handler instance to close the existing connection to the old Generic Collector and attach to the new Generic Collector.

Chapter 13 Resynchronization with Event Sources

13.1 Event Resynchronization

In certain operating configurations, it is important for a UCA server to undergo a process of resynchronization with one or more event sources e.g. an NMS. Resynchronization usually involves retrieving copies of all outstanding events from a source and then replaying them to re-establish the current event state. Depending on the type of correlation required, resynchronization may involve additional processing to resolve differences between the source and the prior event history stored in the UCA server Event database.

Typical scenarios where resynchronization may be required are:

- A UCA Primary or Standalone server is started for the first time. In this situation, the server will have no prior event history and may need to resynchronize with multiple external event sources. Depending on the type of correlation required, it may be necessary to replay the resynchronization events in 'compressed time' to re-establish and maintain the correct temporal correlations. 'Compressed time' event replay is a technique whereby for a given event source, the UCA 'clock' is set back to just before the first resynchronization event and then events in the resynchronization stream are replayed as fast as possible (the UCA 'clock'being automatically advanced during this process). In this way, temporal correlations are correctly handled without the delay involved in replaying events at their original delivery times and the mechanism ensures that events from other sources and associated correlations remain unaffected. Alternatively, 'compressed time' event replay may be dispensed with in situations where strict accuracy of temporal correlations is not required or a minor variation from expected behavior can be tolerated on startup e.g. stream-based correlations.

- A UCA Secondary server (re)connects to a UCA Primary server in a hot standby resilient configuration. In this situation, sophisticated inter-server resynchronization with 'compressed time' event replay and 'ID matching' is necessary to establish and maintain a common view of current correlations on both servers. In essence, all of the existing event and correlation knowledge on the Primary server is copied to the Secondary server and the resynchronization process ensures (as far as possible) that both Primary and Secondary servers present the same correlation views on completion. 'ID matching' is a technique employed to ensure that the same correlation artifacts e.g. Notifications, have the same unique identifiers on both servers. This is done in an attempt to make UCA failover seamless with regard to the event sources. Once synchronized, both servers are then driven independently by dual outputs from a single Generic Collector.

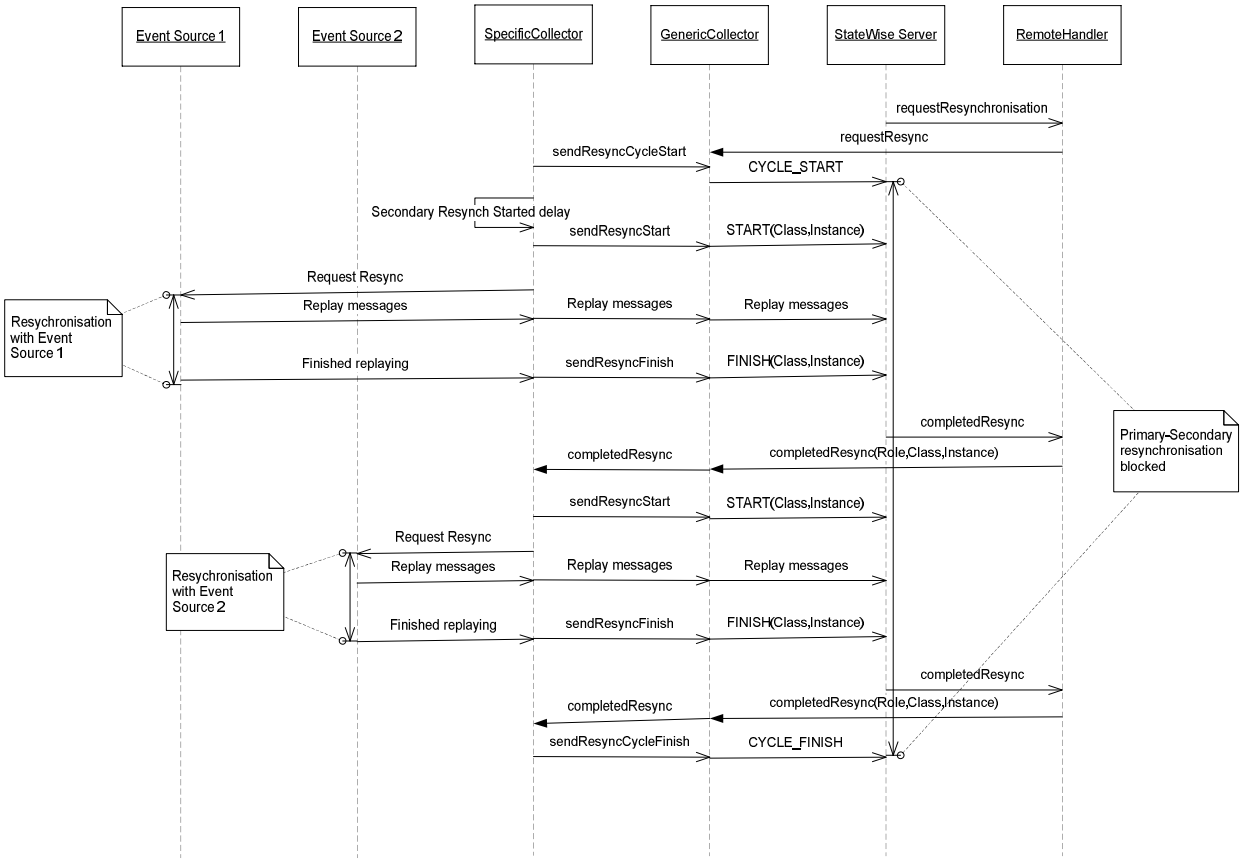
- When a UCA server (Primary, Secondary or Standalone) re-connects to an event source, either following failure and re-establishment of a particular communications link or restart of the event source system. Again, this process may optionally involve 'compressed time' event replay to re-establish and maintain the correct temporal correlations.

Resynchronization processing is handled automatically by built-in functionality in the UCA servers although it is the integrators responsibility to ensure that a Generic Collector specialization interfaces to and manages individual event sources and requests UCA to deliver the required behaviour.
Where a hot-standby resilient configuration is required, an optional package of rules (the Resilience Package) is required to control the special inter-server resynchronization features. This package must be deployed and configured on both Primary and Secondary UCA servers.
During resynchronization involving 'compressed time'event replay, a UCA Primary or Standalone server will adopt a policy of actively preventing certain Remote Handler outputs (e.g. alarm raise requests, script executions) being generated by correlations triggered by the replay of historical events which already existed in the UCA server Events database. In contrast, previously unseen events delivered during resynchronization that trigger new correlations will be allowed to generate such Remote Handler outputs. This policy has been implemented in an attempt to prevent unwanted or 'duplicate' outputs being generated during the process. In contrast, resynchronization of a Secondary UCA server under any circumstances will not generate any outputs because they are globally disabled at the Remote Handler level (provided the integrator has implemented the output enable/disable call-outs)..

The following sections describe the resynchronization process for both Primary/Standalone and Secondary servers.

13.2 Primary/Standalone Server Initial Resynchronization

A UCA Primary (for Primary read Standalone if only one system is used) system initial resynchronization involving 'compressed time' event replay is summarized in the following sequence diagram:



The following sequence of tasks is carried out on initial resynchronization of a Primary system with one or more event sources:

- By default (configurable in uca.properties), the Event and Notification databases are preserved on a system restart.
- A request is issued by the Primary's Server via its Remote Handler and Generic Collector specialization (i.e. Specific Collector in the above diagram) to begin resynchronization with all available event sources (provided that a UCA Primary-Secondary inter-server resynchronization is not already underway). This request takes the form of a Java RMI function call [requestResync()] issued from its Remote Handler REQUEST_RESYNC callout to its Generic Collector ManagementIF. In the default Generic Collector implementation provided with UCA, this call simply prints the request on the system console. It is the responsibility of the integrator to provide a specific implementation (e.g. a Specific Collector) that interfaces with the event source(s) and responds to this call as required.
- For a Primary system (not Standalone), its Specific Collector must also execute a Secondary Resynchronization delay on receipt of requestResync(), before attempting to proceed with the source resynchronization process. Its purpose is to provide a window in which the Primary system waits to determine if a Secondary system has concurrently issued a higher priority inter-system resynchronization request. This request (in the form of a Java RMI function call [secondaryResyncStarted()] is sent from the Secondary's Remote Handler SECONDARY_RESYNC_STARTED callout to the Primary's Generic Collector

ManagementIF). If such a request is received, it must be processed ahead of the outstanding source resynchronization request as described in the following section. Assuming that such a request has not been received during the delay period, the Primary's Specific Collector is free to proceed with a source resynchronization (the Secondary system is then actively prevented from issuing an inter-system resynchronization request until the complete source resynchronization cycle is completed).

- The Primary's Specific Collector sends a CYCLE_START event to the Primary's Server with the following attributes:
    - systemClass = "GenericCollector"
    - systemInstance = "V1.0"
    - eventRank = "resync"
    - moClass = "System"
    - moInstance = "CYCLE_START"
- The CYCLE_START event is automatically consumed by the Primary's Server (no filters or maps are required) and causes it to begin a source resynchronization cycle from one or more individual sources.
- The P rimary's Specific Collector will carry out in turn the following resynchronization sequence involving one or more event sources:
    - The Primary's Specific Collector requests a pre-defined event source to begin delivering a resynchronization stream of events.
    - When the event stream is ready for delivery, the Primary's Specific Collector must send a START event to the Primary's Server with the following attributes:
        - systemClass = event source type name e.g. "NMS"
        - systemInstance = event source instance name e.g. "Source_1"
        - eventRank = "resync"
        - moClass = "System"
        - moInstance = "START"
    - The START event is automatically consumed by the Primary's Server (no filters or maps are required) and causes it to begin buffering any subsequent resynchronization events received from the defined event source in a special area of the Events database. 'Live' events received from all other event sources will be buffered in a memory-resident events buffer until the complete resynchronization operation is completed, whereupon normal processing is resumed. For this reason, the memory configuration of the Primary's Server TomCat JVM (set in the CATALINA_OPTS environment variable) must have been previously set to allow sufficient heap memory resources to accommodate the largest anticipated set of buffered 'live' events from all sources. Memory usage during resynchronization testing may be monitored by examining the Primary's System object from the Working Memory Viewer and adjusted as required.
    - The Primary's Specific Collector will then deliver the set of outstanding (resynchronization) events from the defined event source to the server, which in turn stores them in the Events database. Note that it is no longer necessary for the Specific Collector to know which server to send the events to; this is automatically handled by the underlying Generic Collector implementation using its knowledge of the currently attached server(s). As described above, 'live' events from other event sources will be buffered in memory.
    - When the outstanding (resynchronization) event stream from the defined event source is exhausted, the Primary's Specific Collector must send a FINISH event to the Primary's Server with the following attributes:
        - systemClass = event source type name

        - systemInstance = event source instance name
        - eventRank = “resync”
        - moClass = “System”
        - moInstance = “FINISH”
    - The FINISH event is automatically consumed by the Primary’s Server (no filters or maps are required) and causes it to construct a time ordered ‘replay’ list of events for the defined event source, including time advance events. The ‘replay’ events list contains the following types of events:
        - Time advance events
        - Alarm raise & clear events, corresponding to historical raise and clear events (from the defined event source) that existed in the Events database prior to the resynchronization process
        - Alarm raise events, corresponding to new raise events received in the resynchronization stream (from the defined event source).
        - Alarm update events, derived from differences between previously active historical events and matching but updated raise events received in the resynchronization stream (both types from the defined event source)
        - Alarm clearance events, derived from the necessity to automatically close previously active historical raise events that were not present in the resynchronization stream (both types from the defined event source)
    - The Primary’s Server locks the Timers associated with all existing correlations (to prevent the ‘compressed time’ replay process from inadvertently triggering temporal correlations associated with other event sources). It then sets the UCA ‘clock’ to the second boundary before the first replay event and initiates the ‘compressed time’ event replay process, during which the contents of the ‘replay’ events list are delivered as fast as possible for processing.
    - Each time a time advance event is encountered, the UCA ‘apparent time’ is advanced by the specified number of 1 second steps and the fireAllRules() on the Rules Engine method is called after each 1 second step. In this way, time dependent correlations for the event source only are correctly handled during the accelerated replay.
    - When the ‘replay’ events list is exhausted (and ‘apparent time’ has advanced to the time at which the ‘compressed time’ event replay process began), the Primary’s Server unlocks all previously locked Timers, ceases to buffer live events and begins to process the contents of the live events buffer. As this buffer itself includes time advance events, the ‘apparent time’ at the end of the outstanding event replay process is gradually advanced to match the ‘actual time’ until the system catches up with reality!
    - Finally, the Primary’s Server reports defined source resynchronization completion via its Remote Handler to its Specific Collector. This report takes the form of a Java RMI function call [completedResync()] issued from its Remote Handler callout to its Generic Collector ManagementIF. In the default Generic Collector implementation provided with UCA, this call simply prints the request on the system console. It is the responsibility of the integrator to provide a specific implementation e.g. in a Specific Collector, that recognizes that resynchronization with the defined source is complete and allows it to continue with the next available source.
    - The Primary’s Specific Collector repeats the above sequence for the remaining event sources.
- The Primary’s Specific Collector sends a CYCLE_FINISH event to the Primary’s Server with the following attributes:
    - systemClass = “GenericCollector”

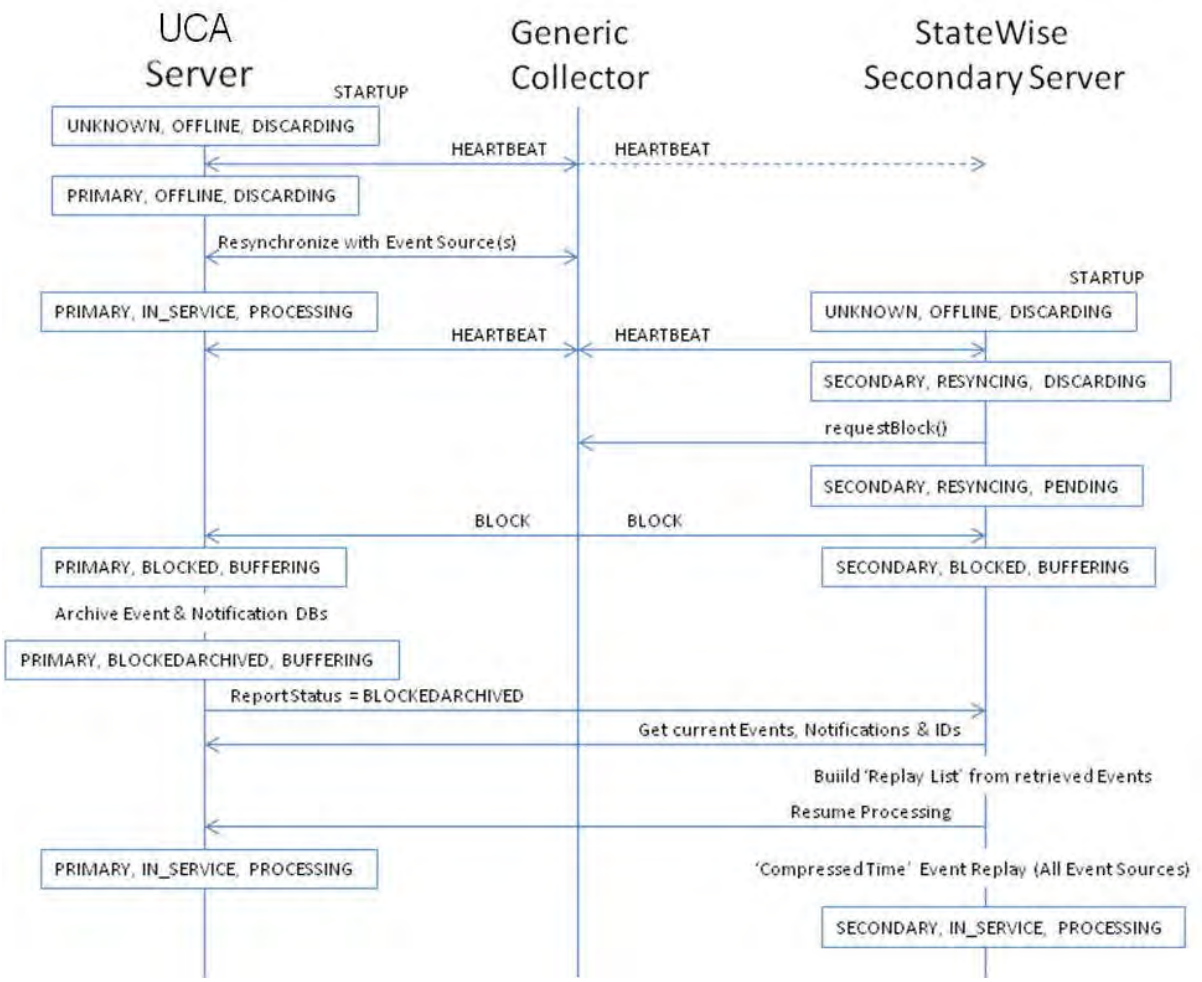
- systemInstance = “V1.0”
- eventRank = “resync”
- moClass = “System”
- moInstance = “CYCLE_FINISH”

- The CYCLE_FINISH event is automatically consumed by the Primary’s Server (no filters or maps are required) and causes it to complete a source resynchronization cycle from one or more individual sources. From this point on, a Secondary system may request an inter-system resynchronization.
- Finally, the UCA Primary system is now resynchronized with its event source(s) and is processing events received in real-time. This is the normal steady state.

As stated previously, depending on the correlation requirements, the ‘time compressed’ event replay process may be ignored. This is simply achieved by not sending the START and FINISH events described in the sequence above and is the responsibility of the integrator to configure when building the Specific Collector. It is also then the responsibility of the integrator to ensure that events are gathered from one or more sources, time ordered and replayed as a composite sequence if required.

13.3 Primary/Secondary Inter-System Resynchronization

Primary/Secondary inter-system resynchronization involving ‘compressed time’ event replay and ‘ID matching’ is summarized in the following sequence diagram:



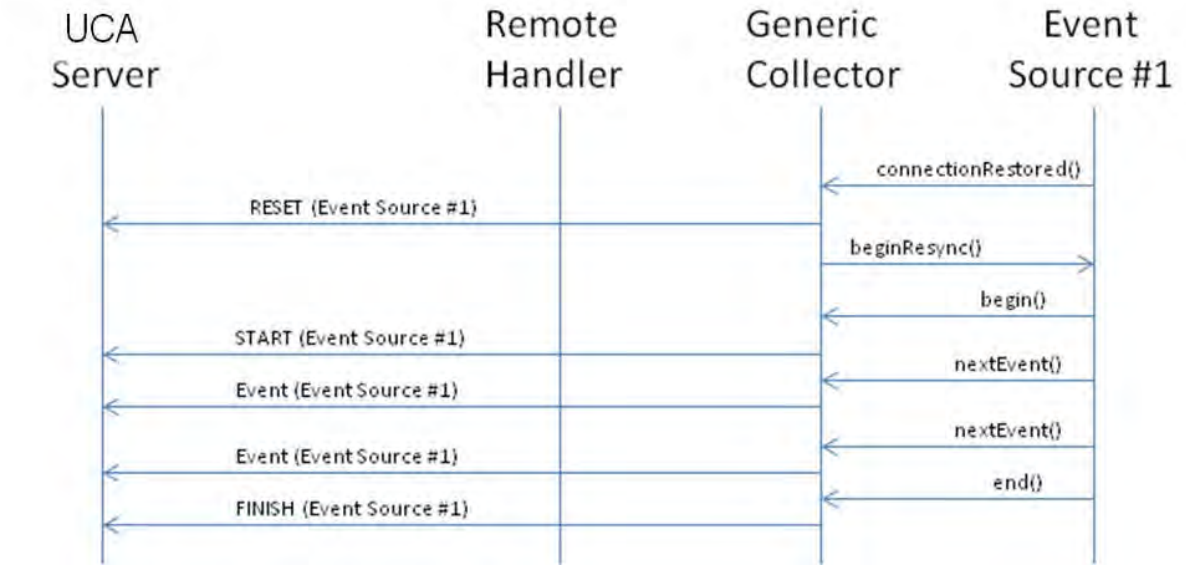
The following sequence of tasks (controlled by the Resilience Package of rules) is carried out when a Secondary server attempts to resynchronize with a Primary server:

- Assuming that the Generic Collector has been configured to deliver HEARTBEAT messages to both Primary and Secondary servers and that the Primary server has already restarted and resynchronized with its event source(s), then the Primary server will be in the IN_SERVICE:PROCESSING state, processing live events.
- On startup of the Secondary's Server (in the UNKNOWN role), it waits in the OFFLINE:DISCARDING state to receive a HEARTBEAT message from the Generic Collector. The HEARTBEAT message informs the second server that the other server is in the Primary role. The second server then adopts the Secondary role and and enters the RESYNCING:DISCARDING state, discarding any live events sent to it by the Generic Collector.
- The Secondary's Server issues a request to the Primary's Generic Collector to issue a BLOCK message. This request takes the form of a Java RMI function call [requestBlock()] issued from the Secondary's Remote Handler REQUEST_BLOCK callout to the Primary's Generic Collector ManagementIF.The purpose of the BLOCK message is to halt live event processing in both the Primary and Secondary servers at exactly the same point in their respective event streams. The Primary's Generic Collector can guarantee to issue the BLOCK message to both servers at this point because it is responsible for duplication and delivery of each event. The BLOCK message has the following attributes:
    - systemClass = "GenericCollector"
    - systemInstance = "V1.0"
    - eventRank = "resync"
    - moClass = "System"
    - moInstance = "BLOCK"
- As a result of receiving the BLOCK message, the Primary's Server will:
    - Enter the BLOCKED:BUFFERING state and begin buffering live events from all event sources in memory.
    - Archive the Event & Notification databases to remove any information that is no longer needed by active events or correlations.
    - On completion of the archive process, enter the BLOCKEDARCHIVED:BUFFERING state and report its new state to the Secondary's Server.
    - Wait until informed by the Secondary's Server that it can resume processing of live events. While waiting, live events are buffered in memory and for this reason, the memory configuration of the Primary's Server TomCat JVM (set in the CATALINA_OPTS environment variable) must have been previously set to allow sufficient heap memory resources to accommodate the largest anticipated set of buffered events. Memory usage during inter-system resynchronization testing may be monitored by examining the Primary's System object from the Working Memory Viewer and adjusted as required.
    - Resume processing of buffered 'live' events (starting with those buffered in memory) when instructed by the Secondary's Server.
- As a result of receiving the BLOCK message, the Secondary's Server will:
    - Enter the BLOCKED:BUFFERING state and wait for the Primary's Server to inform it that it has completed the archive process. Any 'live' events will be buffered in memory and for this reason, the memory configuration of the Secondary's Server TomCat JVM (set in the CATALINA_OPTS environment variable) must have been previously set to allow sufficient heap memory resources to accommodate the largest anticipated set of buffered events. Memory usage during inter-system resynchronization testing may be monitored by examining the Secondary's System object from the Working Memory Viewer and adjusted as required.

- When instructed by the Primary's Server that archiving is complete, it will retrieve details of all current Events, Notifications and the current values of all ID counters used on the Primary's Server. The latter are used to re-initialize the ID counters in the Secondary's Server. It will also retrieve the Primary Server's 'clock' time and set the Secondary Server's 'clock' time to the same value. In order to prevent subsequent drift between the Primary's and Secondary's Servers, it is essential to configure a time synchronization protocol between them e.g. NTP.
- Build the 'replay' events list for all event sources and on completion, instruct the Primary's Server to re-commence live event processing.
- The Secondary's Server will then commence 'compressed time' event replay processing using the 'replay' events list created above. Note that wherever possible, details of the equivalent existing Notifications retrieved from the Primary's Server will be used to re-construct the equivalent Notifications on the Secondary's Server, thus preserving the correspondence of Notification IDs between the Servers.
- On completion of the 'compressed time' event replay processing, the Secondary's Server will adopt the IN_SERVICE:PROCESSING state and begin processing 'live' events (starting with those buffered in memory).

13.4 Server Resynchronization Following Connection Re-establishment

Server resynchronization following connection re-establishment and involving 'compressed time' event replay is summarized in the following sequence diagram:



The following sequence of tasks is carried out when a server attempts to resynchronize with an event source following connection loss and re-establishment:

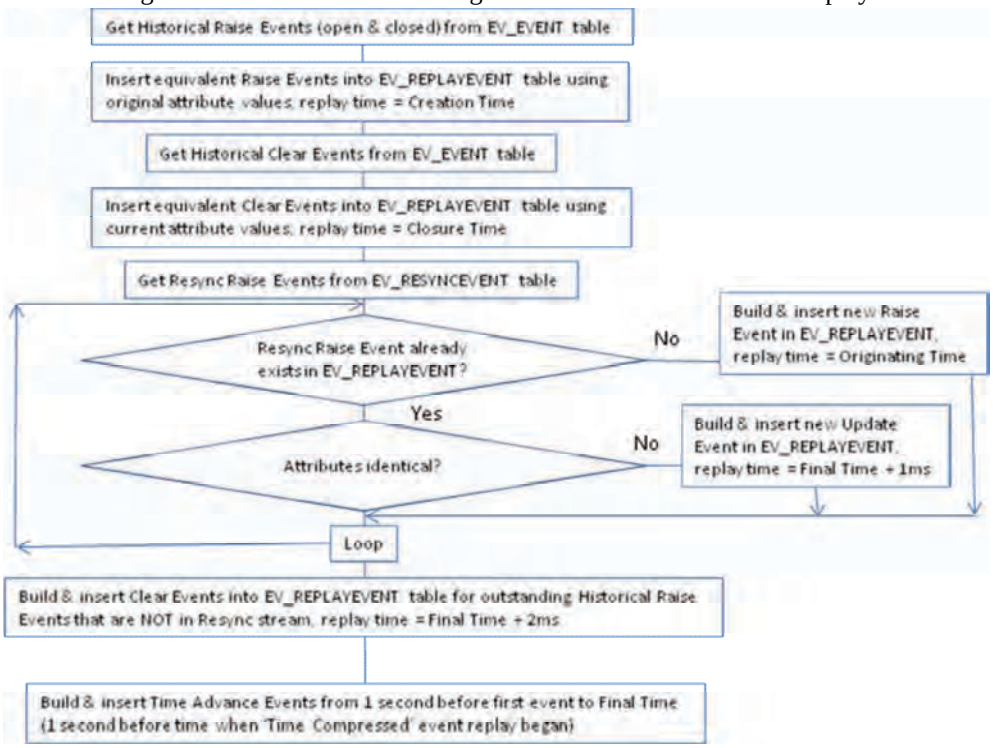
- Either the event source notifies the Generic Collector specialization that connectivity to the event source has been re-established or the Generic Collector itself re-establishes connectivity to the event source.
- The Generic Collector specialization sends a RESET message to the server to automatically generate clear events for all outstanding raise events in the Event database previously received from the defined event source. The RESET message has the following attributes:
  - systemClass = event source type name
  - systemInstance = event source instance name
  - eventRank = "resync"
  - moClass = "System"
  - moInstance = "RESET"

- When the event stream is ready for delivery, processing proceeds as described for the Primary/Standalone initial resynchronization scenario with the delivery of a START message.
- Again and depending on the correlation requirements, the ‘time compressed’ event replay process may be ignored. This is simply achieved by not sending the START and FINISH events described in the sequence above and is the responsibility of the integrator to configure when building the Generic Collector specialization.

In a UCA resilient configuration utilizing two servers operating in hot-standby, it will be necessary for each server to undergo the resynchronization process described above following connection re-establishment. This implies that the Generic Collector spec ialization is responsible for instructing both servers to undergo resynchronization and for delivering the START/Events/FINISH messages simultaneously to each server.

13.5 Replay Event List Construction

The following flow-chart summarises the algorithm used to construct the ‘Replay’ event list.



Chapter 14 Value Packs

14.1 Introduction

A value pack is a collection of information, such as rules, actions etc. that can be packaged up to usefully support a generic capability. For example a value pack might generically address problem identification and impact analysis for a telecoms SDH network, or a general purpose power failure scenario within a digital TV broadcast network.
To be more specific, a value pack bundles the following information:

- **Actions** User defined actions can be included in a value pack. Once a value pack is loaded, user actions will be available to all running rules.
- **Meta-model** Each value pack can have its own meta-model. A value pack meta-model is merged into any currently deployed meta-models and can have classes with an ‘External’ stereotype to link with other value packs or deployed models.
- **Filters and Rules** Each value pack can have its own ‘scenarios’ XML files that will get merged and deployed into the system.
- **Scripts** A value pack must supply any scripts that it runs locally.
- **Configuration** A value pack can supply its own system properties that will be available to all rules.

When UCA is started all previously activated value packs will be initialised in memory.
All system functions are in a single system value pack.

14.2 Description

14.2.1 Internal structure

A value pack is a directory with a known structure that has been put into the ‘valuepacks’ directory of the deployed UCA application.
The top level structure for a value pack is:

- **actions** a directory that contains the action classes
- **configuration** a directory that contains the value pack properties and any other developer properties files
- **models** contains the meta model files.
- **rules** the scenarios XML files.
- **scripts** scripts that are run by the value pack rules
- **vp-manifest.xml** contains the value pack group, name, version and description

14.2.2 Actions

The actions directory can contain:

- The action code as one or more jar files **[name].jar**.
- The **[name]_declarations.properties** properties file.
- The **[name]_classloader.properties** properties file.

14.2.3 Configuration

The configuration directory will contain the system.properties file.

14.2.4 Models

The models directory will contain the meta-model files, these can be UCA XML or ‘XMI’ files. All models in this directory will be loaded.

- Multiple files in the ‘valuepacks/VPName/models/’ directory will be loaded
- Meta-model files can be in either Argo XMI or UCA XML format (the former will be converted to the latter)
- An external node in the meta-model must be prefixed by a namespace e.g. com.name.product.vp.IPLink
- All top level nodes must have 'Model' as the parent

14.2.5 Rules

The rules directory will contain the scenarios XML files as exported by the Scenario Manager. All XML files in the directory will be loaded.
VP rules can consist of new scenarios (which will be deployed as such) and also individual rules (which will be added to the list).

14.2.6 Scripts

This contains scripts used in the ‘runScripts’ action.
Note that the 'valuepack' directory is used as the scripts base directory.
It is usual to include the value pack path and scripts directory for use in VP rules as a system property. e.g example-1.0/scripts/ascript. This prevents any hard-coding of script paths in rules.

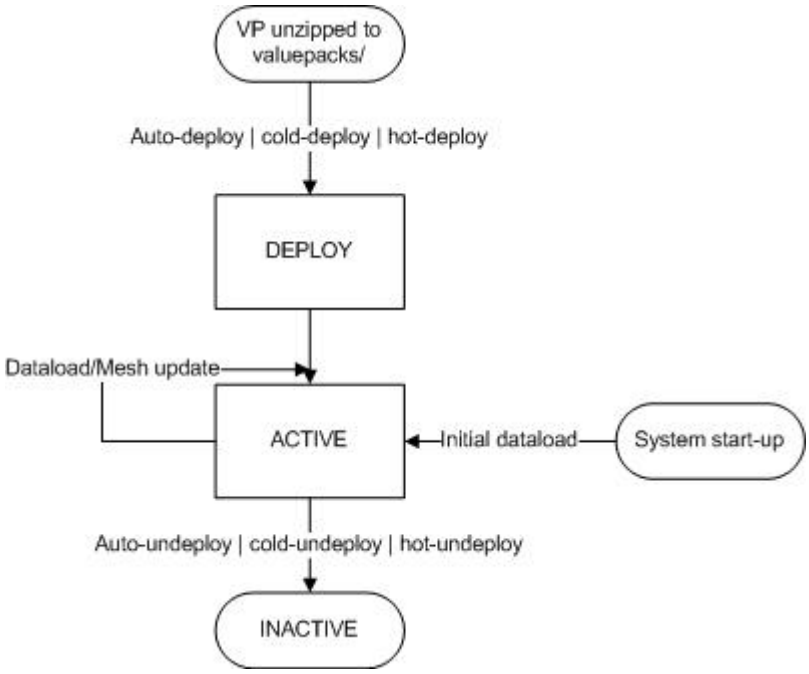
14.2.7 VP Manifest

The vp-manifest.xml file contains information about the value pack such as name, group and version. A manifest file must be included into the value pack directory structure. An example is given below:

```
<?xml version="1.0" encoding="UTF-8"?>
<valuepack vp-format-version="1.0">
  <group>com.HP</group>
  <name>test</name>
  <version>1.0</version>
  <description>A test demonstration value pack.</description>
</valuepack>
```

14.3 Value pack Lifecycle

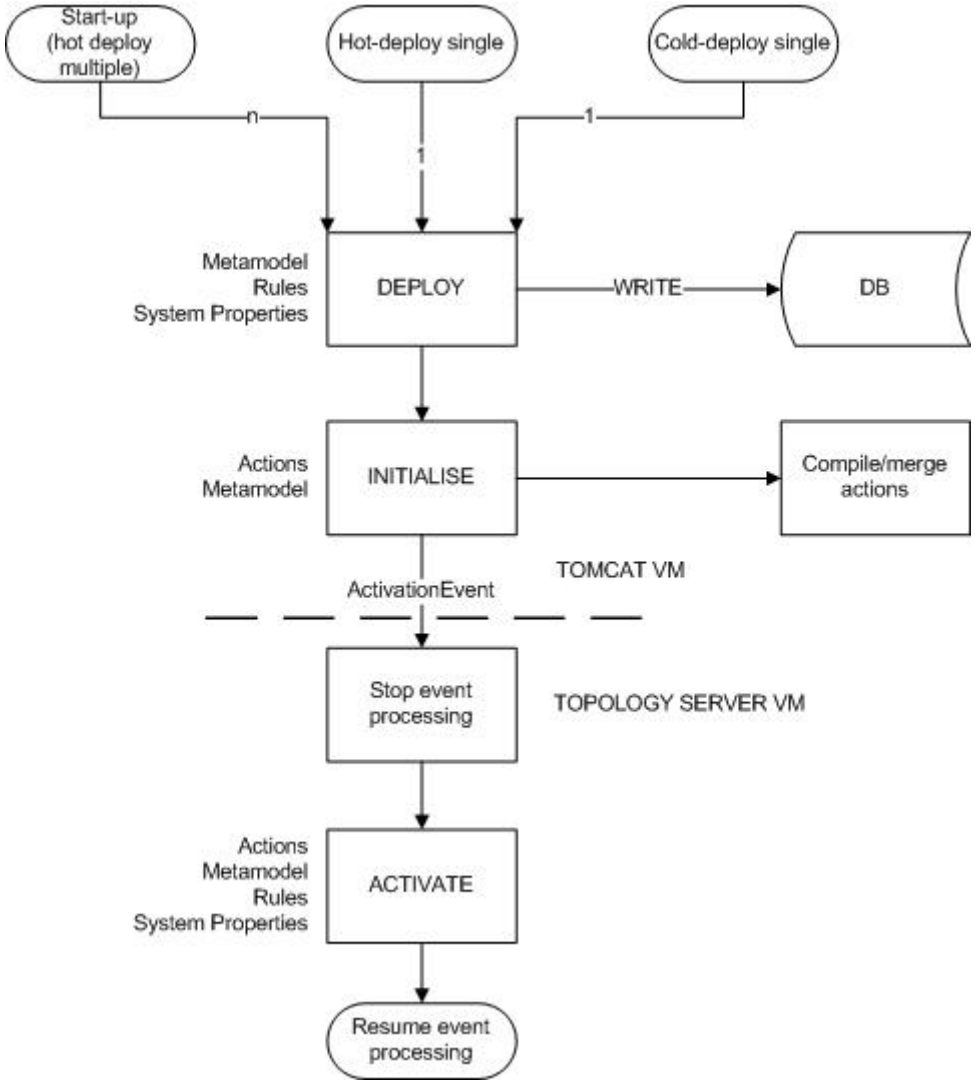
The VP lifecycle is shown below:



The VP moves from a 'deploy' state to an 'active' state through the process of auto-deploy, cold-deploy or hot-deploy. Once the database tables have been updated, the active VP will always be activated by the system on start-up. In the case of hot-deployment, the VP will be automatically activated dynamically.
An active VP may then be deactivated through a process of auto-deploy, cold-deploy or hot-undeploy. The database tables will be removed but the VP files remain on the file system.

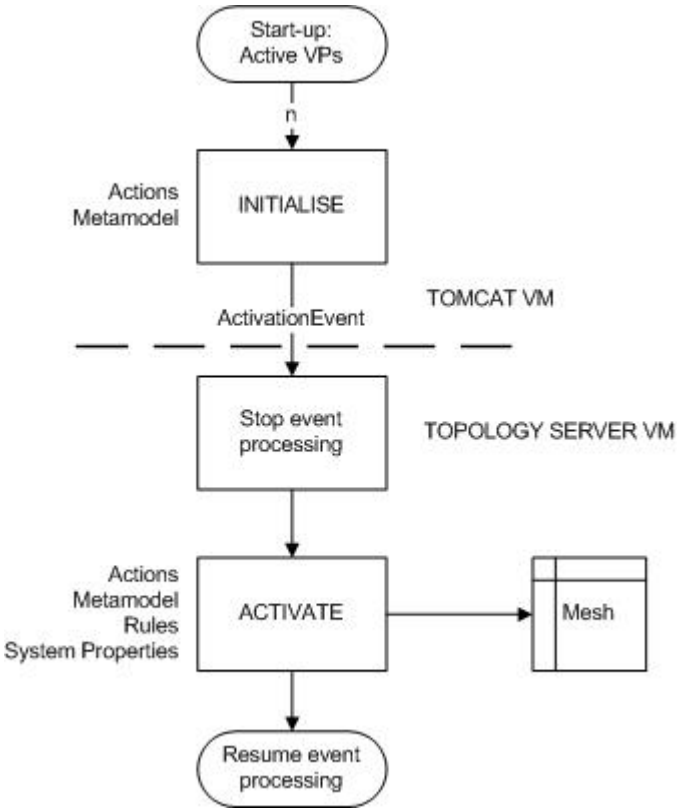
14.3.1 Value Pack Deployment process

Value pack 'deployment' can be divided into three distinct phases: deployment, initialisation and activation.
In the deployment phase the database entries are written. For multiple deployments, the VPs are deployed in priority order (0=highest priority, 20=lowest priority)
In the initialisation phase the rules are compiled and merged with the current rulebase.
In the activation phase all value pack components are loaded into memory. The mesh will not be updated with the inventory for a VP until the inventory is loaded and a mesh update event is fired.



14.3.2 Start up procedure

When the system starts-up, the currently active VPs are loaded before the deployment of any new VPs. The diagram below outlines the start-up process:



The mesh is updated with the inventory currently in the database for each VP. Please note that only the currently activated VPs will be data loaded.
VPs that are cold-deployed can have inventory added before a system start-up to allow them to be data-loaded in this manner.
VPs that are hot-deployed will not be data-loaded and will require an inventory load and mesh update event.

14.3.3 Inventory and Mesh Update Events

The inventory manager can be used to add inventory for the classes contained in the VP. This is useful for newly hot-deployed VPs which will not have been data loaded.
See the section on data-loading via the System Manager (and / or the inventory_manager Python script documentation for further information on how to data-load inventory.
After the database has been populated, a mesh update (scheduled or otherwise) will load the newly loaded inventory in the mesh.

14.4 Deploying a value pack

Value packs are not deployed by default and must be deployed and un-deployed with the scripts provided on a running instance of the application.
Deploying and un-deploying a value pack must be done on each machine separately within a resilient pair. For both deploy and undeploy the UCA instance must be running and started.

14.4.1 How to Deploy

Deploying a value pack involves two steps:

a. Copying the value pack zip file to the server and unzipping it into the **vp/** directory.

b. Using the **bin/vp-deployer.sh** script.

```
Usage:     vp-deploy.sh command [path] user password
```

```
command     list | hot-deploy | cold-deploy | hot-undeploy | cold-
            undeploy (note - only use hot-* when the system is
            running)

path        relative path of the value pack in the 'valuepacks'
            subdirectory [only for deploy/undeploy]

username    the UCA username e.g. system

password    the UCA password e.g. system

options     preserve-inventory (use on hot-deploy/hot-undeploy only)
            force (use on hot- or cold-deploy/undeploy)
            no-resync (use on hot-deploy/undeploy)
```

Note – the path will usually be the name and version of the valuepack i.e. example-1.0

Hot deploy
The hot-deploy command will deploy a value pack into a running system. Any deployment errors will be output to the console. If the value pack is already installed, the user will be informed.

- The only VP that should be at priority 0 is the System value pack
- If the manifest is incorrect for any of the VPs to be deployed, the entire process will be aborted
- The VP deployment script will only work on 'localhost' i.e. you must use it on the UCA server only

Cold deploy
The cold-deploy command will deploy a value pack on a system on which only the manager server is running.

14.4.2 How to Un-deploy

To un-deploy a value pack again use the **bin/vp-deployer.sh** script. This will remove all the components of the value pack from the instance.

hot-undeploy
The hot-deploy command will un-deploy a value pack from a running system. Any un-deployment errors will be output to the console. If the value pack is not installed, the user will be informed.

cold-undeploy
The cold-undeploy command will un-deploy a value pack from a system on which only the manager server is running.

Note – I the case you undeploy and then re-deploy the same valuepack and you want to preserve the instances corresponding to the valuepack model; you have to use the ‘preserve-inventory’ option. By using this option the instance inventory will be kept unchanged.

14.4.3 Listing all active value packs

The ‘**list**’ command (on both a running and non-running system) will output a list of all active value packs.

14.4.4 Deploying a value pack on start up

A value pack can be ‘auto’ deployed when UCA is started up by including the empty file ‘DEPLOY’ in the value pack directory. A value pack will only be deployed the first time this file is detected since the file will be renamed to avoid repeated auto-deployment.
If the ‘DEPLOY’ file is detected for a for a value pack that is already deployed then the value pack will be deactivated and then reactivated.

14.5 Supplied value packs

14.5.1 System actions

The system actions are deployed as a VP with the highest priority. This consists of a single jar file containing the system actions and all configuration files.

14.5.2 Resilience

For resilient configurations licensed to use the 'Resilience VP', the Resilience VP will load all the rules, actions, properties and scripts.
However, the following manual configuration changes will still be necessary:

- Set the correct values in configuration/system.properties for the host and peer before loading the VP
- Edit the uca.properties and set the 'system.mode' property before restarting
- Edit the remotehandler.properties
- Edit the genericcollector.properties

14.6 Assumptions

14.6.1 Namespace

- The namespace is defined as the concatenation of both the group and name information held in the VP manifest file
- The namespace is not case-sensitive (i.e. it will always be converted to lower case only) therefore com.name.vp.example and com.name.vp.EXAMPLE refer to the same namespace
- Individual class names with a VP are case sensitive with respect to data-loading, so if a class is delcared as 'IPLink' in the namespace 'com.company.product.vp', then the fully qualified name in the inventory would be com.company.product.vp.IPLink (i.e. not the lower case variant)
- The namespace information is used when generating the inventory tables in the database. For example com.name.vp.IPLink will create the database table md_com_name_vp_iplink

Class Names

- Class names must not contain the underscore character since this is the escaped class name for VPs

Metamodel

- It is possible to start a system with no metamodels deployed since VPs can be hot deployed into an 'empty' system. Therefore, if no metamodel has been loaded, a default Model-only metamodel will be used by the system

14.7 Current Limitations

- There is currently no support for VP updates
- Oracle tables names longer than 30 chars are currently not supported
- Actions can be hot deployed but NOT hot undeployed or hot updated; UCA will need to be restarted to pick-up the new changes. Currently, this leads to two issues:
  - Undeploying and re-deploying a VP with actions will not pick-up the changes to the actions until a restart
  - Using an action in a non-VP rule and undeploying that VP will have the effect that the action will continue to work until the system is restarted, at which point it will fail to work.
- Rules will require re-compilation – you must change the import and re-compile against the latest codebase

Chapter 15 Reference Information

15.1 Object Type Attributes

15.1.1 Object

Attribute Name	Type	Purpose
Base Class	String	Base class name selected from list of classes defined in metamodel
Sub Class	String	Sub (derived) class name
Instance	String	Friendly name or alias
Unique Reference	String	Unique identifier
State	Enumeration	Selected from list of possible states (normal, degraded, failed)
Service State	Enumeration	Selected from list of possible service states (in service, commissioning, out of service, in maintenance)
Current Problem List Entry Count (Current Total Event Count)	Integer	Number of synthetic and external alarm reports currently attached to this mesh object
Current Problem List Entry Count Changed (Current Total Event Count Trend)	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Total Synthetic (Degraded + Failed) Event Count	Integer	Number of synthetic alarm reports currently attached to this mesh object
Total Synthetic (Degraded + Failed) Event Count Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
External Event Count	Integer	Number of external alarm reports currently attached to this mesh object
External Event Count Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Degraded Synthetic Event Count	Integer	Number of synthetic alarm reports with degraded target state currently attached to this mesh object
Degraded Synthetic Event Count Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Failed Synthetic Event Count	Integer	Number of synthetic alarm reports with failed target state currently attached to this mesh object
Failed Synthetic Event Count Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Parent Base Class	String	Parent mesh object base class name, as for Base Class
Parent Sub Class	String	Parent mesh object sub (derived) class name
Parent Instance	String	Parent mesh object friendly name or alias
Parent Unique Reference	String	Parent mesh object unique identifier
Parent Mesh Object	Mesh Object	Parent mesh object reference e.g. obj0
Grandparent Base Class	String	Grandparent mesh object base class name, as for Base Class

Attribute Name	Type	Purpose
Grandparent Sub Class	String	Grandparent mesh object sub (derived) class name
Grandparent Instance	String	Grandparent mesh object friendly name or alias
Grandparent Unique Reference	String	Grandparent mesh object unique identifier
Grandparent Mesh Object	Mesh Object	Grandparent mesh object reference e.g. obj0
Importance	Enumeration	Chosen from a list of possible values (unknown, gold, silver, bronze)
Parent State	Enumeration	Selected from list of possible states (normal, degraded, failed)
Grandparent State	Enumeration	Selected from list of possible states (normal, degraded, failed)
Timer State	Enumeration	Selected from a list of possible values (undefined, initialised, running, suspended, expired, completed)
Timer State Changed	Boolean	Selected from true or false
Last Event Creation Time	Date	Time at which the latest event mapped to this object was raised in UCA
Last Event Originating Time	Date	Time at which the latest event mapped to this object was raised in the originating system
Last Event MO Instance	String	The name of the Managed Object in the originating system on which the latest event mapped to this object was raised
Last Event MO External Event ID	String	The unique identifier assigned by the originating system to the latest event mapped to this object
Last Event Additional Text (Last Event Additional Data)	String	Contents of the Additional Text field of the latest alarm report
Last Event Probable Cause	String	Contents of the Probable Cause field of the latest alarm report
Last Event Severity	Enumeration	Contents of the Severity field of the latest alarm report
Last Event Previous Severity	Enumeration	Contents of the Severity field of the previous alarm report
Update pending count	Integer	The number of outstanding alarm update events

15.1.2 Child Group

Attribute Name	Type	Purpose
Base Class	String	Base class name of the mesh objects held in this group, selected from list of classes defined in metamodel
Parent Base Class	String	Parent mesh object base class name selected from list of classes defined in metamodel
Parent Sub Class	String	Parent mesh object sub (derived) class name
Parent Instance	String	Parent mesh object friendly name or alias
Parent Unique Reference	String	Parent mesh object unique identifier
Parent Mesh Object	Mesh Object	Parent mesh object reference
Grandparent Base Class	String	Grandparent mesh object base class name selected from list of classes defined in metamodel
Grandparent Sub Class	String	Grandparent mesh object sub (derived) class name
Grandparent Instance	String	Grandparent mesh object friendly name or alias
Grandparent Unique Reference	String	Grandparent mesh object unique identifier
Grandparent Mesh Object	Mesh Object	Grandparent mesh object reference
Member Count	Integer	Number of member mesh objects in group
Normal Count	Integer	Number of normal member mesh objects in group
Normal Count Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Normal Percentage	Integer in range 0 –100%	Percentage of member mesh objects in group that are normal
Normal Percentage Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Degraded Count	Integer	Number of degraded member mesh objects in group
Degraded Count Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Degraded Percentage	Integer in range 0 –100%	Percentage of member mesh objects in group that are degraded
Degraded Percentage Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Failed Count	Integer	Number of failed member mesh objects in group
Failed Count Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Failed Percentage	Integer in range 0 –100%	Percentage of member mesh objects in group that are failed
Failed Percentage Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
List Of Children	Child Group	Reference to Child Group
External Event Count	Integer	Number of external (non-synthetic) events on members of this group
Synthetic Event Count	Integer	Number of synthetic (non-external) events on members of this group
Total (External & Synthetic) Event Count	Integer	Number of synthetic & external events on members of this group

Attribute Name	Type	Purpose
Timer State	Enumeration	Selected from a list of possible values (undefined, initialised, running, suspended, expired, completed)
Timer State Changed	Boolean	Selected from true or false

15.1.3 Associate Group

Attribute Name	Type	Purpose
Base Class	String	Base class name of the mesh objects held in this group, selected from list of classes defined in metamodel
Parent Base Class	String	Parent mesh object base class name selected from list of classes defined in metamodel
Parent Sub Class	String	Parent mesh object sub (derived) class name
Parent Instance	String	Parent mesh object friendly name or alias
Parent Unique Reference	String	Parent mesh object unique identifier
Parent Mesh Object	Mesh Object	Parent mesh object reference
Grandparent Base Class	String	Grandparent mesh object base class name selected from list of classes defined in metamodel
Grandparent Sub Class	String	Grandparent mesh object sub (derived) class name
Grandparent Instance	String	Grandparent mesh object friendly name or alias
Grandparent Unique Reference	String	Grandparent mesh object unique identifier
Grandparent Mesh Object	Mesh Object	Grandparent mesh object reference
Member Count	Integer	Number of member mesh objects in group
Normal Count	Integer	Number of normal member mesh objects in group
Normal Count Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Normal Percentage	Integer in range 0 –100%	Percentage of member mesh objects in group that are normal
Normal Percentage Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Degraded Count	Integer	Number of degraded member mesh objects in group
Degraded Count Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Degraded Percentage	Integer in range 0 –100%	Percentage of member mesh objects in group that are degraded
Degraded Percentage Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Failed Count	Integer	Number of failed member mesh objects in group
Failed Count Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Failed Percentage	Integer in range 0 –100%	Percentage of member mesh objects in group that are failed
Failed Percentage Changed	Enumeration	Selected from a list of possible values (increased, unchanged, decreased)
Hops	Integer	Number of ‘hops’ to propagate state changes to peers
List Of Associates	Associate Group	Reference to Associate Group
Timer State	Enumeration	Selected from a list of possible values (undefined, initialised, running, suspended, expired, completed)
Timer State Changed	Boolean	Selected from true or false

15.1.4 Notification

Attribute Name	Type	Purpose
Notification Type	Enumeration	Selected from list of possible types(primary, marker, problem report, service impact, root cause)
Notification Rank	Integer	Severity or Importance of the notification, in range 1 to 10 (1 = highest)
Base Class	String	Base class name of mesh object that Notification is owned by, selected from list of classes defined in metamodel
Unique Reference	String	Unique identifier of mesh object that Notification is owned by
Context Name	String	Name of the 'target' context in which this Notification may also be inserted
Originating Base Class	String	Base class name of mesh object that Notification originates from (same as Base Class if this is a primary Notification), selected from list of classes defined in metamodel
Originating Unique Reference	String	Unique identifier of originating mesh object
Originating Context Name	String	Name of the context in which this Notification is inserted
Notification ID	Integer	Unique numerical identifier (-1 if marker Notification)
Notification Master Alarm Status	Enumeration	Selected from list of possible states (not created, pending, present, terminated) – reports existence or otherwise of master alarm report from NMS
Associated Trouble Ticket ID	String	Unique identifier of an associated Trouble Ticket (empty if none present)
Associated Trouble Ticket Status	Enumeration	Selected from list of possible states (not created, pending, present, closed)
Associated Trouble Visibility	Enumeration	Selected from list of possible visibilities (unknown, visible, hidden)
Associated Trouble Ticket State Changed	Boolean	Selected from true or false
Notification Alarms Demoted	Boolean	Indicates whether attached alarm reports have been demoted under master alarm report in NMS
Administrative State	Integer	Selected from list of possible states (active, locked, no action)
Event List Size	Integer	Current alarm report list size of mesh object that Notification is attached to
Original Problem List Size	Integer	Previous alarm report list size of mesh object that Notification is attached to
Build Time	Date	Time that Notification object was created
Current Time	Date	The current UCA system time
Notification Owner ID	Integer	Unique numerical identifier (Notification ID of primary Notification if this is a marker Notification, otherwise identical to Notification ID)
Notification Common Base Classes	Boolean	Indicates if Base Class and Originating Base Class fields are identical (do not rely on this as a test for a primary Notification)
Notification Common Unique References	Boolean	Indicates if Unique Reference and Originating Unique Reference fields are identical (if true, then this is a primary Notification)

Attribute Name	Type	Purpose
Notification Common Context Names	Boolean	Indicates if Context Name and Originating Context Name are identical
Timer State	Enumeration	Selected from a list of possible values (undefined, initialised, running, suspended, expired, completed)
Timer State Changed	Boolean	Selected from true or false
Notification Creation Time	Date	Time that the Notification object was created
Notification Locked Time	Date	Time that the Notification object was administratively locked
Notification Message	String	Message associated with Notification

15.1.5 Script

Attribute Name	Type	Purpose
Script Name	String	Name of script file to execute
Script Owner Base Class	String	Base class name of mesh object that originated this Script selected from list of classes defined in metamodel
Script Owner Unique Reference	String	Unique identifier of mesh object that originated this Script
Script State	Enumeration	Selected from list of possible states (initialising, running, finished)
Script Status	Enumeration	Selected from list of possible status (normal, error)
Script Exit Code	Integer	Script return code
Script Output	String	Latest Script stdout text
Script Error	String	Latest Script stderr text
Timer State	Enumeration	Selected from a list of possible values (undefined, initialised, running, suspended, expired, completed)
Timer State Changed	Boolean	Selected from true or false

15.1.6 System

Attribute Name	Type	Purpose
Platform Average CPU	Integer in range 0 –100%	UCA server platform average CPU load
Platform Disk #1 Free Space	Integer in range 0 –100%	UCA server platform disk #1 free space
Platform Disk #2 Free Space	Integer in range 0 –100%	UCA server platform disk #2 free space
Platform Database Percentage Tablespace Used	Integer in range 0 –100%	UCA server platform database tablespace used
Platform OS Physical Memory Used	Integer in range 0 –100%	UCA server platform physical memory used
Platform OS Swap Memory Used	Integer in range 0 –100%	UCA server platform swap memory used
System JVM Heap Memory Used	Integer in range 0 –100%	UCA system JVM heap memory used
System JVM Non-Heap Memory Used	Integer in range 0 –100%	UCA system JVM non-heap memory used
TomCat JVM Heap Memory Used	Integer in range 0 –100%	UCA TomCat JVM heap memory used
TomCat JVM Non-Heap Memory Used	Integer in range 0 –100%	UCA TomCat JVM non-heap memory used
Latest Information Exception Text	String	UCA system latest Information exception text
Latest Warning Exception Text	String	UCA system latest Warning exception text
Latest Non-Recoverable Exception Text	String	UCA system latest Non-Recoverable exception text
Latest Fatal Exception Text	String	UCA system latest Fatal exception text
Server Identfier	String	Either “A” or “B”
Server Operating Mode	Enumeration	Selected from a list of possible values (standalone, resilient)
Server Resync Cycle Running	Boolean	Selected from true or false
Server Operating Role	Enumeration	Selected from a list of possible values (singleton, primary, secondary, unknown)
Server Operating State	Enumeration	Selected from a list of possible values (offline, in service, archiving, updating, resyncing, blocked, blocked and archived, closed down, unknown)
Server Event Processing Mode	Enumeration	Selected from a list of possible values (discarding, pending, buffering, gathering, processing)
Peer Server Resync Cycle Running	Boolean	Selected from true or false
Server Event Activity	Enumeration	Selected from a list of possible values (normal, missing, unknown)

Attribute Name	Type	Purpose
Local (this) Server to Peer Server Link State	Enumeration	Selected from a list of possible values (normal, timeout, failed, bad arguments, unknown)
Peer (Server) Operating Role	Enumeration	Selected from a list of possible values (singleton, primary, secondary, unknown)
Peer Server Operating State	Enumeration	Selected from a list of possible values (offline, in service, archiving, updating, resyncing, blocked, blocked and archived, closed down, unknown)
Current NMS Heartbeat Source	String	DNS Name or IP Address of current NMS Heartbeat Source (Platform on which Generic Collector is running)
Previous NMS Heartbeat Source	String	DNS Name or IP Address of previous NMS Heartbeat Source (Platform on which Generic Collector is running)
Current & Previous Heartbeat Sources Are Same	Boolean	Selected from true or false
Heartbeat From Generic Collector (on NMS) Late	Boolean	Selected from true or false
State of Link between Generic Collector (on NMS) and Local (this) Server	Enumeration	Selected from a list of possible values (normal, failed, unknown)
State of Link between Generic Collector (on NMS) and Peer Server	Enumeration	Selected from a list of possible values (normal, failed, unknown)
Local (this) Server Role reported by Generic Collector (on NMS)	Enumeration	Selected from a list of possible values (singleton, primary, secondary, unknown)
Peer Server Role reported by Generic Collector (on NMS)	Enumeration	Selected from a list of possible values (singleton, primary, secondary, unknown)
Older Than Peer	Boolean	Selected from true or false if System Time of Local (this) Server is older than System Time of Peer Server
Timer State	Enumeration	Selected from a list of possible values (undefined, initialised, running, suspended, expired, completed)
Last Update Type	Enumeration	Selected from a list of possible values (unknown, system status, peer status, timer status, event activity status, heartbeat status, platform attributes, information exception, warning exception, non-recoverable exception, fatal exception)..

The ’Last Update Type’ attribute is an indicator that allows the user to identify which sub-group of attributes in the System object were last updated. The following table lists the possible values of the ‘Last Update Type’ indicator and the associated attributes that may have been updated:

Last Update Type Indicator	Attributes Updated
system status	Server Operating Role Server Operating State Server Event Processing Mode Server Resync Cycle Running
peer status	Local (this) Server to Peer Server Link State Peer Server Operating Role Peer Server Operating State Older Than Peer Peer Server Resync Cycle Running
Timer status	Timer State
event activity status	Server Event Activity
heartbeat status	Current NMS Heartbeat Source Previous NMS Heartbeat Source Current & Previous Heartbeat Sources Are Same Heartbeat From Generic Collector (on NMS) Late State of Link between Generic Collector (on NMS) and Local (this) Server State of Link between Generic Collector (on NMS) and Peer Server Local (this) Server Role reported by Generic Collector (on NMS) Peer Server Role reported by Generic Collector (on NMS)
platform attributes	Platform Average CPU Platform Disk #1 Free Space Platform Disk #2 Free Space Platform Database Tablespace Used Platform Physical Memory Used Platform Swap Memory Used System JVM Heap Memory Used System JVM Non-Heap Memory Used TomCat JVM Heap Memory Used TomCat JVM Non-Heap Memory Used
information exception	Latest Information Exception Text
warning exception	Latest Warning Exception Text
non-recoverable exception	Latest Non-Recoverable Exception Text
Fatal exception	Latest Fatal Exception Text

15.2 Actions

15.2.1 External and Synthetic Alarm Reports

UCA processes alarm reports from two distinct sources:

- External alarm reports are those that originate from an external NMS and as a result of the filtering and mapping process are attached to target mesh objects in the state mesh.
- Synthetic alarm reports originate from actions carried out by UCA in response to Rules firing. They are the mechanism by which UCA artificially modifies the state of mesh objects in the state mesh.

Each mesh object maintains a current problem list and this may simultaneously contain both external and synthetic alarm reports. The overall state of a mesh object is determined by the highest state of any alarm reports attached to it (external and synthetic).
External alarm reports are uniquely identifiable and UCA is able to identify the full details of the original alarm report received from the external NMS using the event database. In contrast, synthetic alarm reports do not have a unique identifier and simply serve to modify the state of an object.
A mesh object may, as a result of 'overlapping' or simultaneous correlations contain any number of synthetic alarm reports of the same or different severity.
To aid with processing simultaneous correlations, each mesh object maintains a number of alarm report counts and trend indicators. These include:

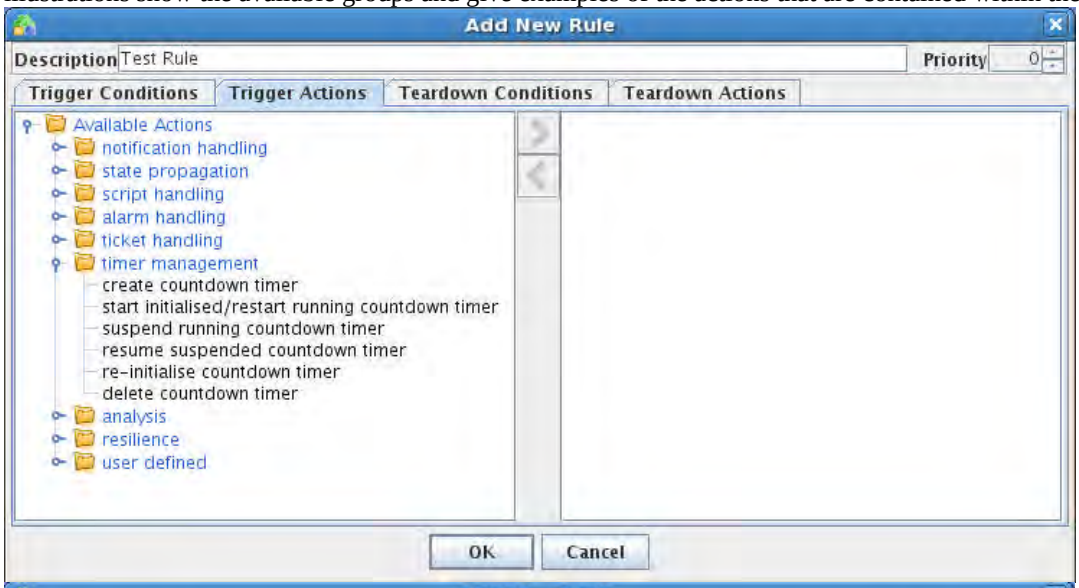
- Current problem list count – the sum of all external and synthetic alarm reports in the current problem list.
- Current problem list count changed – the trend in the current problem list count (increased, unchanged, decreased).
- External event count - the sum of all external alarm reports in the current problem list.
- External event count changed - the trend in the external event list count (increased, unchanged, decreased).
- Synthetic degraded event count – the sum of all synthetic degraded alarm reports in the current problem list.
- Synthetic degraded event count changed – the trend in the synthetic degraded event count (increased, unchanged, decreased).
- Synthetic failed event count – the sum of all synthetic failed alarm reports in the current problem list.
- Synthetic failed event count changed – the trend in the synthetic failed event count (increased, unchanged, decreased).

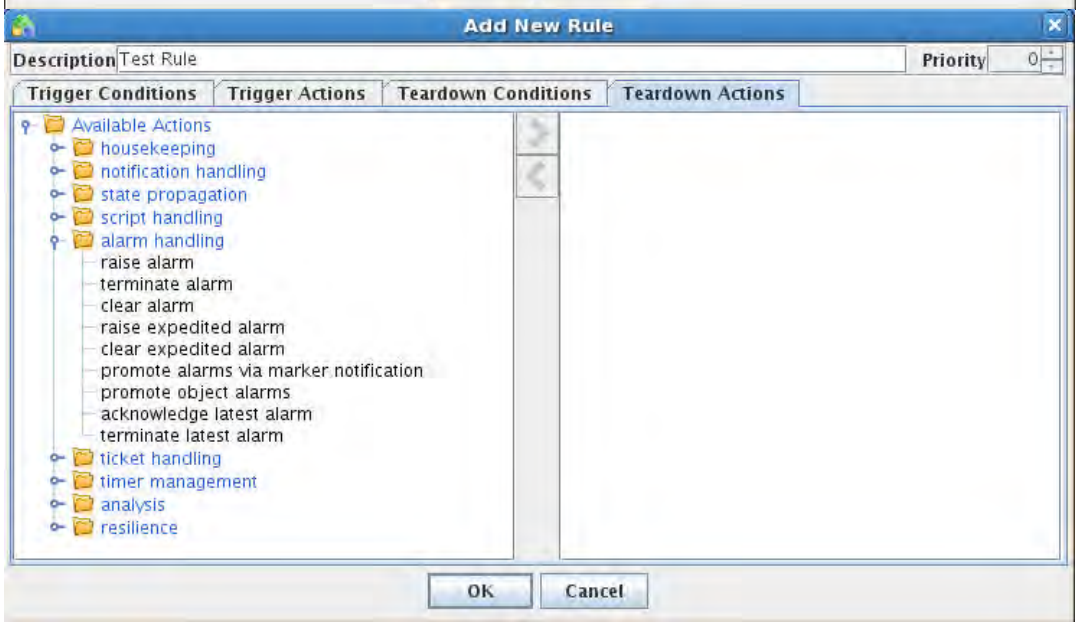
The following table summarises the values of these attributes under varying conditions:

Event	Current Problem List Count	Current Problem List Count Changed	External Event Count	External Event Count Changed	Synthetic Degraded Event Count	Synthetic Degraded Event Count Changed	Synthetic Failed Event Count	Synthetic Failed Event Count Changed
External Alarm Raise	+1	increased	+1	increased	(as before)	unchanged	(as before)	unchanged
External Alarm Clear	-1	decreased	-1	decreased	(as before)	unchanged	(as before)	unchanged
Synthetic Degraded Raise	+1	increased	(as before)	unchanged	+1	increased	(as before)	unchanged
Synthetic Degraded Clear	-1	decreased	(as before)	unchanged	-1	decreased	(as before)	unchanged
Synthetic Failed Raise	+1	increased	(as before)	unchanged	(as before)	unchanged	+1	increased
Synthetic Failed Clear	-1	decreased	(as before)	unchanged	(as before)	unchanged	-1	decreased

15.2.2 Action Groups

The Trigger and Teardown Action tabs in the UCA Scenario Manager contain a number of action groups. Each such group gathers together those actions that are logically related e.g. timer management. The following illustrations show the available groups and give examples of the actions that are contained within them:





Each group may simultaneously contain symmetric actions (where the same action is available from both Trigger and Teardown rules) e.g. Run Script, and asymmetric actions (where complementary or opposite actions only made available in Trigger or Teardown rules) e.g. Lock Notification. In addition, the Housekeeping group is only available from Teardown Rules.

Depending on system configuration, the Resilience group may not be available in a standalone system and the User-defined actions group may be extended with user-supplied actions.

The following sections describe the currently available set of system actions.

15.2.2.1 Housekeeping

Remove Object In Normal State from WM

State Mesh Model

Fired Rule Viewer Mnemonic

tearRemoveMONormStateWM

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action removes the supplied (mesh) object from the current context (working memory). If the object was dynamically created it is also destroyed, otherwise it continues to exist in the state mesh.
This action is normally called from a low-priority housekeeping rule in the current context after all other processing has been completed and the supplied object has returned to the normal state.

Scenario Manager Configuration Dialogue

The context (working memory) in which the triggering rule is deployed and where the mesh object is inserted.

The mesh object to remove from the current context.

Option to record action execution details in the database.

Remove Associate Group In Normal State From WM

State Mesh Model

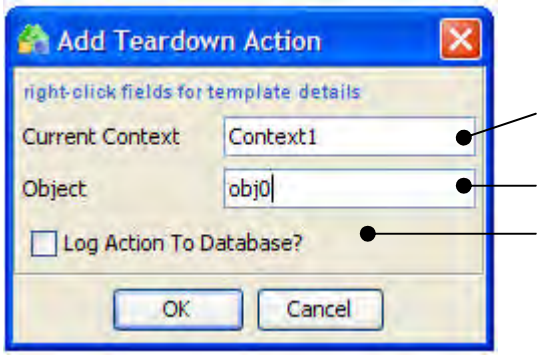
Fired Rules Viewer Mnemonic

tearRemoveAssocGrpNormState

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action removes the supplied associate group from the current context (working memory) although it continues to exist in the state mesh.
This action is normally called from a low-priority housekeeping rule in the current context after all other processing has been completed and there are no longer any degraded or failed associate group member objects.

Scenario Manager Configuration Dialogue

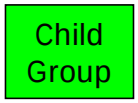


The context (working memory) in which the triggering rule is deployed and where the associate group is inserted.

The associate group to remove from the current context.
Option to record action execution details in the database.

Remove Child Group In Normal State from WM

State Mesh Model



Fired Rules Viewer Mnemonic

tearRemoveChildGrpNormStateWM

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action removes the supplied child group from the current context (working memory) although it continues to exist in the state mesh.

This action is normally called from a low-priority housekeeping rule in the current context after all other processing has been completed and there are no longer any degraded or failed child group member objects.

Scenario Manager Configuration Dialogue

The context (working memory) in which the triggering rule is deployed and where the child group is inserted.

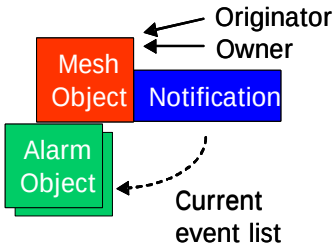
The child group to remove from the current context.

Option to record action execution details in the database.

15.2.2.2 Notification Handling

Create Notification Against Object

State Mesh Model



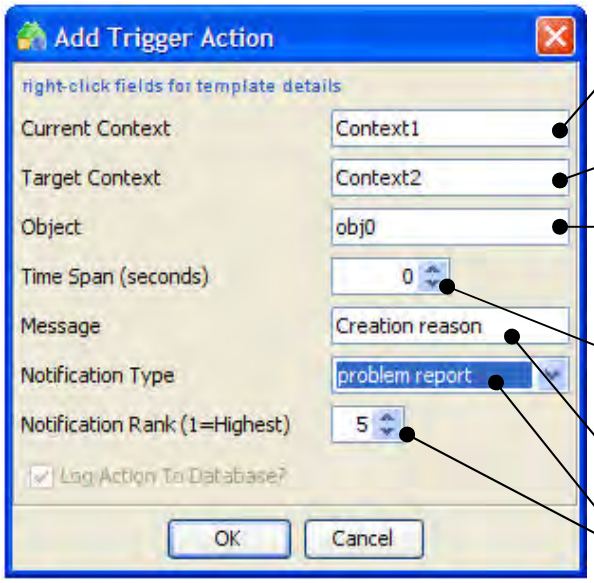
Fired Rules Viewer Mnemonics

trigCreateNotMO
tearCreateNotMO

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action builds a contributory events list in the database from the active alarm reports attached to the supplied (mesh) object, creates a notification record in the database and attaches the contributory events list to it.
An 'active' notification report with a list of contributory events (alarm reports) is automatically displayed on the Notification Viewer GUI.
A new notification object (of the requested type and rank) is created and is inserted into the current context (working memory) and an optional target context. Note that both the originating and owning object references in the notification object are set to the supplied object (it is a primary notification object). The current event list is also initialised with the contents of the contributory events list.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed, where the supplied mesh object is inserted and where the new notification object will be inserted.

An alternative context in which the new notification object may also inserted (if un-used, set as Current Context).

The (mesh) object providing zero or more active alarm reports that both originates and owns the new notification object.

The maximum age of active alarm reports in the object that will be added to the contributory events list (0 = use all active alarm reports).

(Optional) message to be displayed in the notification report on the Notification Viewer GUI.

Type and rank of notification object to create.

Create Notification Against Object Using Latest Event

State Mesh Model

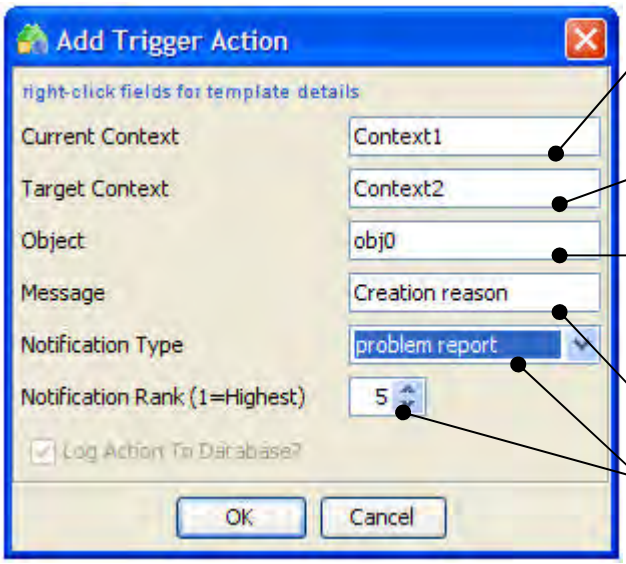
Fired Rules Viewer Mnemonic

trigCreateNotMOLatestEvent
tearCreateNotMOLatestEvent

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action builds a contributory events list in the database from the latest active alarm report attached to the supplied (mesh) object, creates a notification record in the database and attaches the contributory events list to it. An 'active' notification report with a single contributory event (alarm report) is automatically displayed on the Notification Viewer GUI.
A new notification object (of the requested type and rank) is created and is inserted into the current context (working memory) and an optional target context. Note that both the originating and owning object references in the notification object are set to the supplied object (it is a primary notification object). The current event list is also initialised with the contents of the contributory events list.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed, where the supplied mesh object is inserted and where the new notification object will be inserted.

An alternative context in which the new notification object may also inserted (if un-used, set as Current Context).

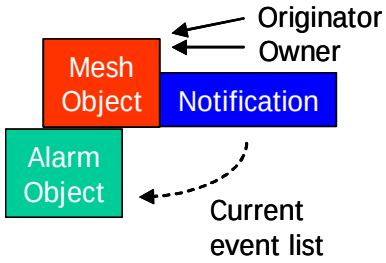
The (mesh) object providing the latest active alarm report, that both originates and owns the new notification object.

(Optional) message to be displayed in the notification report on the Notification Viewer GUI.

Type and rank of notification object to create.

Update Notification Against Object

State Mesh Model



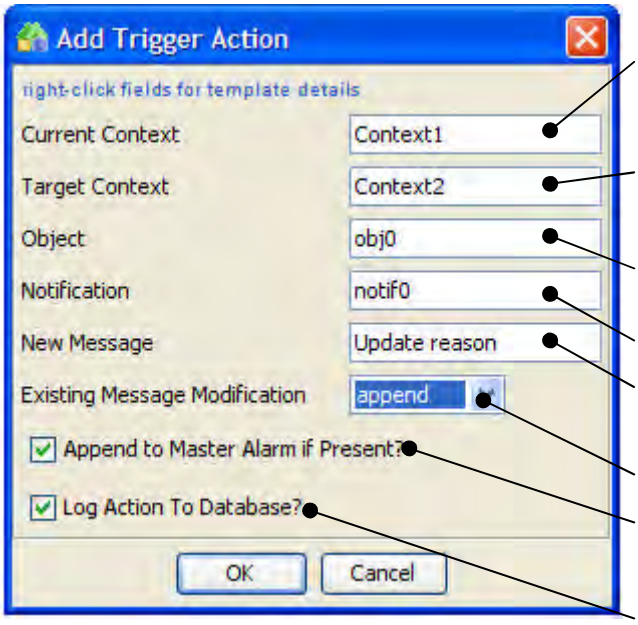
Fired Rules Viewer Mnemonics

trigUpdateNotMO
tearUpdateNotMO

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action updates the contributory event list attached to the notification record in the database for the supplied notification object, using the latest active alarm report attached to the supplied (mesh) object
The contributory events list of the notification report associated with the supplied notification object is automatically updated with the new alarm report on the Notification Viewer GUI.
The event list count and trend attributes of the supplied notification object are updated in the current context (working memory) & (if used) optional target context. The current event list is also updated in line with the contents of the contributory events list.
Optionally, the message to be displayed in the notification report on the Notification Viewer GUI may be replaced or additional information may be appended.
Optionally (and if it is present), the Master Alarm associated with the supplied notification object may be updated with the details of the latest active alarm report attached to the supplied (mesh) object.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied mesh & notification objects are inserted.

An alternative context in which the supplied notification object may also inserted (if un-used, set as Current Context).

The object providing an additional active alarm report that both originates and owns the supplied notification object.

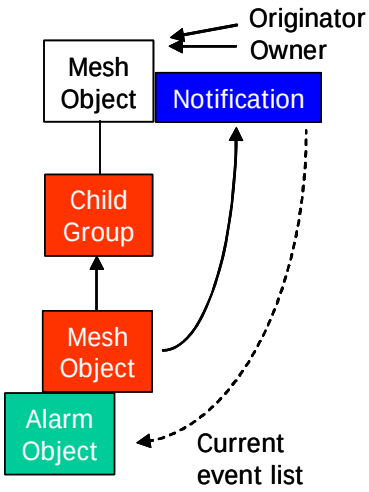
The notification object to be updated.
(Optional) updated/replacement message to be displayed in the notification report on the Notification Viewer GUI.

Message modification options {unchanged|append|replace}
Option to append the latest active alarm report to the Master Alarm associated with the notification (if present)

Option to record action execution details in the database.

Update Notification Against Object Parent

State Mesh Model



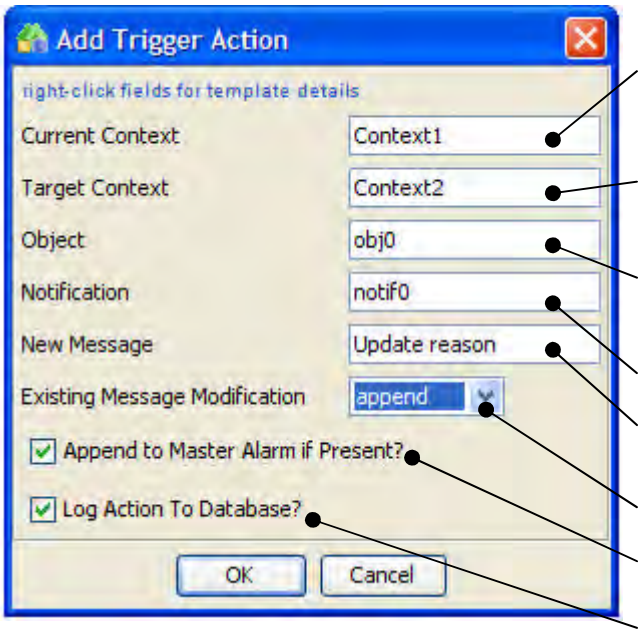
Fired Rules Viewer Mnemonic

trigUpdateNotMOParent
tearUpdateNotMOParent

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action updates the contributory event list attached to the notification record in the database for the supplied notification object, using the latest active alarm report attached to the supplied (mesh) object
The contributory events list of the notification report associated with the supplied notification object is automatically updated with the new alarm report on the Notification Viewer GUI.
The event list count and trend attributes of the supplied notification object are updated in the current context (working memory) & (if used) optional target context. The current event list is also updated in line with the contents of the contributory events list.
Optionally, the message to be displayed in the notification report on the Notification Viewer GUI may be replaced or additional information may be appended.
Optionally (and if it is present), the Master Alarm associated with the supplied notification object may be updated with the details of the latest active alarm report attached to the supplied (mesh) object.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied mesh & notification objects are inserted.

An alternative context in which the supplied notification object may also inserted (if un-used, set as Current Context).

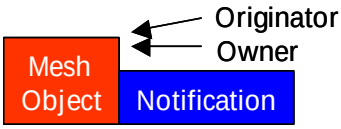
The object providing an additional active alarm report whose parent mesh object both originates and owns the supplied notification object.

The notification object to be updated.
(Optional) updated/replacement message to be displayed in the notification report on the Notification Viewer GUI.

Message modification options {unchanged|append|replace}
Option to append the latest active alarm report to the Master Alarm associated with the notification (if present)
Option to record action execution details in the database.

Remove Notification Against Object

State Mesh Model



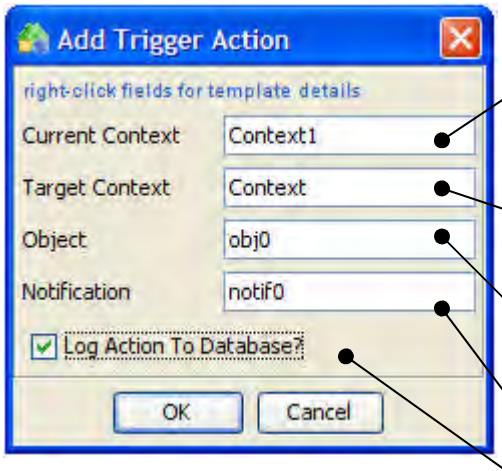
Fired Rules Viewer Mnemonics

trigRemoveNotMO
tearRemoveNotMO

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action closes the notification record in the database associated with the supplied primary notification object. The status of the notification report associated with the supplied notification object is automatically set to 'closed' on the Notification Viewer GUI.
The supplied notification object is detached from the supplied (mesh) object and removed from the current context (working memory) & (if used) optional target context. The notification object is then destroyed.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied mesh & notification objects are inserted.

An alternative context in which the supplied notification object may also inserted (if un-used, set as Current Context).

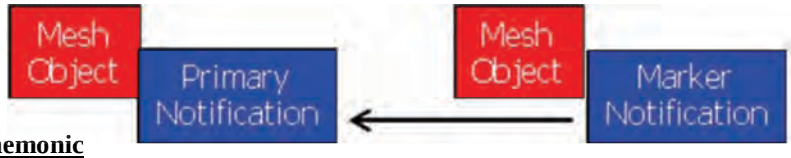
The (mesh) object that owns the supplied notification object.

The notification object to be removed and destroyed.

Option to record action execution details in the database.

Create Marker Notification Against Object

State Mesh Model



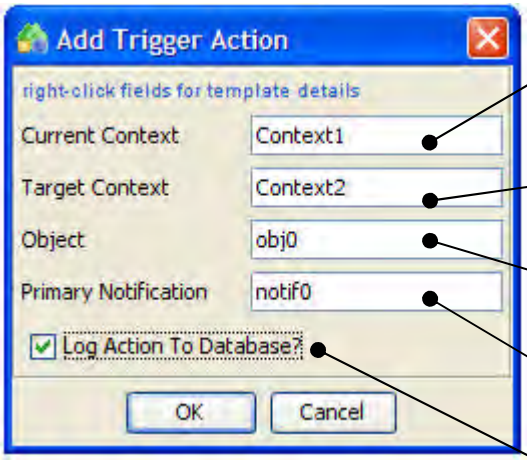
Fired Rules Viewer Mnemonic

trigCreateMarkerNotMO

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to create a new marker notification against the supplied (mesh) object, associated to the supplied primary notification. The new marker notification is inserted into the current context (working memory) & (if used) optional target context.
The supplied (mesh) object is added to the affected objects list maintained for the primary notification in the Notification database.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied marker notification object is inserted.

An alternative context in which the supplied marker notification object may also inserted (if un-used, set as Current Context).

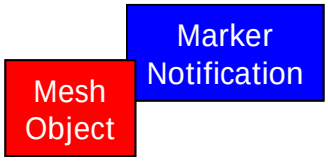
The (mesh) object to which the new marker notification is attached

The primary notification object to which the new marker notification is associated

Option to record action execution details in the database.

Remove Marker Notification Against Object

State Mesh Model



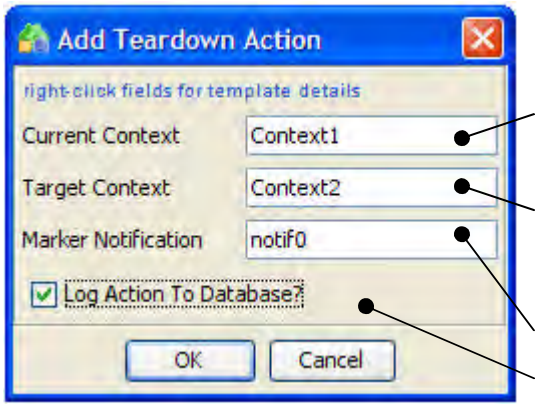
Fired Rules Viewer Mnemonic

tearRemoveMarkerNot

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action removes the supplied marker notification from the current context (working memory) & (if used) optional target context and it is then destroyed.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied marker notification object is inserted.

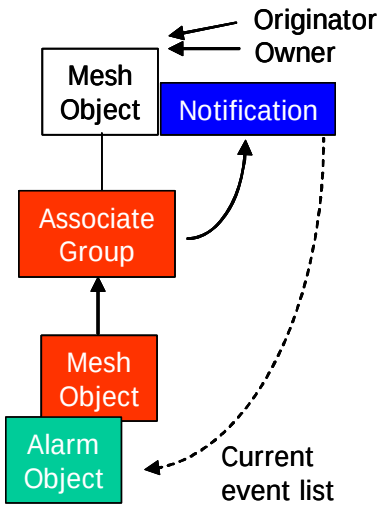
An alternative context in which the supplied marker notification object may also inserted (if un-used, set as Current Context).

The marker notification object to be removed.

Option to record action execution details in the database.

Create Notification Against Associate Group Parent

State Mesh Model



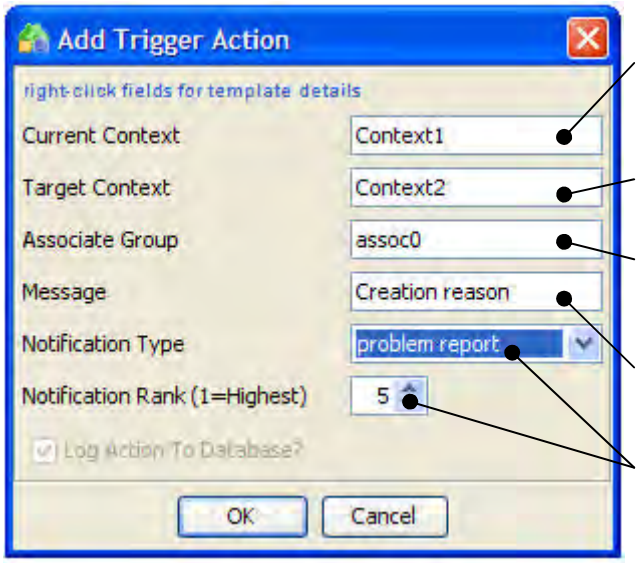
Fired Rules Viewer Mnemonic

trigCreateNotAssocGrpParent

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action builds a contributory event list in the database from the active alarm reports attached to the (mesh) objects contained in the supplied associate group, creates a notification record in the database and attaches the contributory event list to it.
An active notification report with a list of contributory events (alarm reports) is automatically displayed on the Notification Viewer GUI.
A new notification object (of the requested type and rank) is created and is inserted into the current context (working memory) and an optional target context. Note that both the originating and owning object references in the notification object are set to the associate group's parent object (it is a primary notification object). The current event list is also initialised with the contents of the contributory events list.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed, where the supplied associate group is inserted and where the new notification object will be inserted.

An alternative context in which the new notification object may also inserted (if un-used, set as Current Context).

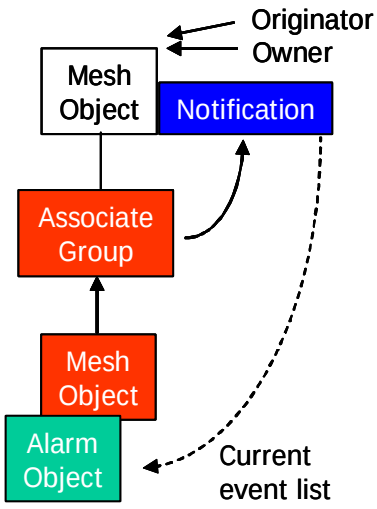
The associate group whose member objects will provide zero or more active alarm reports and whose parent object both originates and owns the new notification object.

(Optional) message to be displayed in the notification report on the Notification Viewer GUI.

Type and rank of notification object to create

Remove Notification Against Associate Group Parent

State Mesh Model



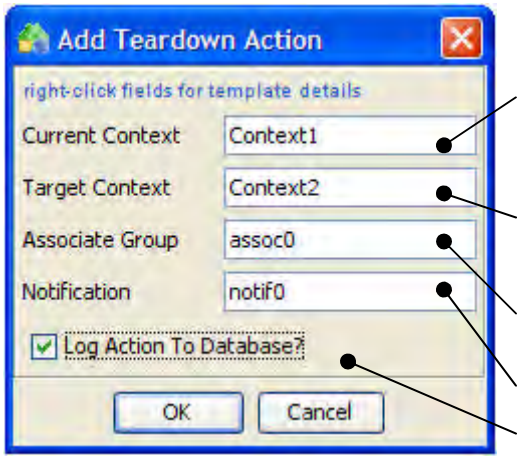
Fired Rules Viewer Mnemonic

tearRemoveNotAssocGrpParent

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action closes the notification record in the database associated with the supplied primary notification object. The status of the notification report associated with the supplied notification object is automatically set to 'closed' on the Notification Viewer GUI.
The supplied notification object is detached from the supplied associate group's parent (mesh) object and removed from the current context (working memory) & (if used) optional target context. The notification object is then destroyed.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied associate group & notification objects are inserted.

An alternative context in which the supplied notification object may also inserted (if un-used, set as Current Context).

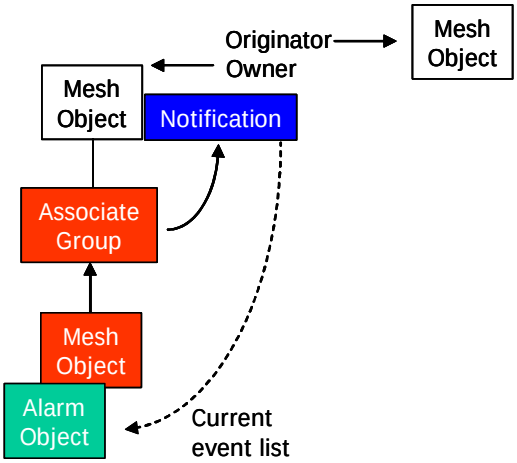
The associate group whose parent (mesh) object owns the supplied notification object.

The notification object to be removed and destroyed.

Option to record action execution details in the database.

Create Notification Against Referenced Associate Group Parent

State Mesh Model



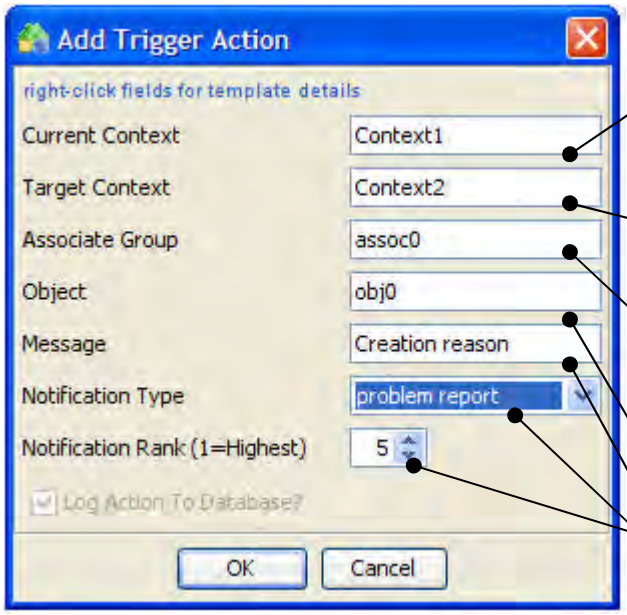
Fired Rules Viewer Mnemonic

trigCreateNotRefAssocGrpParent

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action builds a contributory event list in the database from the active alarm reports attached to the (mesh) objects contained in the supplied associate group, creates a notification record in the database and attaches the contributory event list to it.
An active notification report with a list of contributory events (alarm reports) is automatically displayed on the Notification Viewer GUI.
A new notification object (of the requested type and rank) is created and is inserted into the current context (working memory) and an optional target context. Note that the originating object reference in the notification object is set to the supplied object while the owning object reference is set to the associate group's parent object.
The current event list is also initialised with the contents of the contributory events list.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed, where the supplied mesh object and associate group are inserted and where the new notification object will be inserted.

An alternative context in which the new notification object may also inserted (if un-used, set as Current Context).

The associate group whose member objects will provide zero or more active alarm reports and whose parent object owns the new notification object.

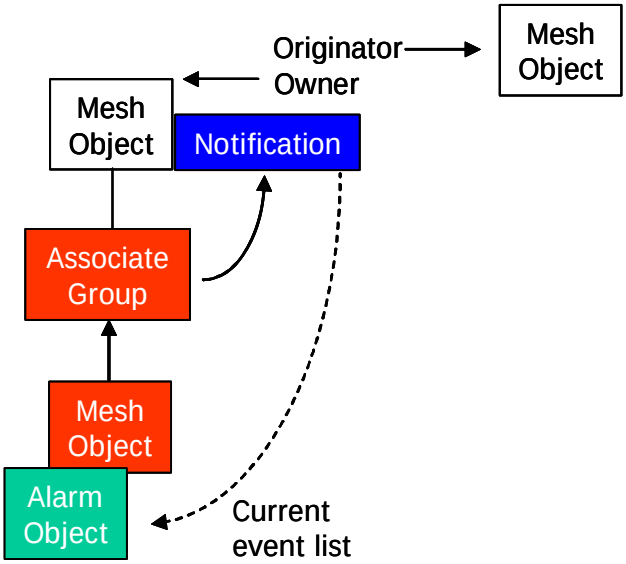
The object that originates the new notification object.

(Optional) message to be displayed in the notification report on the Notification Viewer GUI.

Type and rank of notification object to create

Remove Notification Against Referenced Associate Group Parent

State Mesh Model



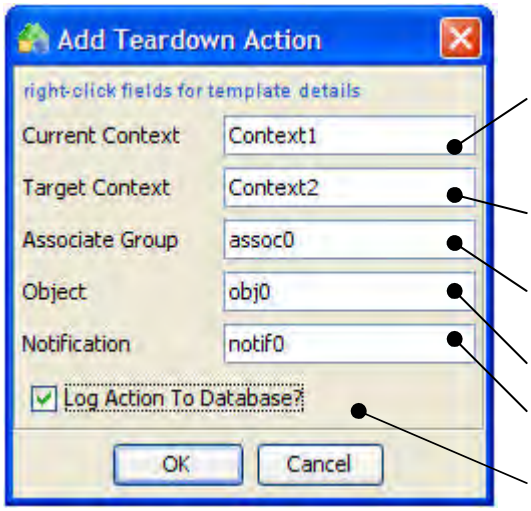
Fired Rules Viewer Mnemonic

tearRemoveNotRefAssocGrpParent

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action closes the notification record in the database associated with the supplied primary notification object. The status of the notification report associated with the supplied notification object is automatically set to 'closed' on the Notification Viewer GUI.
The supplied notification object is detached from the supplied associate group's parent (mesh) object and removed from the current context (working memory) & (if used) optional target context. The notification object is then destroyed.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied mesh object, associate group and notification object are inserted.

An alternative context in which the supplied notification object may also inserted (if un-used, set as Current Context).

The associate group whose parent (mesh) object owns the supplied notification object.

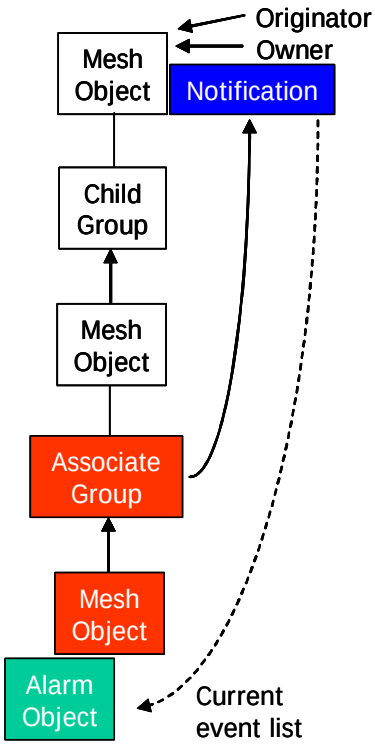
The (mesh) object that originates the notification object.

The notification object to be removed and destroyed.

Option to record action execution details in the database.

Create Notification Against Associate Group Grandparent

State Mesh Model



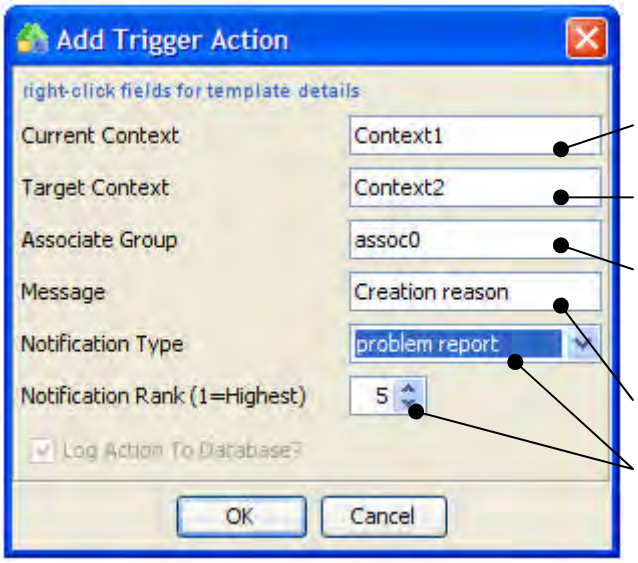
Fired Rules Viewer Mnemonic

trigCreateNotAssocGrpGparent

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action builds a contributory event list in the database from the active alarm reports attached to the (mesh) objects contained in the supplied associate group, creates a notification record in the database and attaches the contributory event list to it.
An active notification report with a list of contributory events (alarm reports) is automatically displayed on the Notification Viewer GUI.
A new notification object (of the requested type and rank) is created and is inserted into the current context (working memory) and an optional target context. Note that both the originating and owning object references in the notification object are set to the associate group's grandparent object (it is a primary notification object). The current event list is also initialised with the contents of the contributory events list.

Scenario Manager Configuration Dialogue

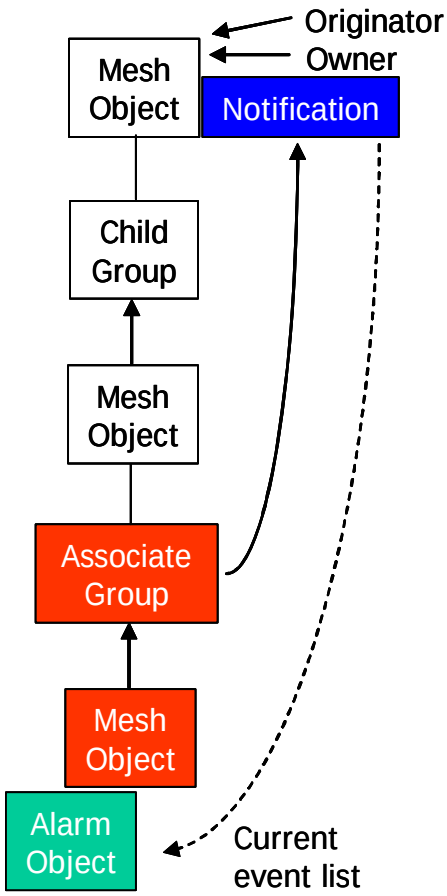


The context (working memory) in which the triggering rule is deployed, where the supplied associate group is inserted and where the new notification object will be inserted.

An alternative context in which the new notification object may also inserted (if un-used, set as Current Context).
The associate group whose member objects will provide zero or more active alarm reports and whose grandparent object both originates and owns the new notification object.
(Optional) message to be displayed in the notification report on the Notification Viewer GUI.
Type and rank of notification object to create

Remove Notification Against Associate Group Grandparent

State Mesh Model



Fired Rules Viewer Mnemonic

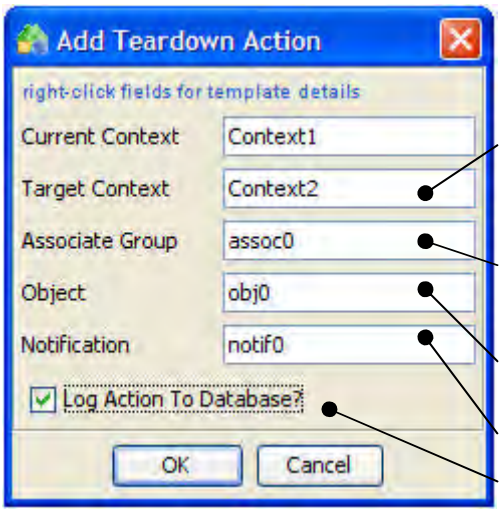
tearRemoveNotAssocGrpGparent

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action closes the notification record in the database associated with the supplied primary notification object.

The status of the notification report associated with the supplied notification object is automatically set to 'closed' on the Notification Viewer GUI.
The supplied notification object is detached from the supplied associate group's grandparent (mesh) object and removed from the current context (working memory) & (if used) optional target context. The notification object is then destroyed.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied associate group & notification objects are inserted.

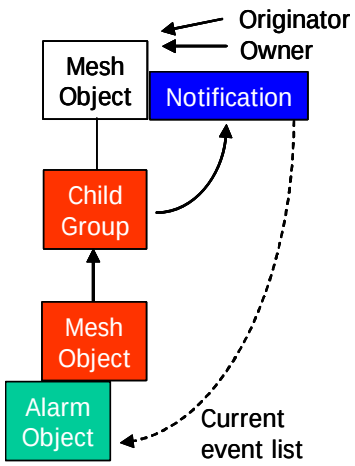
An alternative context in which the supplied notification object may also inserted (if un-used, set as Current Context).

The associate group whose grandparent (mesh) object owns the supplied notification object.
The notification object to be removed and destroyed.

Option to record action execution details in the database.

Create Notification Against Child Group Parent

State Mesh Model



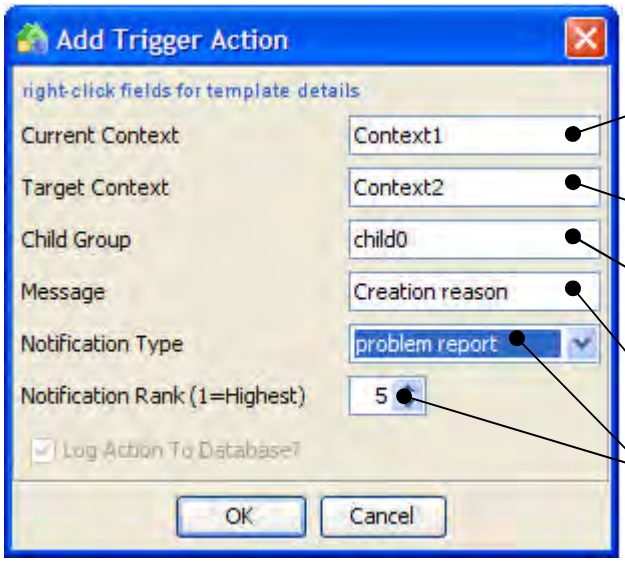
Fired Rules Viewer Mnemonic

trigCreateNotChildGrpParent

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action builds a contributory event list in the database from the active alarm reports attached to the (mesh) objects contained in the supplied child group, creates a notification record in the database and attaches the contributory event list to it.
An active notification report with a list of contributory events (alarm reports) is automatically displayed on the Notification Viewer GUI.
A new notification object (of the requested type and rank) is created and is inserted into the current context (working memory) and an optional target context. Note that both the originating and owning object references in the notification object are set to the child group's parent object (it is a primary notification object). The current event list is also initialised with the contents of the contributory events list.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed, where the supplied child group is inserted and where the new notification object will be inserted.

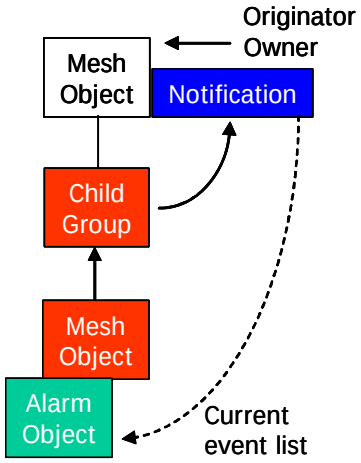
An alternative context in which the new notification object may also inserted (if un-used, set as Current Context).

The child group whose member objects will provide zero or more active alarm reports and whose parent object both originates and owns the new notification object.
(Optional) message to be displayed in the notification report on the Notification Viewer GUI.

Type and rank of notification object to create

Update Notification Against Child Group Parent

State Mesh Model



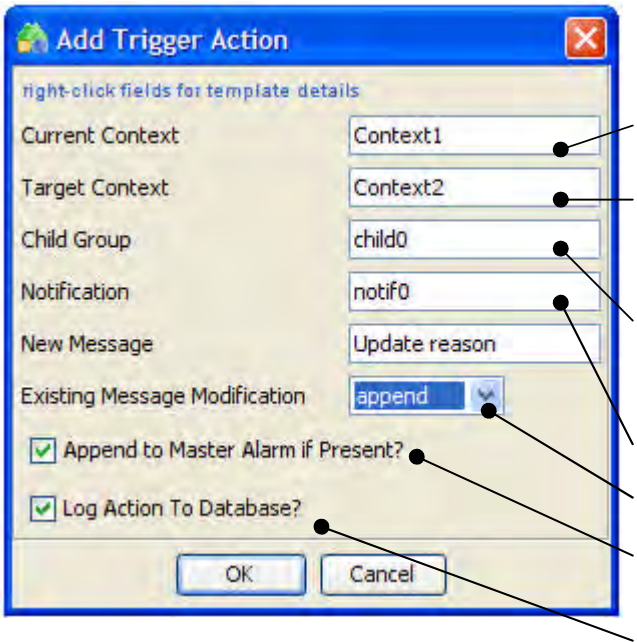
Fired Rules Viewer Mnemonics

trigUpdateNotChildGrpParent
tearUpdateNotChildGrpParent

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action updates the contributory event list attached to the notification record in the database for the supplied notification object, using the latest new active alarm report attached to each (mesh) object contained in the supplied child group.
The contributory events list of the notification report associated with the supplied notification object is automatically updated with the new alarm report on the Notification Viewer GUI.
The event list count and trend attributes of the supplied notification object are updated in the current context (working memory) & (if used) optional target context. The current event list is also updated in line with the contents of the contributory events list.
Optionally, the message to be displayed in the notification report on the Notification Viewer GUI may be replaced or additional information may be appended.
Optionally (and if it is present), the Master Alarm associated with the supplied notification object may be updated with the details of the latest active alarm reports attached to each (mesh) object contained in the supplied child group.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied child group & notification objects are inserted.

An alternative context in which the supplied notification object may also inserted (if un-used, set as Current Context).

The child group containing one or more objects which may provide their latest new active alarm report, whose parent object both originates and owns the supplied notification object.

The notification object to be updated.

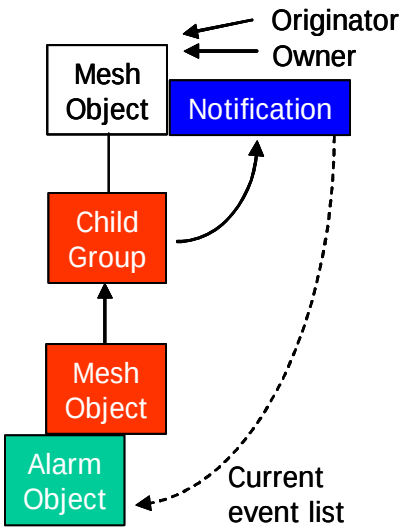
Message modification options {unchanged|append|replace}

Option to append the latest active alarm reports to the Master Alarm associated with the notification (if present)

Option to record action execution details in the database.

Remove Notification Against Child Group Parent

State Mesh Model



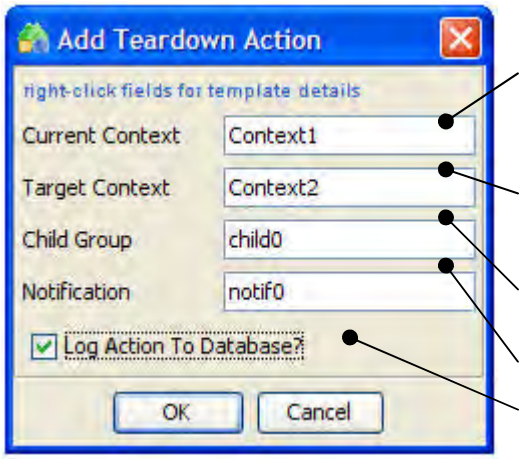
Fired Rules Viewer Mnemonic

tearRemoveNotChildGrpParent

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action closes the notification record in the database associated with the supplied primary notification object. The status of the notification report associated with the supplied notification object is automatically set to 'closed' on the Notification Viewer GUI.
The supplied notification object is detached from the supplied child group's parent (mesh) object and removed from the current context (working memory) & (if used) optional target context. The notification object is then destroyed.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied child group & notification objects are inserted.

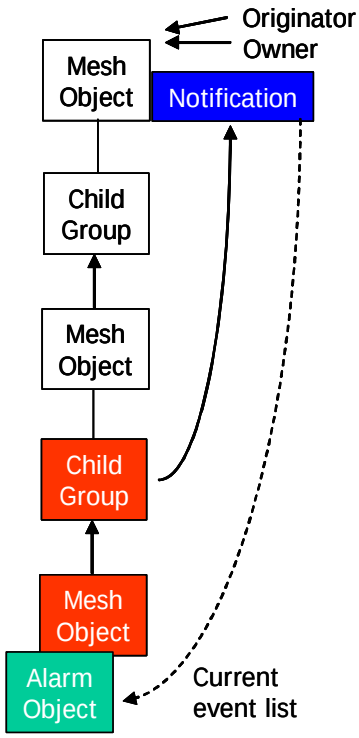
An alternative context in which the supplied notification object may also inserted (if un-used, set as Current Context).

The child group whose parent (mesh) object owns the supplied notification object.
The notification object to be removed and destroyed.

Option to record action execution details in the database.

Create Notification Against Child Group Grandparent

State Mesh Model



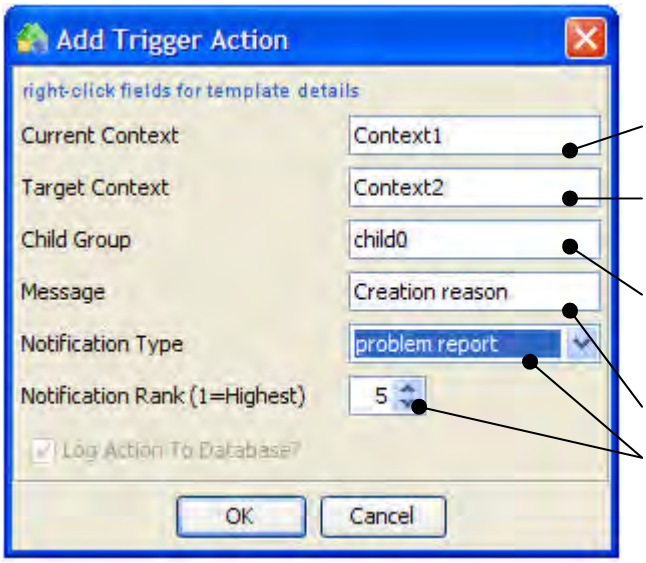
Fired Rules Viewer Mnemonic

trigCreateNotChildGrpGparent

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action builds a contributory event list in the database from the active alarm reports attached to the (mesh) objects contained in the supplied child group, creates a notification record in the database and attaches the contributory event list to it.
An active notification report with a list of contributory events (alarm reports) is automatically displayed on the Notification Viewer GUI.
A new notification object (of the requested type and rank) is created and is inserted into the current context (working memory) and an optional target context. Note that both the originating and owning object references in the notification object are set to the child group's grandparent object (it is a primary notification object). The current event list is also initialised with the contents of the contributory events list.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed, where the supplied child group is inserted and where the new notification object will be inserted.

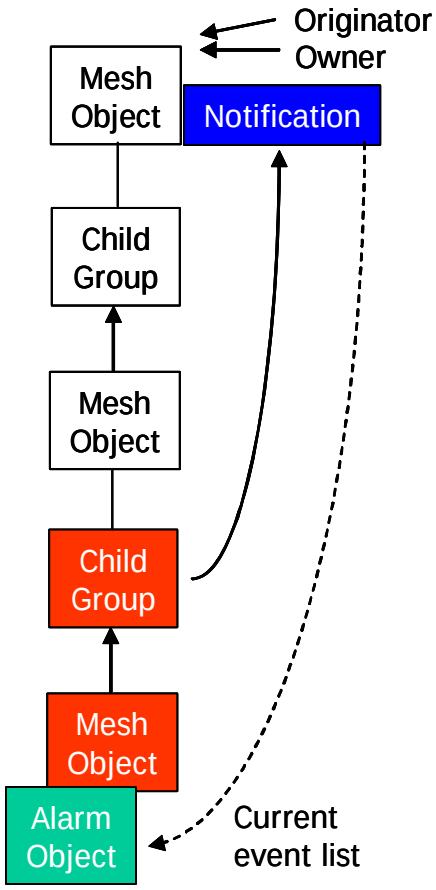
An alternative context in which the new notification object may also inserted (if un-used, set as Current Context).

The child group whose member objects will provide zero or more active alarm reports and whose grandparent object both originates and owns the new notification object.

(Optional) message to be displayed in the notification report on the Notification Viewer GUI.
Type and rank of notification object to create

Remove Notification Against Child Group Grandparent

State Mesh Model



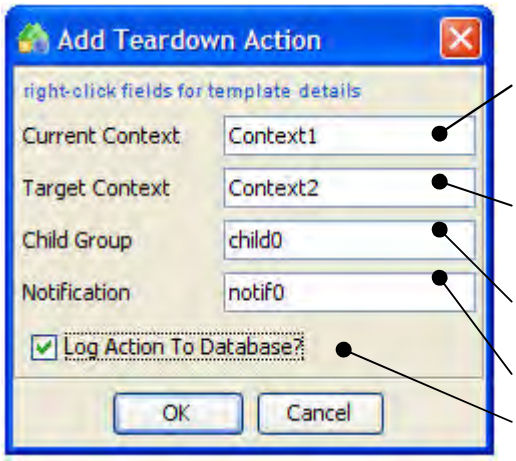
Fired Rules Viewer Mnemonic

tearRemoveNotChildGrpGparent

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action closes the notification record in the database associated with the supplied primary notification object. The status of the notification report associated with the supplied notification object is automatically set to 'closed' on the Notification Viewer GUI.
The supplied notification object is detached from the supplied child group's grandparent (mesh) object and removed from the current context (working memory) & (if used) optional target context. The notification object is then destroyed.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied child group & notification objects are inserted.

An alternative context in which the supplied notification object may also inserted (if un-used, set as Current Context).

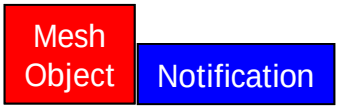
The child group whose grandparent (mesh) object owns the supplied notification object.

The notification object to be removed and destroyed.

Option to record action execution details in the database.

Force Removal Of Notification Against Object

State Mesh Model



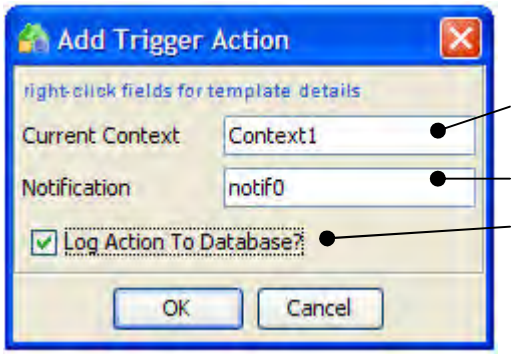
Fired Rules Viewer Mnemonic

trigForceRemNotMO
tearForceRemNotMO

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action closes the notification record in the database associated with the supplied primary notification object.
The status of the notification report associated with the supplied notification object is automatically set to 'closed' on the Notification Viewer GUI.
The supplied notification object is removed from the current context (working memory) and is then destroyed

Scenario Manager Configuration Dialogue



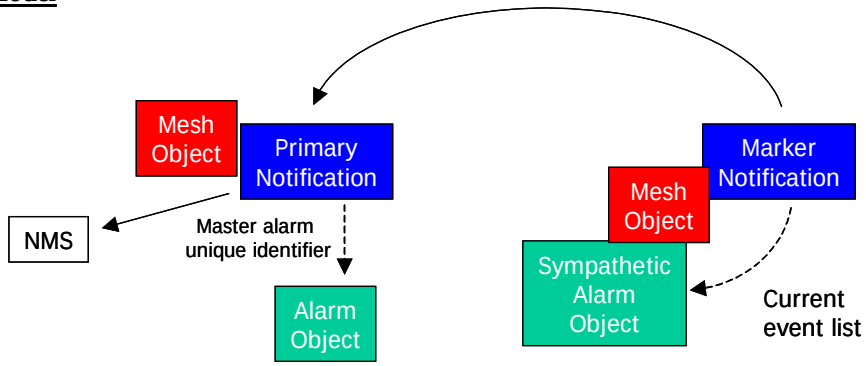
The context (working memory) in which the triggering rule is deployed and the supplied notification object is inserted.

The notification object to be removed.

Option to record action execution details in the database.

Append Event To Notification Sympathetic Event List

State Mesh Model



Fired Rules Viewer Mnemonic

trigAppEventNotSymList

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
The purpose of this action is to associate the latest sympathetic alarm report in the supplied (mesh) object's current event list with a 'master' alarm report in an external NMS.
The action also supports the option to build this association only if the latest sympathetic alarm report's EMS originating time (or alternatively its creation time in this system) lies within a configurable time exclusion window either side of the primary notification object's creation time
This action updates the current problem list size in the supplied marker notification (attached to the supplied object).
If the sympathetic alarm report lies within the exclusion time window (or the exclusion time window is not used):

The sympathetic alarm report is also added to a sympathetic event list attached to the notification record in the database associated with the supplied primary notification object.
The sympathetic alarm list in the notification report associated with the primary notification object is automatically updated on the Notification Viewer GUI.
If the option to append the sympathetic alarm report to an existing Master Alarm is chosen (and the Master Alarm is present):

A sympathetic alarm report request (including the 'master' alarm report external NMS reference) is sent to the external NMS via the Remote Handler's REPORT_SYMPATHETIC_ALARMS callout function. The effect in the external NMS depends on the level of integration and its inherent capabilities.

If the sympathetic alarm enrichment option is chosen, a list of the primary notification's contributory alarm report external NMS references, together with the sympathetic alarm report's external NMS reference is sent to the external NMS via the Remote Handler's ENRICH_SYMPATHETIC_ALARMS callout function. The effect in the external NMS depends on the level of integration and its inherent capabilities.

On successful completion of the action, the 'child alarms demoted' attribute in the supplied marker notification object is set to true and this may be evaluated by additional rules.

Scenario Manager Configuration Dialogue

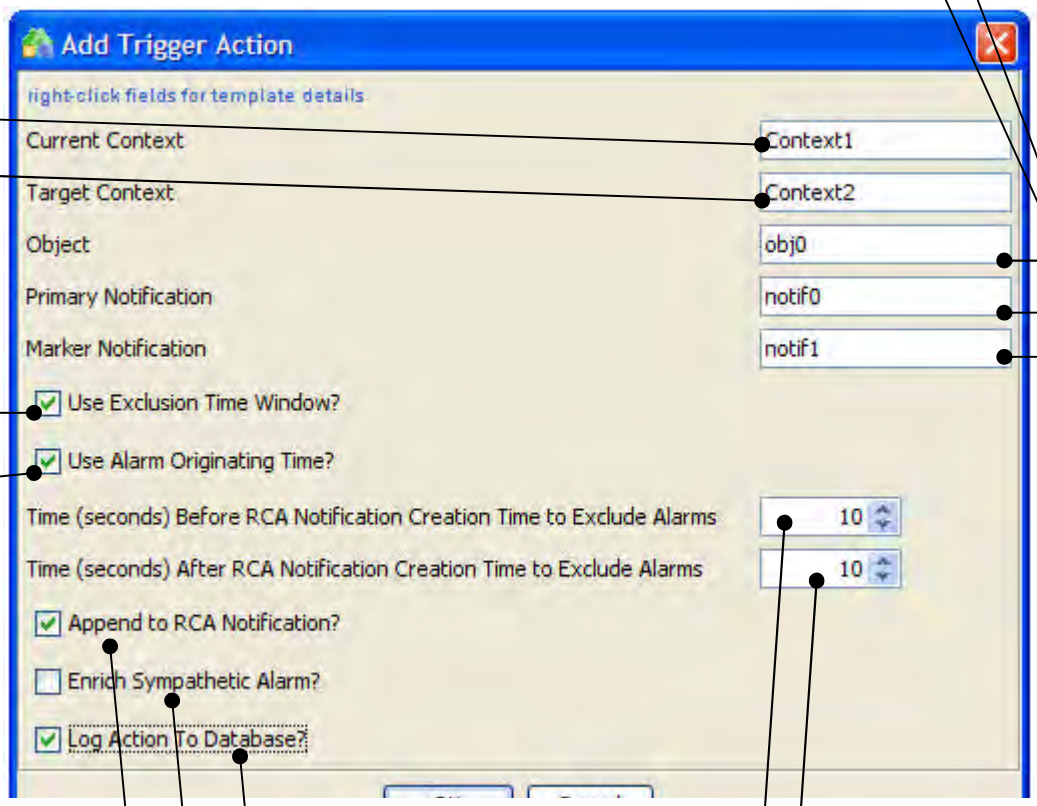
The context (working memory) in which the triggering rule is deployed and where the mesh and primary & marker notification objects are inserted.

An alternative context in which the mesh and primary & marker notification objects may also be inserted (if un-used, set as Current Context).

The mesh object whose current event list contains the latest sympathetic alarm report.

The primary notification object containing the external NMS 'master' alarm report reference.

The marker notification object whose current event list is updated with the latest sympathetic alarm report.



Option to use time exclusion window.

Option to use the sympathetic alarm report's EMS originating time (checked) or the UCA creation time (unchecked) in conjunction with the time exclusion window.

Time exclusion window early limit (in seconds before primary notification creation time).
Time exclusion window late limit (in seconds after primary notification creation time).

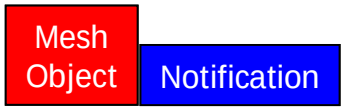
Option to append sympathetic alarm report to primary notification master alarm if present.

Option to enrich primary notification contributory alarm reports with details of sympathetic alarm report.

Option to record action execution details in the database.

Update Notification Rank

State Mesh Model



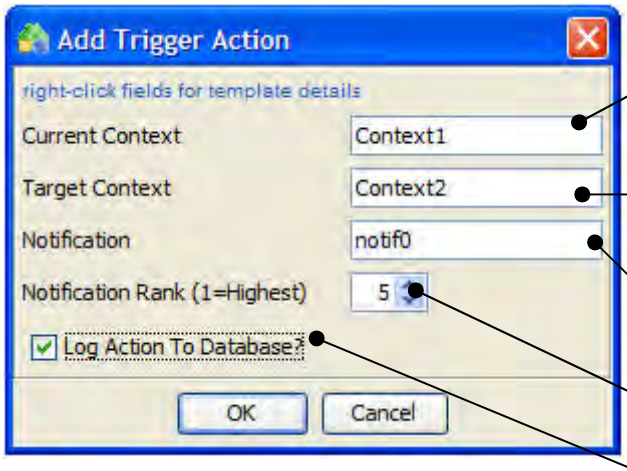
Fired Rules Viewer Mnemonic

trigUpdateNotRank
tearUpdateNotRank

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action updates the rank of the notification record in the database associated with the supplied primary notification object.
The rank of the notification report associated with the supplied notification object is automatically updated on the Notification Viewer GUI.
The rank of the supplied notification object is updated in the current context (working memory) and in the optional target context (if used)

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied child group & notification objects are inserted

An alternative context in which the supplied notification object may also inserted (if un-used, set as Current Context).

The notification object whose rank is to be updated.

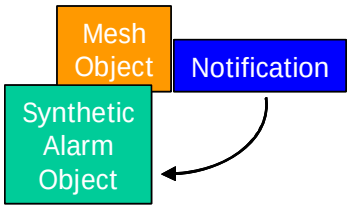
Updated rank value

Option to record action executino details in the database.

15.2.2.3 State Propagation

Force Object To Degraded State Via Notification

State Mesh Model



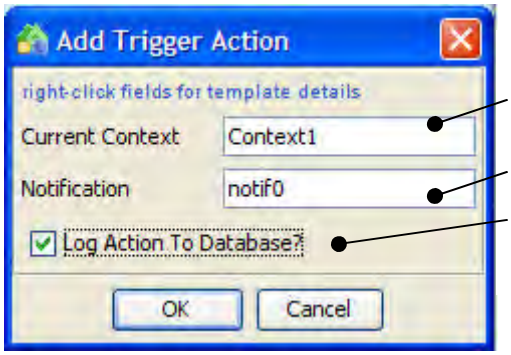
Fired Rules Viewer Mnemonic

trigForceMODegViaNotif

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to locate the (mesh) object that owns the supplied notification. It then creates and attaches a synthetic alarm report to the object (with a target state of degraded) to attempt to force it to the degraded state. Note that the object may not actually change state if it is already degraded or failed; however the synthetic alarm report will remain attached and may affect the future state of the object as other attached alarm reports are cleared.

Scenario Manager Configuration Dialogue



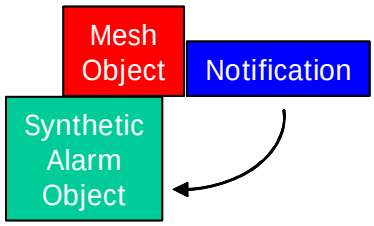
The context (working memory) in which the triggering rule is deployed and the supplied notification object is inserted.

The notification object owned by the target object.

Option to record action execution details in the database.

Force Object To Failed State Via Notification

State Mesh Model



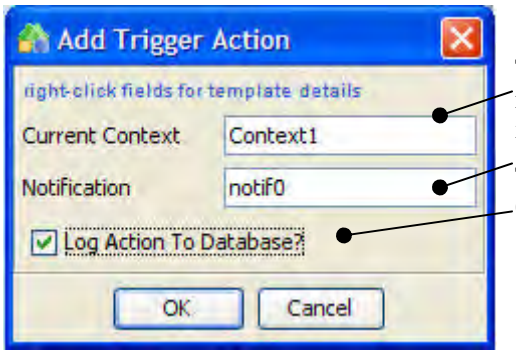
Fired Rules Viewer Mnemonic

trigForceMOFailedViaNotif

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to locate the (mesh) object that owns the supplied notification. It then creates and attaches a synthetic alarm report to the object (with a target state of failed) to attempt to force it to the failed state.
Note that the object may not actually change state if it is already failed; however the synthetic alarm report will remain attached and may affect the future state of the object as other attached alarm reports are cleared.

Scenario Manager Configuration Dialogue



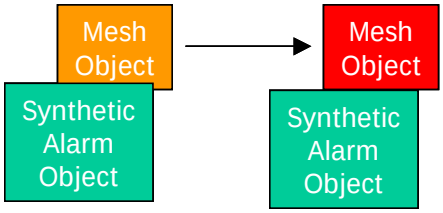
The context (working memory) in which the triggering rule is deployed and the supplied notification object is inserted.

The notification object owned by the target object.

Option to record action execution details in the database.

Force Degraded Object To Failed State

State Mesh Model



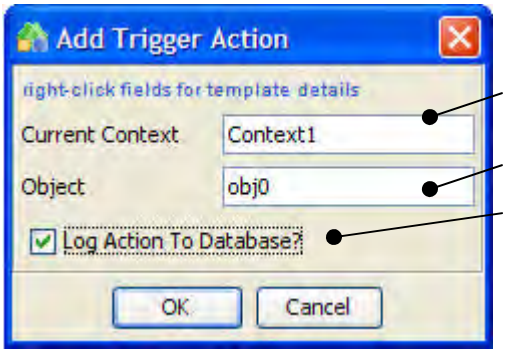
Fired Rules Viewer Mnemonic

trigForceMOStateChange

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action creates and attaches a synthetic alarm report to the (mesh) object (with a target state of failed) to attempt to force it to the failed state. A common use of this action is to force an already degraded object to the failed state.
Note that the object may not actually change state if it is already failed; however the synthetic alarm report will remain attached and may affect the future state of the object as other attached alarm reports are cleared.

Scenario Manager Configuration Dialogue



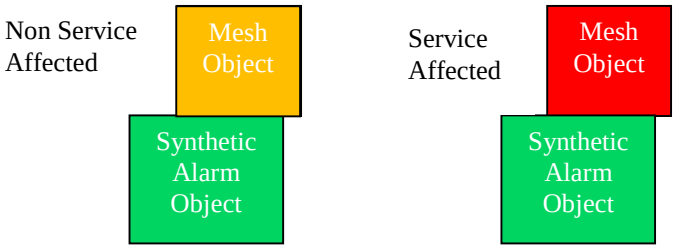
The context (working memory) in which the triggering rule is deployed and the supplied mesh object is inserted.

The mesh object to be forced to the failed state.

Option to record action execution details in the database.

Force Named Object To Change State

State Mesh Model



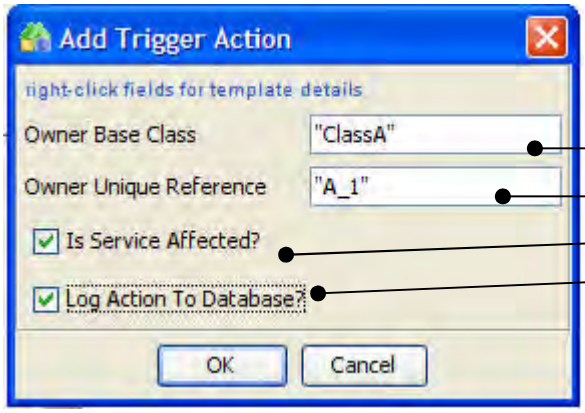
Fired Rules Viewer Mnemonic

trigForceNamedMO

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
The purpose of this action is to attempt to force a state change on a (mesh) object for which the triggering rule does not have an existing (mesh) object reference and so has to provide an explicit class and instance name.
The action first verifies that the explicitly named (mesh) object currently exists in the system. If it does not exist, then an exception is reported and the action is aborted.
If the 'Is Service Affected' option is chosen, this action creates and attaches a synthetic alarm report to the (mesh) object (with a target state of failed) to attempt to force it to the failed state.
If the 'Is Service Affected' option is not chosen, this action creates and attaches a synthetic alarm report to the (mesh) object (with a target state of degraded) to attempt to force it to the degraded state.
Note that the object may not actually change state if it is already failed; however the synthetic alarm report will remain attached and may affect the future state of the object as other attached alarm reports are cleared.

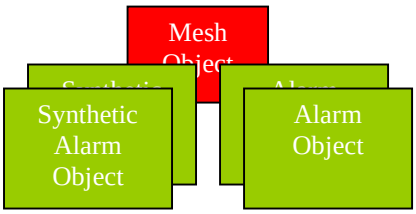
Scenario Manager Configuration Dialogue



The base class of the mesh object to be forced to change state (literal or stored in a rule variable).
The unique reference of the mesh object to be forced to change state (literal or stored in a rule variable).
Option to treat as 'service affecting'
Option to record action execution details in the database.

Reset Object to Normal State

State Mesh Model



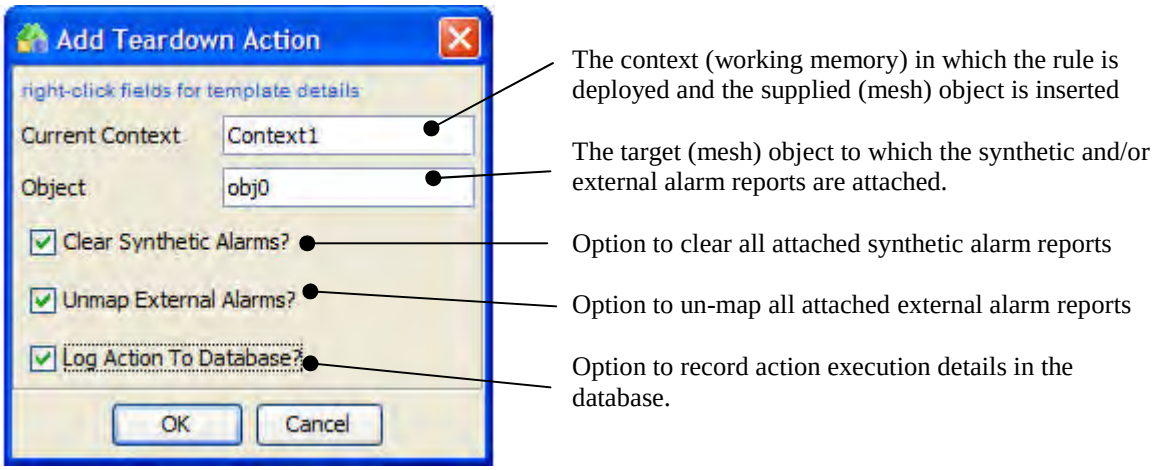
Fired Rules Viewer Mnemonic

tearForceFailedMONormState

Summary

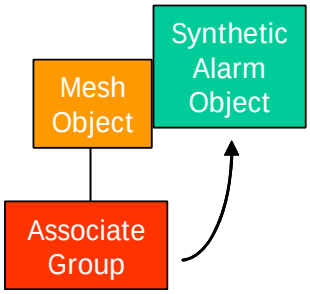
If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
If the option is selected, this action attempts to clear all synthetic alarm reports (with a target state of degraded and/or failed) from the supplied (mesh) object.
If the option is selected, this action attempts to un-map i.e. remove from the Current Problem List, any associated external alarm reports.
If no alarm reports remain in the objects Current Problem List, it will automatically return to the normal state.

Scenario Manager Configuration Dialogue



Force Parent Object To Degraded State Via Associate Group

State Mesh Model



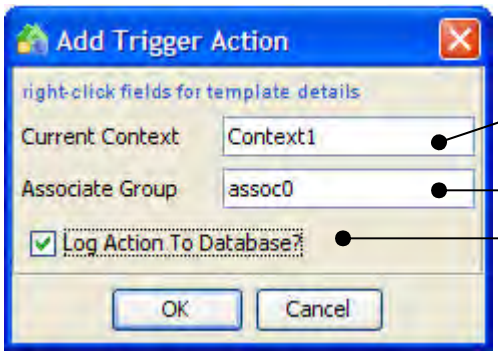
Fired Rules Viewer Mnemonic

trigForceParentDegViaAssoc

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to locate the parent (mesh) object that owns the supplied associate group. It then creates and attaches a synthetic alarm report to the object (with a target state of degraded) to attempt to force it to the degraded state.
Note that the object may not actually change state if it is already degraded or failed; however the synthetic alarm report will remain attached and may affect the future state of the object as other attached alarm reports are cleared.

Scenario Manager Configuration Dialogue



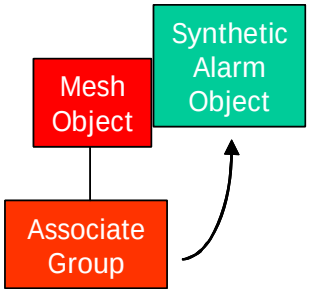
The context (working memory) in which the triggering rule is deployed and the supplied associate group is inserted.

The associate group owned by the target object.

Option to record action execution details in the database.

Force Parent Object To Failed State Via Associate Group

State Mesh Model



Fired Rules Viewer Mnemonic

trigForceParentFailedViaAssoc

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to locate the parent (mesh) object that owns the supplied associate group. It then creates and attaches a synthetic alarm report to the object (with a target state of failed) to attempt to force it to the failed state.
Note that the object may not actually change state if it is already failed; however the synthetic alarm report will remain attached and may affect the future state of the object as other attached alarm reports are cleared.

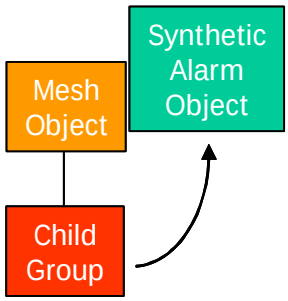
Scenario Manager Configuration Dialogue

The context (working memory) in which the triggering rule is deployed and the supplied associate group is inserted.

The associate group owned by the target object.
Option to record action execution details in the database.

Force Parent Object To Degraded State Via Child Group

State Mesh Model



Fired Rules Viewer Mnemonic

trigForceParentDegViaChild

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to locate the parent (mesh) object that owns the supplied child group. It then creates and attaches a synthetic alarm report to the object (with a target state of degraded) to attempt to force it to the degraded state.
Note that the object may not actually change state if it is already degraded or failed; however the synthetic alarm report will remain attached and may affect the future state of the object as other attached alarm reports are cleared.

Scenario Manager Configuration Dialogue

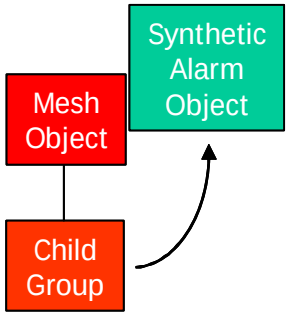
The context (working memory) in which the triggering rule is deployed and the supplied child group is inserted.

The child group owned by the target object.

Option to record action execution details in the database.

Force Parent Object To Failed State Via Child Group

State Mesh Model



Fired Rules Viewer Mnemonic

trigForceParentFailedViaChild

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to locate the parent (mesh) object that owns the supplied child group. It then creates and attaches a synthetic alarm report to the object (with a target state of failed) to attempt to force it to the failed state.
Note that the object may not actually change state if it is already failed; however the synthetic alarm report will remain attached and may affect the future state of the object as other attached alarm reports are cleared.

Scenario Manager Configuration Dialogue

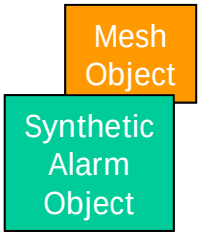
The context (working memory) in which the triggering rule is deployed and the supplied child group is inserted.

The child group owned by the target object.

Option to record action execution details in the database.

Force DegradedObject To Normal State

State Mesh Model



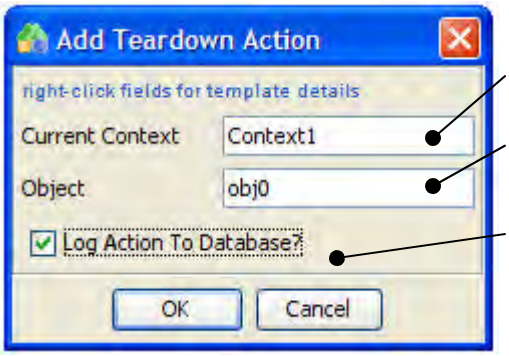
Fired Rules Viewer Mnemonic

tearForceDegMONormState

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to clear a synthetic alarm report (with a target state of degraded) from the supplied (mesh) object.
Following clearance and if no other alarm reports with degraded or failed target state are attached to the supplied object, it will automatically return to the normal state.

Scenario Manager Configuration Dialogue



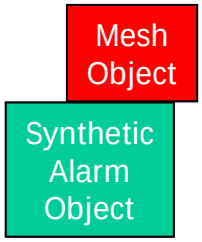
The context (working memory) in which the triggering rule is deployed and the supplied (mesh) object is inserted.

The target (mesh) object to which at least one synthetic alarm report with a target state of degraded is attached.

Option to record action execution details in the database.

Force Failed Object To Normal State

State Mesh Model



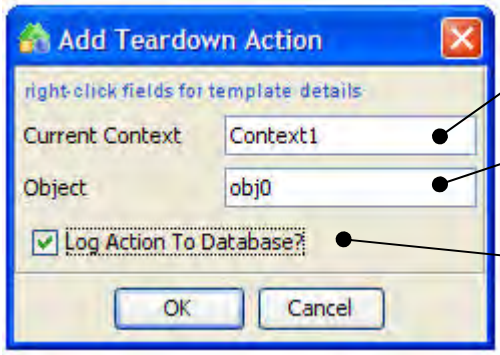
Fired Rules Viewer Mnemonic

tearForceFailedMONormState

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to clear a synthetic alarm report (with a target state of failed) from the supplied (mesh) object.
Following clearance and if no other alarm reports with degraded or failed target state are attached to the supplied object, it will automatically return to the normal state.

Scenario Manager Configuration Dialogue



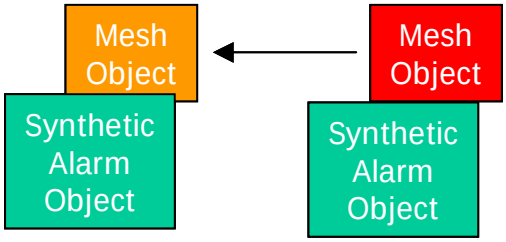
The context (working memory) in which the triggering rule is deployed and the supplied (mesh) object is inserted.

The target (mesh) object to which at least one synthetic alarm report with a target state of failed is attached.

Option to record action execution details in the database.

Forced Failed Object To Degraded State

State Mesh Model



Fired Rules Viewer Mnemonic

tearForceMOStateChange

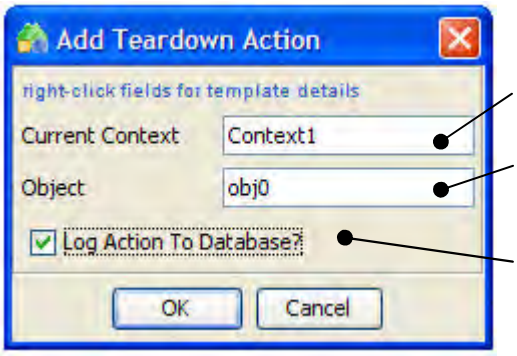
Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to clear a synthetic alarm report (with a target state of failed) from the supplied (mesh) object. It is commonly used to return an object that also has a synthetic alarm report (with a target state of degraded) to the degraded state.
Following clearance and if:

- at least one alarm report with a target state of degraded is attached to the target object;
- no other alarm reports with a target state of failed are attached to the supplied object;

It will automatically return to the degraded state.

Scenario Manager Configuration Dialogue



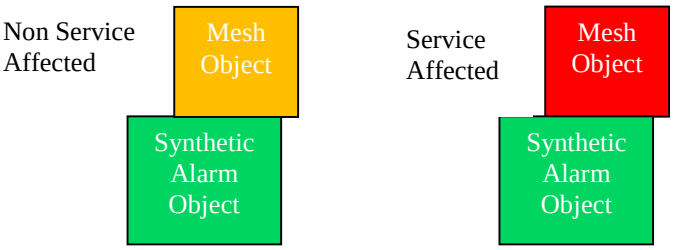
The context (working memory) in which the triggering rule is deployed and the supplied (mesh) object is inserted.

The target (mesh) object to which at least two synthetic alarm reports with target states of degraded and failed are attached.

Option to record action execution details in the database.

Force Named Object To Normal State

State Mesh Model



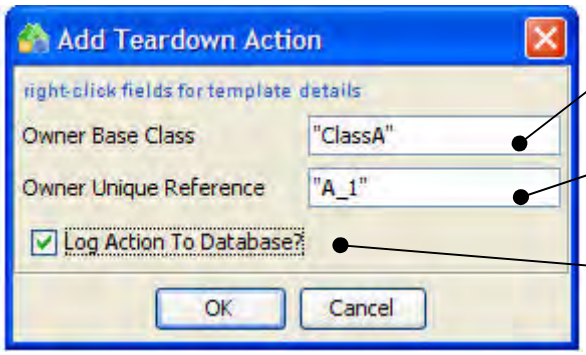
Fired Rules Viewer Mnemonic

trigForceNamedMO

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
The purpose of this action is to attempt to force a state change back to the normal state on a (mesh) object for which the triggering rule does not have an existing (mesh) object reference and so has to provide an explicit class and instance name.
The action first verifies that the explicitly named (mesh) object currently exists in the system. If it does not exist, then an exception is reported and the action is aborted.
If the (mesh) object is in the failed state, this action sends a clear failed synthetic alarm report to the (mesh) object to attempt to force it to the normal state.
If the (mesh) object is in the degraded state, this action sends a clear degraded synthetic alarm report to the (mesh) object to attempt to force it to the degraded state.
Note that the object may not actually change state if other non-normal alarm reports are associated with it; however the system will attempt to clear and remove one synthetic alarm report of the specified severity.

Scenario Manager Configuration Dialogue



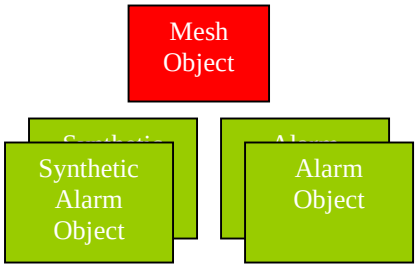
The base class of the mesh object to be forced to the normal state (literal or stored in a rule variable).

The unique reference of the mesh object to be forced to the normal state (literal or stored in a rule variable).

Option to record action execution details in the database.

Reset Object to Normal State

State Mesh Model



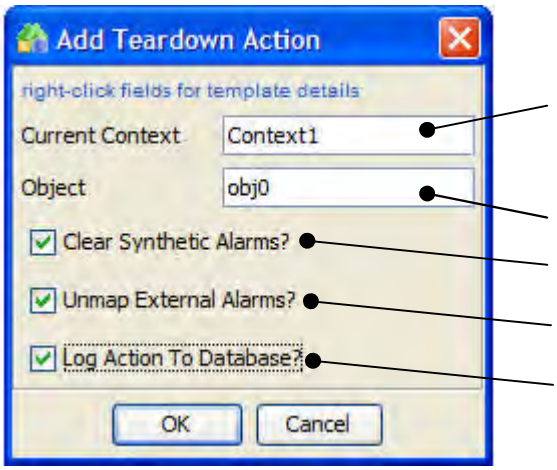
Fired Rules Viewer Mnemonic

tearForceFailedMONormState

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
If the option is selected, this action attempts to clear all synthetic alarm reports (with a target state of degraded and/or failed) from the supplied (mesh) object.
If the option is selected, this action attempts to un-map i.e. remove from the Current Problem List, any associated external alarm reports.
If no alarm reports remain in the objects Current Problem List, it will automatically return to the normal state.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied (mesh) object is inserted.

The target (mesh) object to which the synthetic and/or external alarm reports are attached.

Option to clear all attached synthetic alarm reports

Option to un-map all attached external alarm reports

Option to record action execution details in the database.

Script Handling

Run Script

State Mesh Model

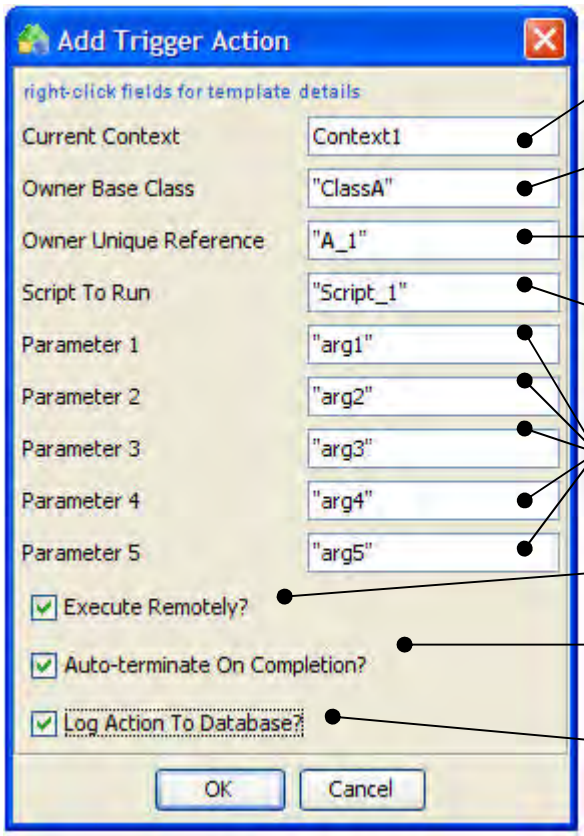
Fired Rules Viewer Mnemonics

trigRunScript
tearRunScript

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action builds a new script object in the current context (working memory). The script object acts as a proxy object for the actual script and to preserve concurrency automatically executes the requested script in a separate thread of execution. Depending on configuration, the actual script may be executed directly by the Notification Manager on the host platform or remotely via an instance of the Remote Handler (using the RUN_SCRIPT callout function) which in turn may be running on the local and/or remote platforms.
Once the script has finished executing (and again depending on configuration) the script object may remain in the current context. At this point completion status, normal and error outputs and return codes are available to be evaluated by rules deployed in the current context.
Alternatively, the script object may be automatically removed from the current context and destroyed on script completion.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed.

The base class (literal or stored in a rule variable) of the owning mesh object.
The unique reference (literal or stored in a rule variable) of the owning mesh object.

The name of the script file to execute. The file must be executable and reside in the UCA_HOME/scripts directory on the target platform.

Optional arguments (literal values or stored in rule variables). Gaps in the argument list are not supported - use “”.

Option to execute the script remotely via the Remote Handler on a local and/or remote platform.

Option to automatically remove & destroy the script object from the current context on script completion.

Option to record action execution details in the database.

End Script

State Mesh Model

Script

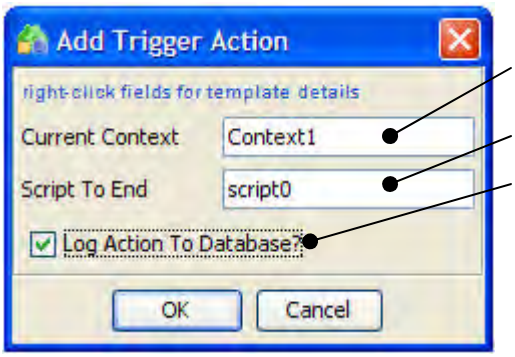
Fired Rules Viewer Mnemonics

trigEndScript
tearEndScript

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action removes the supplied script object from the current context (working memory) and terminates the thread of execution it is running in. The script object is then destroyed.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied script object is inserted.

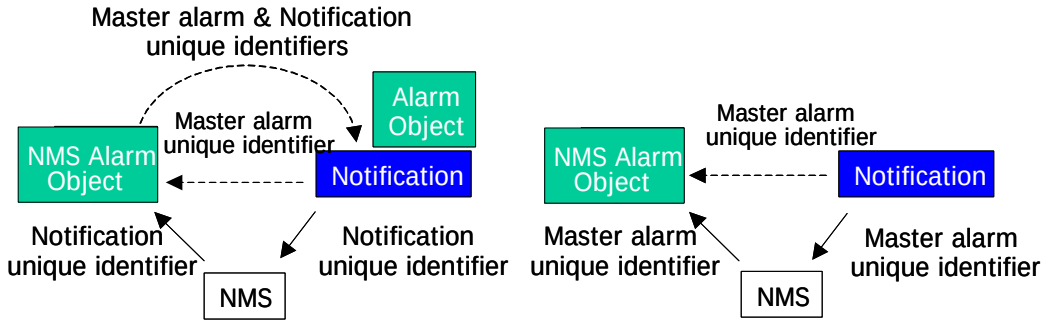
The script object to terminate and remove.

Option to record action execution details in the database.

15.2.2.5 Alarm Handling

Raise Alarm

State Mesh Model



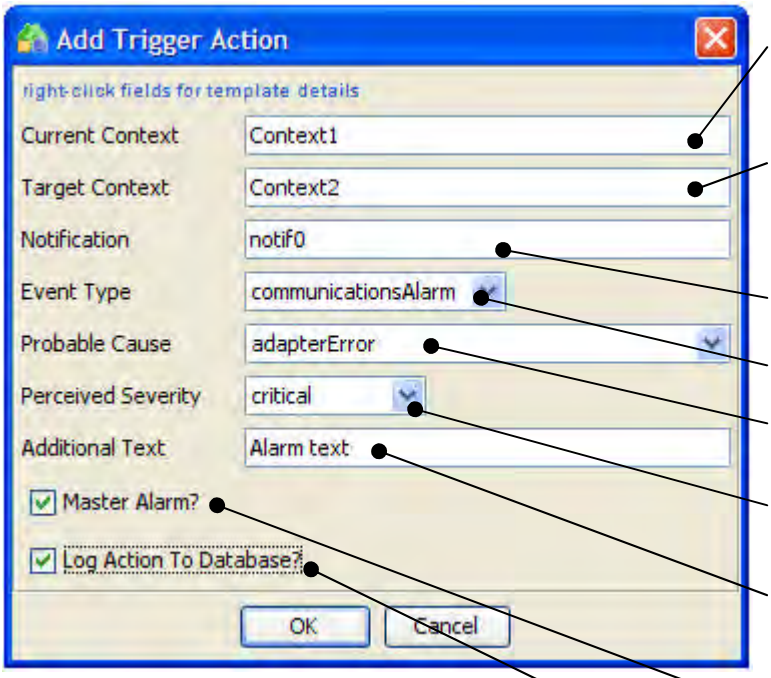
Fired Rules Viewer Mnemonics

trigRaiseAlarm
tearRaiseAlarm

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
If the option to create a master alarm report in the NMS is chosen, the master alarm PENDING option is set in the supplied notification object and it is updated in the chosen context(s).
This action submits an alarm creation request to the external NMS via the Remote Handler RAISE_ALARM callout function. The alarm creation request is targeted at the (mesh) object to which the supplied notification object is attached.
Depending on the level of integration with the external NMS and the intended use of the new alarm e.g. creation of a master alarm report, the unique identifier of the supplied notification object may or may not be passed in the alarm creation request.
If the external NMS subsequently delivers a master alarm report to the system in direct response to the creation request and it includes the notification object unique identifier, the master alarm report (with its own external NMS unique identifier) may be mapped (automatically or manually) directly into the originating notification object itself rather than the targeted (mesh) object.
An alternative Remote Handler integration, again triggered on receipt of a master alarm creation request, may artificially generate a system-specific external NMS master alarm report unique identifier and set this directly in the supplied notification object, without the need for the actual external master alarm report to be delivered back to the system and mapped onto the originating notification. This mechanism still requires the system to send the generated system-specific master alarm report unique identifier along with the creation request out to the external NMS, so that it can (in subsequent requests from the system) associate the generated system-specific master alarm report unique identifier with the equivalent identifier for the actual external NMS alarm report.
Regardless of integration technique, this optional ability to map generated alarms to notification objects is useful for the creation and handling of ‘master’ alarms. These are typically used to act as an artificial indicator of a problem, often on an object that may not otherwise report events. They may also act as a container for contributory and/or sympathetic alarms since the existence of this mapped ‘master’ alarm report in a notification object may be evaluated in rules using the ‘master alarm status’ attribute and further actions may attach contributory and/or sympathetic alarms to it. Existence of the ‘master’ alarm report therefore implies that the system has access to the ‘master’ alarm’s external NMS unique identifier, since it will need to issue instructions to the external NMS to carry out such operations.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied notification object is inserted.

An alternative context in which the supplied notification object may also be inserted (if un-used, set as Current Context).

The notification object attached to the targeted (mesh) object.

X.733 Event Type for the new alarm report.

X.733 Probable Cause for the new alarm report.

X.733 Perceived Severity for the new alarm report.

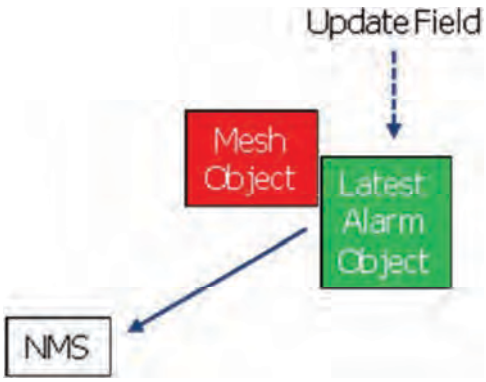
Optional Additional Text message to be inserted into the alarm report in the external NMS e.g. creation reason.

Option to create a normal or master alarm report.

Option to record action execution details in the database.

Update Alarm Field In Latest Alarm

State Mesh Model



Fired Rules Viewer Mnemonic

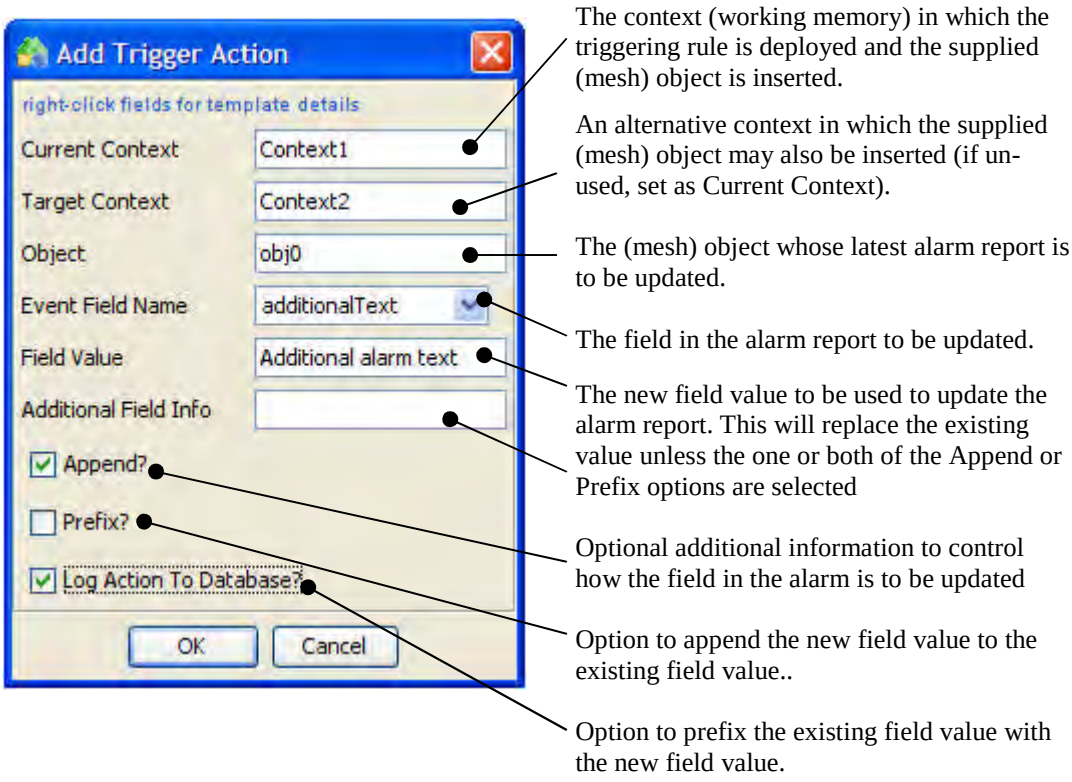
trigUpdateAlarmField

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to retrieve details of the latest alarm report from the supplied (mesh) object and if successful, an update alarm request is sent to the external NMS via the Remote Handler's UPDATE_ALARM callout function.
The update pending flag is set on the alarm object representing the alarm report and the update pending count is incremented in the supplied (mesh) object. When the alarm report update is received from the external NMS, the update pending flag is cleared on the alarm object representing the alarm report and the update pending count is decremented in the supplied (mesh) object.

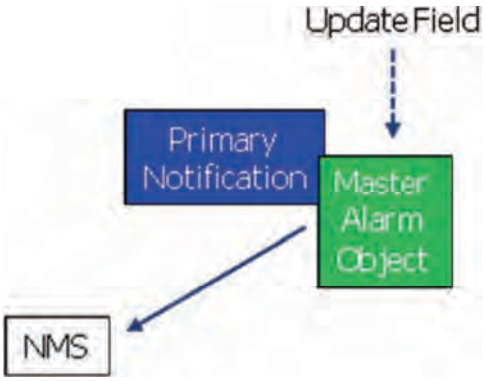
When the alarm field to be updated is chosen, the new field value entered will override the existing alarm field value unless either or both of the Append or Prefix are selected.

Scenario Manager Configuration Dialogue



Update Alarm Field In Master Alarm

State Mesh Model



Fired Rules Viewer Mnemonic

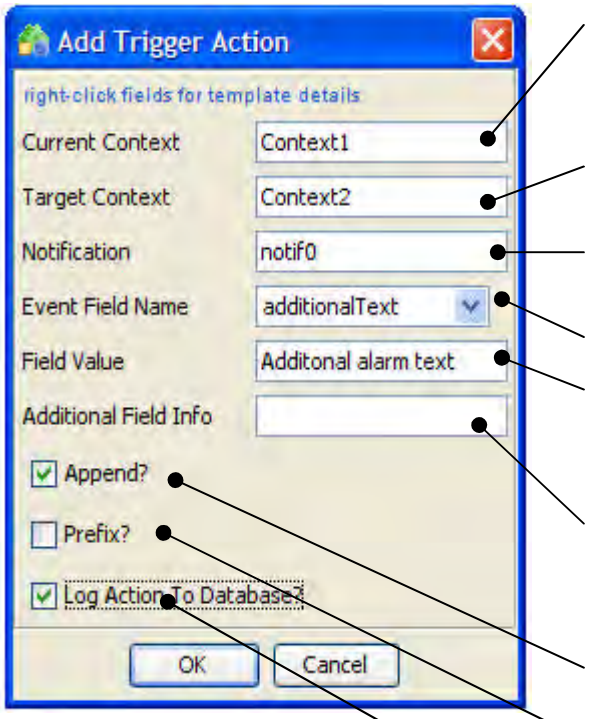
trigUpdateAlarmFieldForNotif

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to retrieve details of the master alarm report from the supplied primary notification object and if successful, an update alarm request is sent to the external NMS via the Remote Handler's UPDATE_ALARM callout function.
The update pending flag is set on the alarm object representing the master alarm report. When the master alarm report update is received from the external NMS, the update pending flag is cleared on the alarm object representing the master alarm report.

When the alarm field to be updated is chosen, the new field value entered will override the existing alarm field value unless either or both of the Append or Prefix are selected.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied notification object is inserted.

An alternative context in which the supplied notification object may also be inserted (if un-used, set as Current Context).

The notification object whose master alarm report is to be updated.

The field in the alarm report to be updated.

The new field value to be used to update the alarm report. This will replace the existing value unless one or both of the Append or Prefix options are selected

Optional additional information to control how the field in the alarm is to be updated This will replace the existing value unless the one or both of the Append or Prefix options are selected.

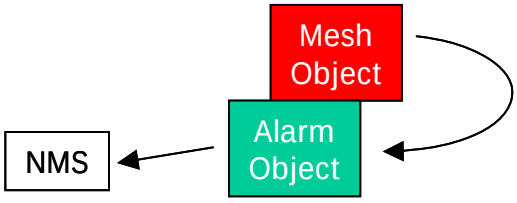
Option to append the new field value to the existing field value.

Option to prefix the existing field value with the new field value.

Option to record action execution details in the database.

Acknowledge Latest Object Alarm

State Mesh Model



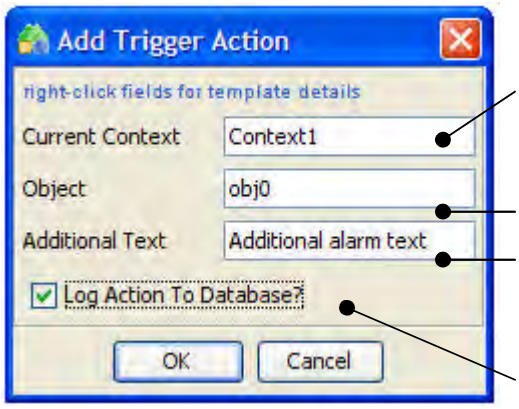
Fired Rules Viewer Mnemonics

trigAckLatestAlarm
tearAckLatestAlarm

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted. This action sends an acknowledge alarm request to the external NMS via the Remote Handler's ACKNOWLEDGE_CAUSAL_ALARM callout function. It includes the external NMS alarm report unique identifier extracted from the latest alarm report received by the supplied (mesh) object.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied mesh object is inserted.

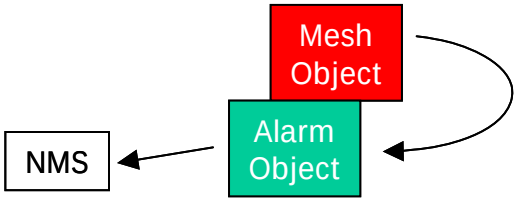
The mesh object containing the latest alarm report.

Optional Additional Text message to be inserted into the alarm report in the external NMS e.g. acknowledgement reason.

Option to record action execution details in the database.

Terminate Latest Object Alarm

State Mesh Model



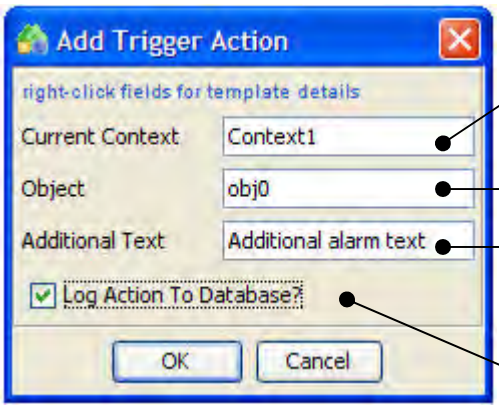
Fired Rules Viewer Mnemonic

trigTermLatestAlarm
tearTermLatestAlarm

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action sends a terminate alarm request to the external NMS via the Remote Handler's TERMINATE_CAUSAL_ALARM callout function. It includes the external NMS alarm report unique identifier extracted from the latest alarm report received by the supplied (mesh) object.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied mesh object is inserted.

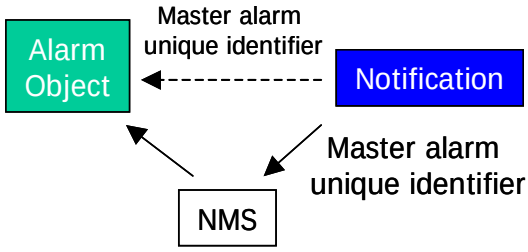
The mesh object containing the latest alarm report.

Optional Additional Text message to be inserted into the alarm report in the external NMS e.g. termination reason.

Option to record action execution details in the database.

Terminate Master Alarm

State Mesh Model



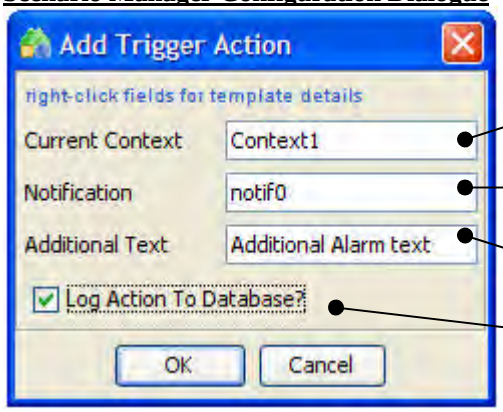
Fired Rules Viewer Mnemonic

trigTermMasterAlarm
tearTermMasterAlarm

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action examines the supplied notification object for the presence of a 'master' alarm report.
If a 'master' alarm report has never been received, a record is added to this effect in the action log in the database and processing is terminated.
If a 'master' alarm report has been received but is not attached to the supplied notification object, a record is added to this effect in the action log in the database and processing is terminated.
An alarm termination request is sent to the external NMS via the Remote Handler TERMINATE_MASTERALARM callout function. The alarm termination request is implicitly targeted at the equivalent alarm report maintained by the external NMS, identified by the previously received 'master' alarm external NMS unique identifier held in the supplied notification object.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied notification object is inserted.

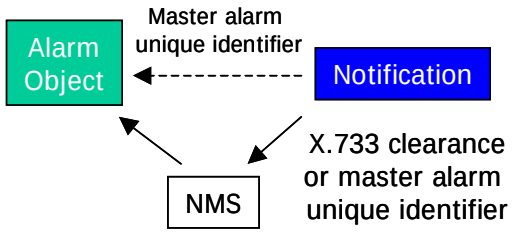
The notification object attached to the targeted (mesh) object.

Optional Additional Text message to be appended to the alarm report in the external NMS e.g. termination reason.

Option to record action execution details in the database.

Clear Alarm

State Mesh Model



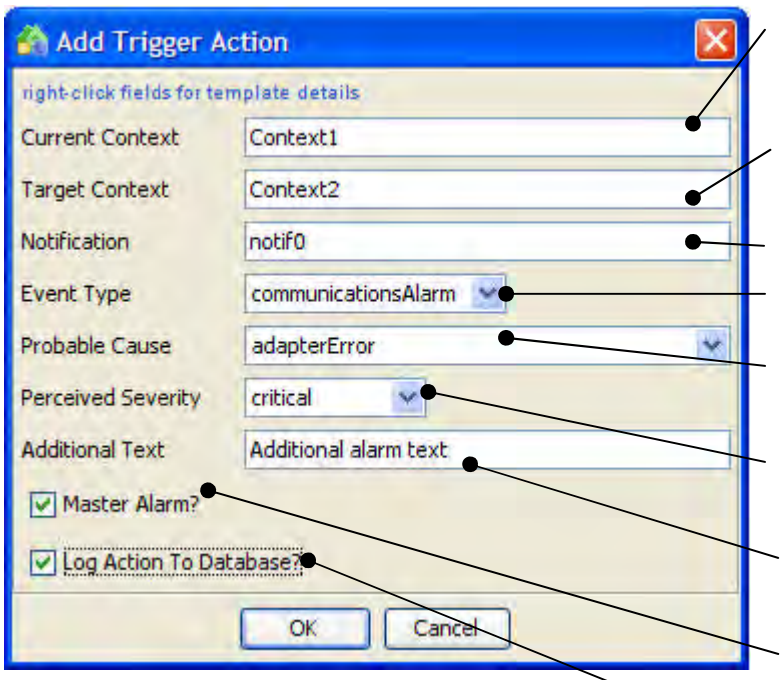
Fired Rules Viewer Mnemonic

trigClearAlarm
tearClearAlarm

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
If the option to create a master alarm report in the NMS is chosen, the master alarm PENDING option is set in the supplied notification object and it is updated in the chosen context(s).
This action sends an alarm clearance request to the external NMS via the Remote Handler CLEAR_ALARM callout function.
Depending on the level of integration with the external NMS and the availability or otherwise of a previously received or generated external NMS 'master' alarm unique identifier in the supplied notification object, the Remote Handler integration must adopt an appropriate technique to clear an existing alarm in the external NMS. This may vary from an X.733-style alarm clearance relying solely on the supplied fields to a closure based on an external NMS 'master' alarm unique identifier. An alarm clearance request may clear alarm reports on (mesh) objects or 'master' alarm reports on notification objects.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied notification object is inserted.

An alternative context in which the supplied notification object may also be inserted (if un-used, set as Current Context).
The notification object attached to the targeted (mesh) object.
X.733 Event Type for the clearance alarm report.
X.733 Probable Cause for the clearance alarm report.

X.733 Perceived Severity for the clearance alarm report.

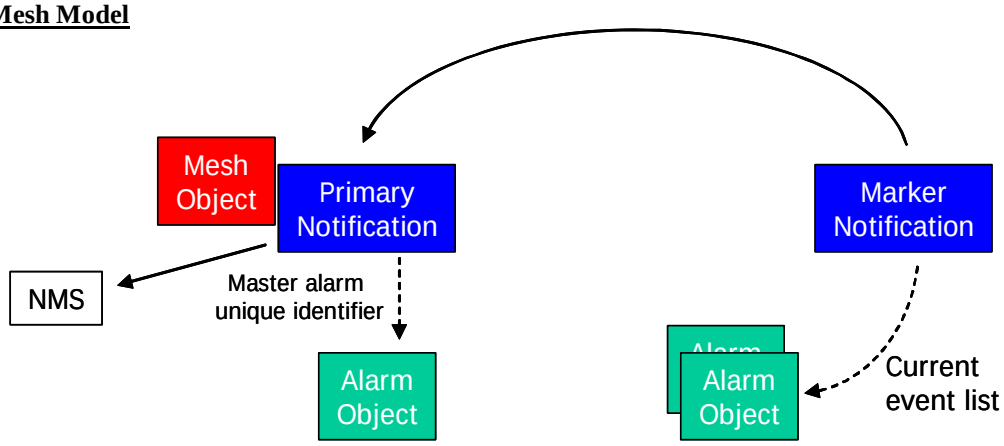
Optional Additional Text message to be inserted into the alarm report in the external NMS e.g. clearance reason.

Option to clear a normal or master alarm report.

Option to record action execution details in the database.

Associate Marker Notification Alarms to Master

State Mesh Model



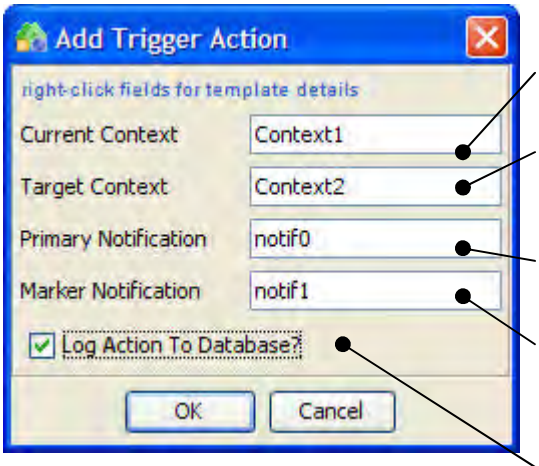
Fired Rules Viewer Mnemonic

trigAssociateMarkerAlarmsToMaster

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
The purpose of this action is to associate one or more alarm reports in the supplied marker notification's current event list with a 'master' alarm report in an extermal NMS.
This action examines the supplied primary notification object for the presence of a 'master' alarm report.
If a 'master' alarm report has never been received, a record is added to this effect in the action log in the database and processing is terminated.
If a 'master' alarm report has been received but is not attached to the supplied notification object, a record is added to this effect in the action log in the database and processing is terminated.
For each alarm object in the supplied marker notification object's current event list, an alarm demotion request (including the 'master' alarm report external NMS reference) is sent to the external NMS via the Remote Handler's DEMOTE_CHILD_ALARMS callout function. The effect in the external NMS depends on the level of integration and its inherent capabilities.
On successful completion of the action, the 'child alarms demoted' attribute in the supplied marker notification object is set to true and this may be evaluated by additional rules.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and where the primary and marker notification objects are inserted.

An alternative context in which the primary and marker notification objects may also be inserted (if un-used, set as Current Context).

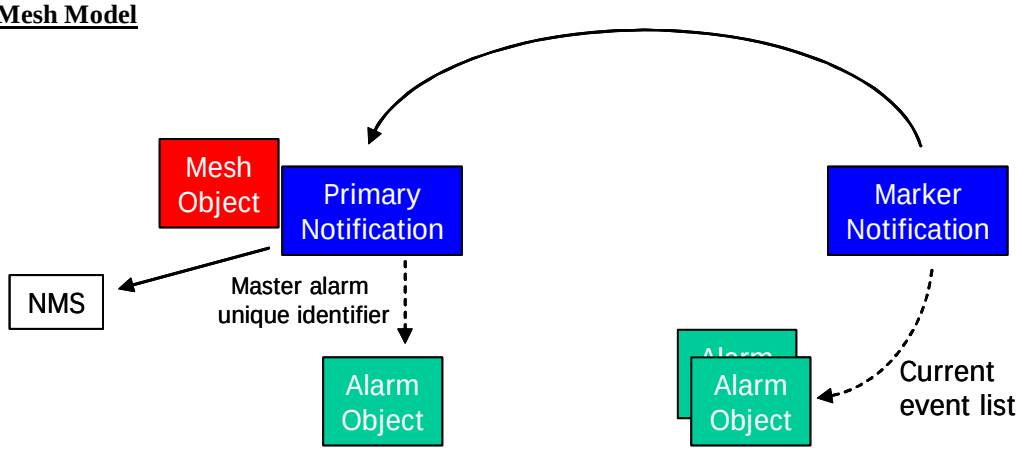
The primary notification containing the external NMS 'master' alarm report reference.

The marker notification object whose current event list contains the set of alarm reports to be associated with the 'master' alarm report.

Option to record action execution details in the database.

Dissociate Marker Notification Alarms From Master

State Mesh Model



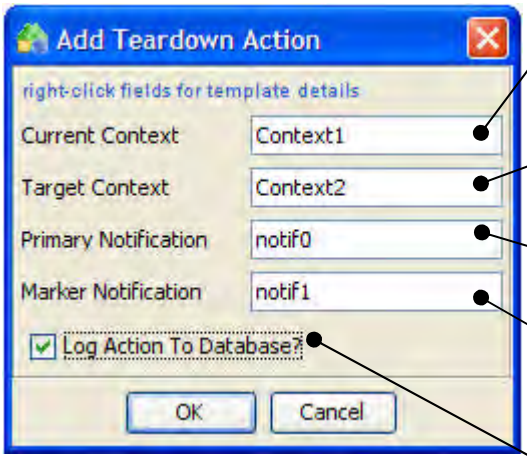
Fired Rules Viewer Mnemonic

tearDissociateMarkerAlarmsFromMaster

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
The purpose of this action is to dissociate one or more alarm reports in the supplied marker notification's current event list from a 'master' alarm report in an extermal NMS.
This action examines the supplied primary notification object for the presence of a 'master' alarm report.
If a 'master' alarm report has never been received, a record is added to this effect in the action log in the database and processing is terminated.
If a 'master' alarm report has been received but is not attached to the supplied notification object, a record is added to this effect in the action log in the database and processing is terminated.
For each alarm object in the supplied marker notification object's current event list, an alarm promotion request (including the 'master' alarm report external NMS reference) is sent to the external NMS via the UCA Remote Handler's PROMOTE_CHILD_ALARMS callout function. The effect in the external NMS depends on the level of integration with UCA and its inherent capabilities.
On successful completion of the action, the 'child alarms demoted' attribute in the supplied marker notification object is set to false and this may be evaluated by additional rules.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and where the primary and marker notification objects are inserted.

An alternative context in which the primary and marker notification objects may also be inserted (if un-used, set as Current Context).

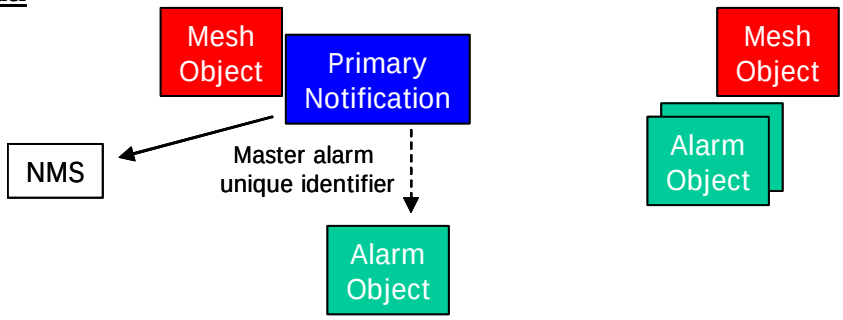
The primary notification containing the external NMS 'master' alarm report reference.

The marker notification object whose current event list contains the set of alarm reports to be promoted from under the 'master' alarm report.

Option to record action execution details in the database.

Associate Object Alarms To Master

State Mesh Model



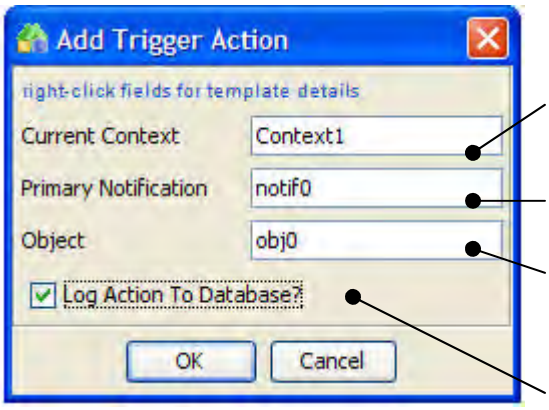
Fired Rules Viewer Mnemonic

trigAssociateObjectAlarmsToMaster

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
The purpose of this action is to associate one or more alarm reports in the supplied (mesh) object's current event list with a 'master' alarm report in an external NMS.
This action examines the supplied primary notification object for the presence of a 'master' alarm report.
If a 'master' alarm report has never been received, a record is added to this effect in the action log in the database and processing is terminated.
If a 'master' alarm report has been received but is not attached to the supplied notification object, a record is added to this effect in the action log in the database and processing is terminated.
For each alarm object in the supplied object's current event list, an alarm demotion request (including the 'master' alarm report external NMS reference) is sent to the external NMS via the Remote Handler's DEMOTE_CHILD_ALARMS callout function. The effect in the external NMS depends on the level of integration and its inherent capabilities.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and where the mesh and primary notification objects are inserted.

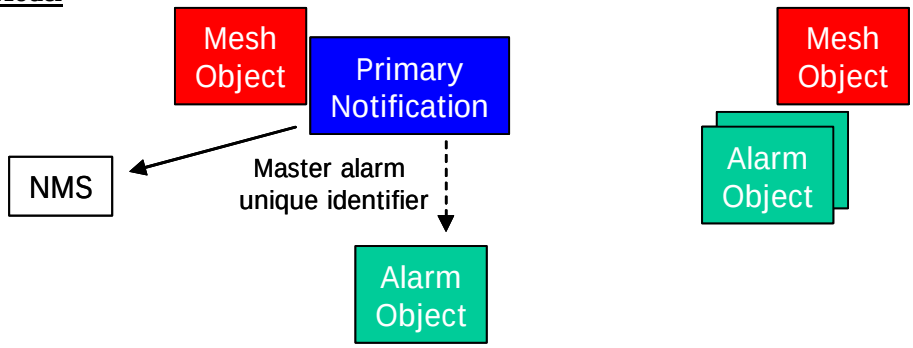
The primary notification object containing the external NMS 'master' alarm report reference.

The (mesh) object whose current event list contains the set of alarm reports to be associated with the 'master' alarm report.

Option to record action execution details in the database.

Dissociate Object Alarms From Master

State Mesh Model



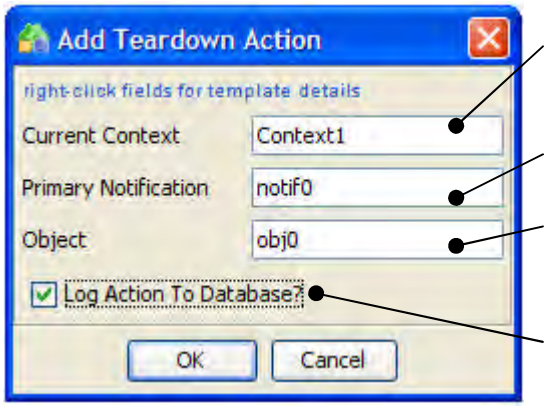
Fired Rules Viewer Mnemonic

tearDissociateObjectAlarmsFromMaster

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
The purpose of this action is to dissociate one or more alarm reports in the supplied (mesh) object's current event list from a 'master' alarm report in an external NMS.
This action examines the supplied primary notification object for the presence of a 'master' alarm report.
If a 'master' alarm report has never been received, a record is added to this effect in the action log in the database and processing is terminated.
If a 'master' alarm report has been received but is not attached to the supplied notification object, a record is added to this effect in the action log in the database and processing is terminated.
For each alarm object in the supplied object's current event list, an alarm promotion request (including the 'master' alarm report external NMS reference) is sent to the external NMS via the UCA Remote Handler's PROMOTE_CHILD_ALARMS callout function. The effect in the external NMS depends on the level of integration with UCA and its inherent capabilities.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and where the mesh and primary notification objects are inserted.

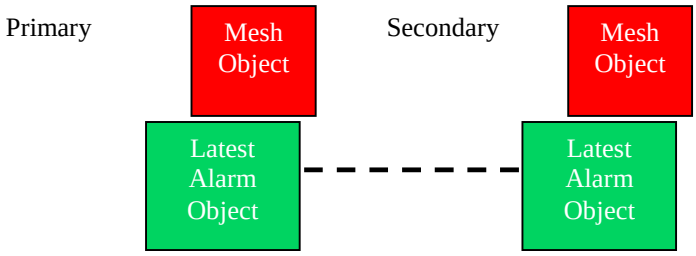
The primary notification object containing the external NMS 'master' alarm report reference.

The mesh object whose current event list contains the set of alarm reports to be promoted from under the 'master' alarm report.

Option to record action execution details in the database.

Associate Alarms

State Mesh Model



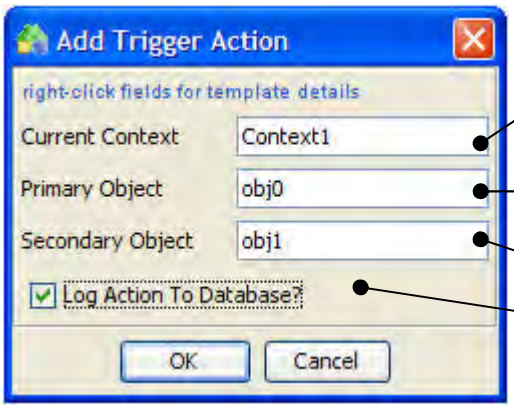
Fired Rules Viewer Mnemonic

trigAssociateAlarms

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
The action attempts to obtain the latest alarm report from each of the supplied (mesh) objects.
If the latest alarm reports are not obtainable from either of the supplied (mesh) objects, a record is added to this effect in the action log in the database and processing is terminated.
An alarm associate request (including both alarm report external NMS references) is sent to the external NMS via the Remote Handler's DEMOTE_CHILD_ALARMS callout function. The association request will attempt to make the latest alarm report from the secondary (mesh) object a child of the latest alarm report from the primary (mesh) object. The effect in the external NMS depends on the level of integration and its inherent capabilities.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and where the primary and secondary (mesh) objects are inserted.

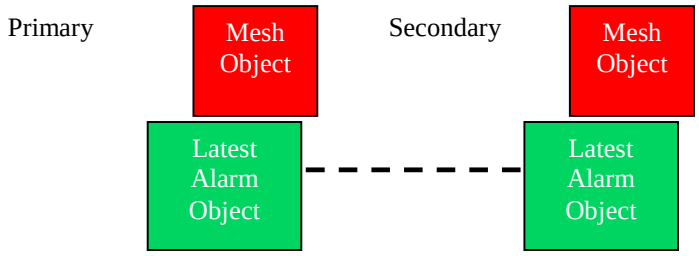
The primary (mesh) object whose current event list contains the parent external NMS alarm reference.

The secondary (mesh) object whose current event list contains the child external NMS alarm reference.

Option to record action execution details in the database.

Dissociate Alarms

State Mesh Model



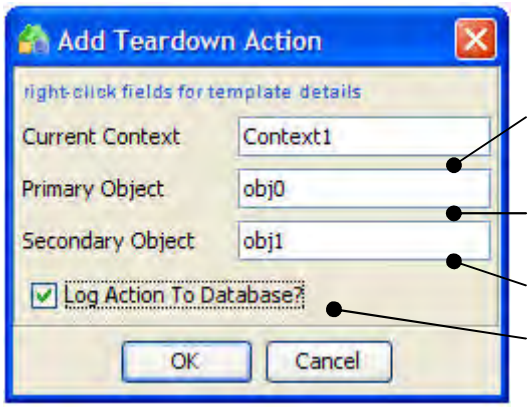
Fired Rules Viewer Mnemonic

tearDissociateAlarms

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
The action attempts to obtain the latest alarm report from each of the supplied (mesh) objects.
If the latest alarm reports are not obtainable from either of the supplied (mesh) objects, a record is added to this effect in the action log in the database and processing is terminated.
An alarm dissociation request (including both alarm report external NMS references) is sent to the external NMS via the Remote Handler's DEMOTE_CHILD_ALARMS callout function. The dissociation request will attempt to remove the latest alarm report from the secondary (mesh) object as a child of the latest alarm report from the primary (mesh) object. The effect in the external NMS depends on the level of integration and its inherent capabilities.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and where the primary and secondary (mesh) objects are inserted.

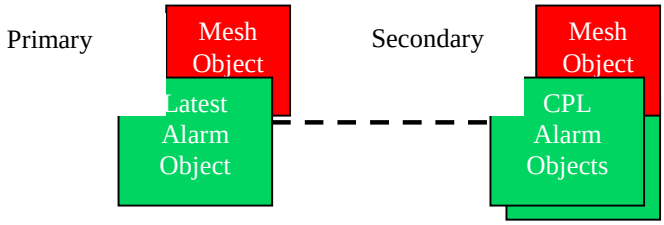
The primary (mesh) object whose current event list contains the parent external NMS alarm reference.

The secondary (mesh) object whose current event list contains the child external NMS alarm reference.

Option to record action execution details in the database.

Associate CPL Alarms

State Mesh Model



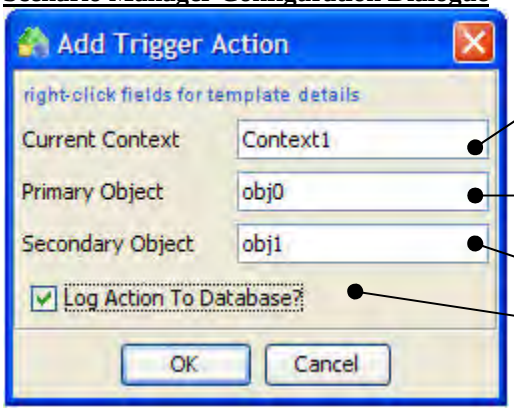
Fired Rules Viewer Mnemonic

trigAssociateCPLAlarms

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
The action attempts to obtain the latest alarm report from the Primary mesh object and the CPL (Contributing Problem List) of the Secondary mesh object.
If the either are not obtainable from the supplied (mesh) objects, a record is added to this effect in the action log in the database and processing is terminated.
An alarm associate request (including both alarm report external NMS references) is sent to the external NMS via the Remote Handler's DEMOTE_CHILD_ALARMS callout function for each alarm in the CPL. The association request will attempt to make the alarm reports from the secondary (mesh) object a child of the latest alarm report from the primary (mesh) object. The effect in the external NMS depends on the level of integration and its inherent capabilities.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and where the primary and secondary (mesh) objects are inserted.

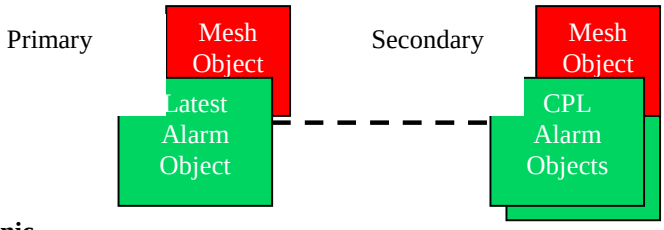
The primary (mesh) object whose current event list contains the parent external NMS alarm reference.

The secondary (mesh) object whose current event list contains the child external NMS alarm reference.

Option to record action execution details in the database.

Dissociate CPL Alarms

State Mesh Model



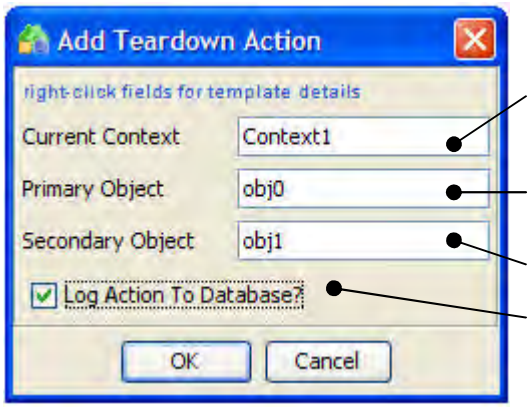
Fired Rules Viewer Mnemonic

tearDissociateCPLAlarms

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
The action attempts to obtain the latest alarm report from the Primary mesh object and the CPL (Contributing Problem List) of the Secondary mesh object.
If the either are not obtainable from the supplied (mesh) objects, a record is added to this effect in the action log in the database and processing is terminated.
An alarm dissociation request (including both alarm report external NMS references) is sent to the external NMS via the Remote Handler's DEMOTE_CHILD_ALARMS callout function. The dissociation request will attempt to remove the alarm reports from the secondary (mesh) object CPL as a child of the latest alarm report from the primary (mesh) object. The effect in the external NMS depends on the level of integration and its inherent capabilities.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and where the primary and secondary (mesh) objects are inserted.

The primary (mesh) object whose current event list contains the parent external NMS alarm reference.
The secondary (mesh) object whose current event list contains the child external NMS alarm reference.
Option to record action execution details in the database.

Forward Last Alarm

State Mesh Model

N/A

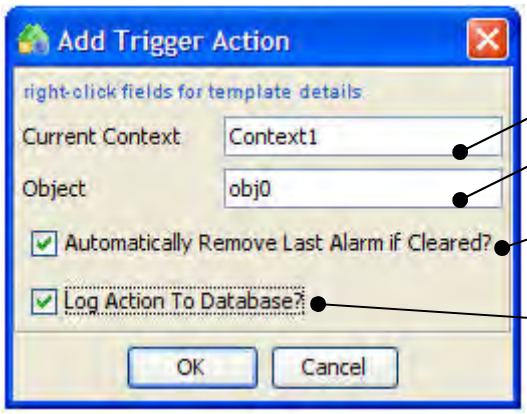
Fired Rules Viewer Mnemonics

trigForwardLastAlarm
tearForwardLastAlarm

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
The action attempts to obtain the latest alarm report the supplied (mesh) object and in turn attempts to retrieve the original alarm report details from the Alarms database.
The alarm report details are sent on to the destination external NMS via the Remote Handler's FORWARD_LAST_ALARM callout function.
If the latest alarm report is of cleared severeity and the option to remove cleared alarms is checked in the action dialogue, the cleared alarm will be automatically removed from the Alarms database.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and where the (mesh) object is inserted.

The (mesh) object whose last alarm report is to be forwarded to the destination external NMS

Option to automatically remove the last alarm report from the Alarm database if its severity is cleared.

Option to record action execution details in the database.

Remove Accumulated Alarms

State Mesh Model

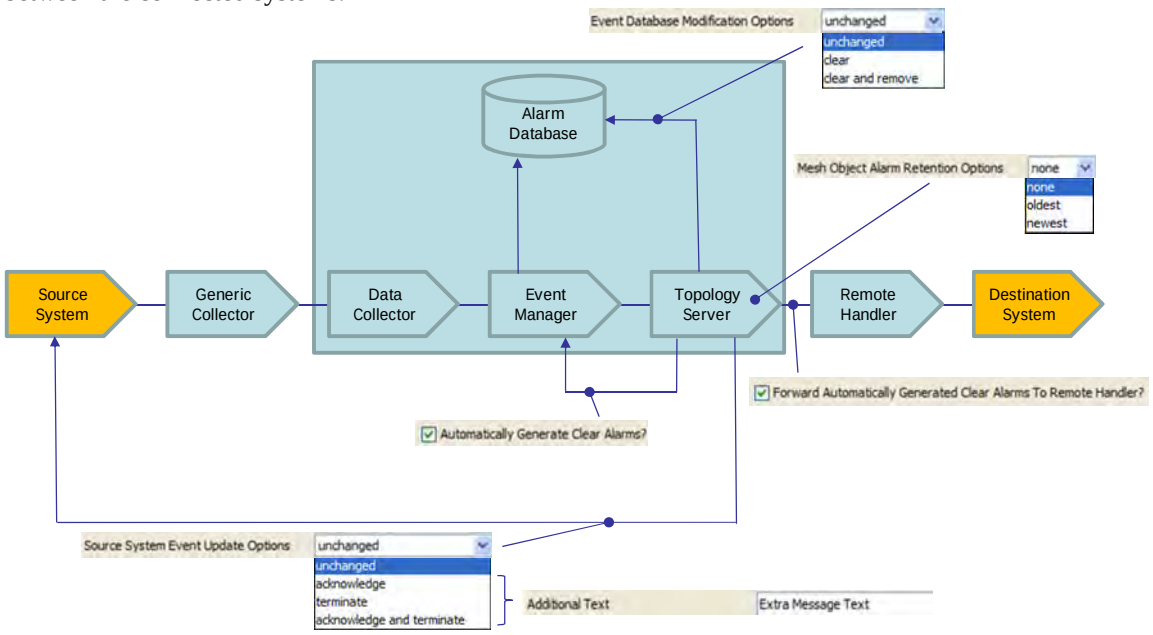
N/A

Fired Rules Viewer Mnemonics

trigRemoveAccumulatedAlarms
tearRemoveAccumulatedAlarms

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to prevent the build-up of accumulated external alarms on a (mesh) object in the situation where alarm clears are never received from the source system. It is normally used when the count of external alarms in the target (mesh) object's Current Problem List has reached or exceeded a threshold value.
The action has been designed to be as flexible as possible and may be used in a number of alternate configurations, depending on the capabilities of the source system and individual user requirements. It may be safely used in combination with the Forward Last Alarm action, provide that action is executed at a higher priority to avoid false triggers. Care should also be taken to ensure that the effect of each option is fully understood and that combinations of options are chosen to avoid conflict.
The following diagram illustrates the effect of the various configuration options on information flows within and between the connected systems:



The action attempts to retrieve the Current Problem List (CPL) from the supplied originating (mesh) object.
If the list is valid (i.e. contains at least on external alarm):

The Mesh Object Alarm Retention option is applied to the CPL to identify the Oldest or Newest alarm if required. If an entry is identified, it is excluded from further processing and will be left unmodified in the (mesh) object's CPL on completion of the action.

Each of the non-excluded alarms in the CPL is subjected to the following processing:

If the alarm's event ID is invalid (e.g. it is a sympathetic alarm), then it is ignored and an exception is reported.

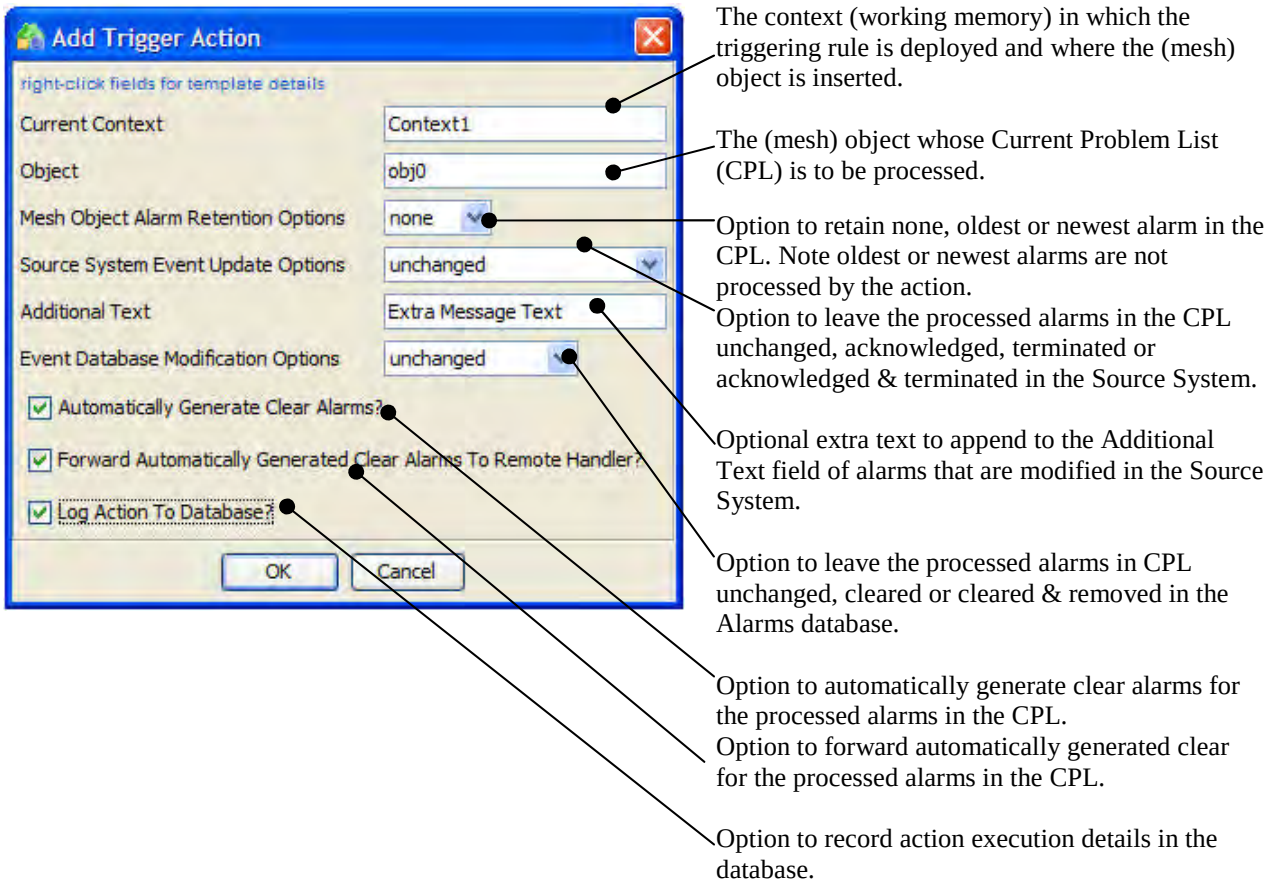
If acknowledgement (or acknowledgement & termination) of the alarm is required in the Source System according to the Source System Event Update option, an acknowledgement callout is delivered to the Remote Handler. It is the responsibility of the integrator to ensure that the appropriate operation is carried out on the Source System in response to the callout.

If termination (or acknowledgement & termination) of the alarm is required in the Source System according to the Source System Event Update option, a termination callout is delivered to the Remote Handler. It is the responsibility of the integrator to ensure that the appropriate operation is carried out on the Source System in response to the callout. Note: termination of an alarm in the Source System would normally be expected to result in an equivalent alarm update message being received by the

system, in turn causing the alarm to be terminated (and therefore closed) within the system.
Whenever the Source System Event Update option is exercised, extra text may be appended to the end of the Additional Text field of the alarm in the Source System using the Additional Text dialogue field.
If clearance of the alarm in the Alarms database is required according to the Event Database Modification option, the relevant entry is updated to close the alarm and the originating time of the clearance is set to be the current time.
If removal of the alarm in the Alarms database is required according to the Event Database Modification option, .the relevant entry is removed.
If internal generation of a clearance alarm is required for the alarm according to the Automatically Generate Clear Alarms checkbox:

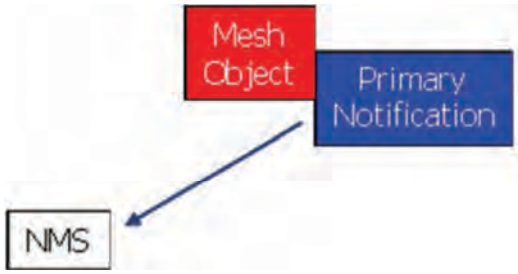
If the subsequent generated alarm clearance is NOT required to be forwarded by the Remote Handler according to the Forward Automatically Generated Clear Alarms To Remote Handler checkbox, the system will prepend "IGNORE:" to any text from the Additional Text dialogue field (the Forward Last Alarm action will subsequently ignore any alarm clearance whose Additional Text field starts with "IGNORE:").
The alarm clearance will be automatically generated and sent internally to the system Event Manager where it will be processed, resulting in the alarm being removed from the CPL of the supplied (mesh) object.

Scenario Manager Configuration Dialogue



Raise Expedited Alarm

State Mesh Model



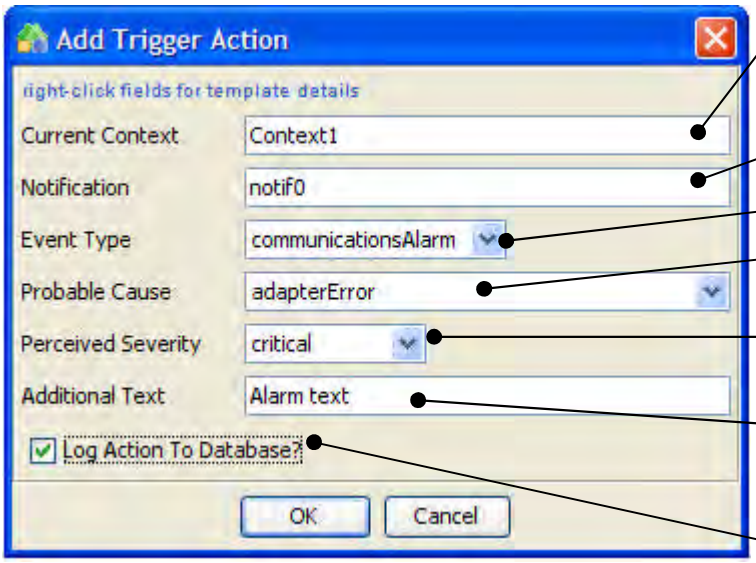
Fired Rules Viewer Mnemonic

trigRaiseExpeditedAlarm
tearRaiseExpeditedAlarm

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action submits an alarm creation request to the external NMS via the Remote Handler RAISE_EXPEDITED_ALARM callout function. The purpose of this callout is to allow a system to report a host platform resource problem e.g. disk space exhaustion, to the external NMS. This implies that an expedited alarm creation request should be carried out even if the system is operating in secondary mode (when its Remote handler outputs are normally turned off, thus preventing the standard alarm creation mechanism from being used for this purpose). The supplied notification is that created by rules in a user-supplied host platform problem detection scenario and may contain additional information relevant to the detected problem.
Depending on the level of integration with the external NMS and the intended use of the new alarm, the unique identifier of the supplied notification object may or may not be passed in the expedited alarm creation request, although it is not intended that this action will create a master alarm as described in the Raise Alarm action.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied notification object is inserted.

The notification object attached to the targeted (mesh) object.

X.733 Event Type for the new alarm report.

X.733 Probable Cause for the new alarm report.

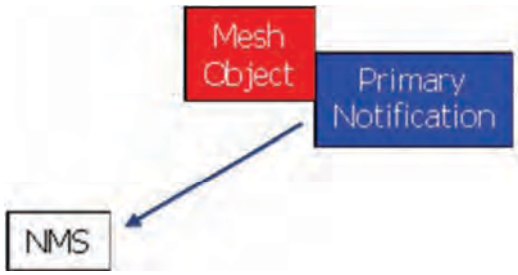
X.733 Perceived Severity for the new alarm report.

Optional Additional Text message to be inserted into the alarm report in the external NMS e.g. creation reason.

Option to record action execution details in the database.

Clear Expedited Alarm

State Mesh Model



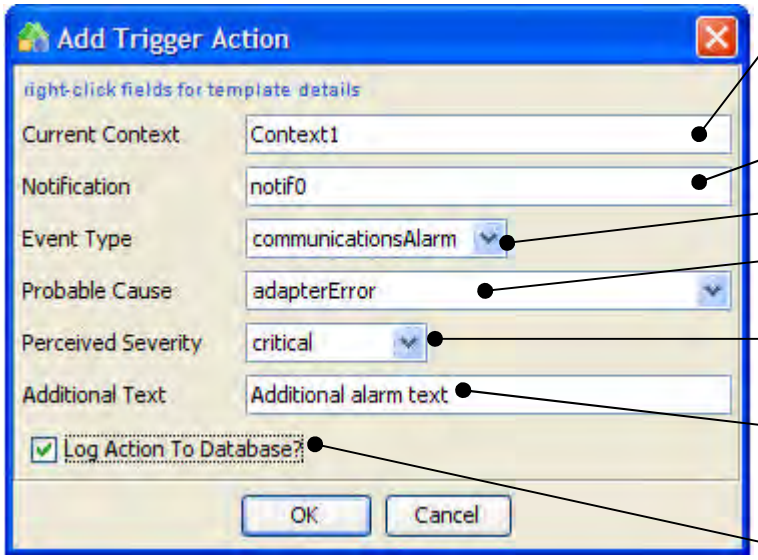
Fired Rules Viewer Mnemonic

trigClearExpeditedAlarm
tearClearExpeditedAlarm

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action submits an alarm clearance request to the external NMS via the Remote Handler CLEAR_EXPEDITED_ALARM callout function. The purpose of this callout is to allow a system to report the resolution of a host platform resource problem e.g. disk space exhaustion, to the external NMS. This implies that an expedited alarm clearance request should be carried out even if the system is operating in secondary mode (when its Remote handler outputs are normally turned off, thus preventing the standard alarm clearance mechanism from being used for this purpose). The supplied notification is that created by rules in a user-supplied host platform problem detection scenario and may contain additional information relevant to resolution of the previously detected problem.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and the supplied notification object is inserted.

The notification object attached to the targeted (mesh) object.

X.733 Event Type for the clearance alarm report.

X.733 Probable Cause for the clearance alarm report.

X.733 Perceived Severity for the clearance alarm report.

Optional Additional Text message to be inserted into the cleared alarm report in the external NMS e.g. clearance reason.

Option to record action execution details in the database.

15.2.2.6 Timer Management

Create Countdown Timer

State Mesh Model

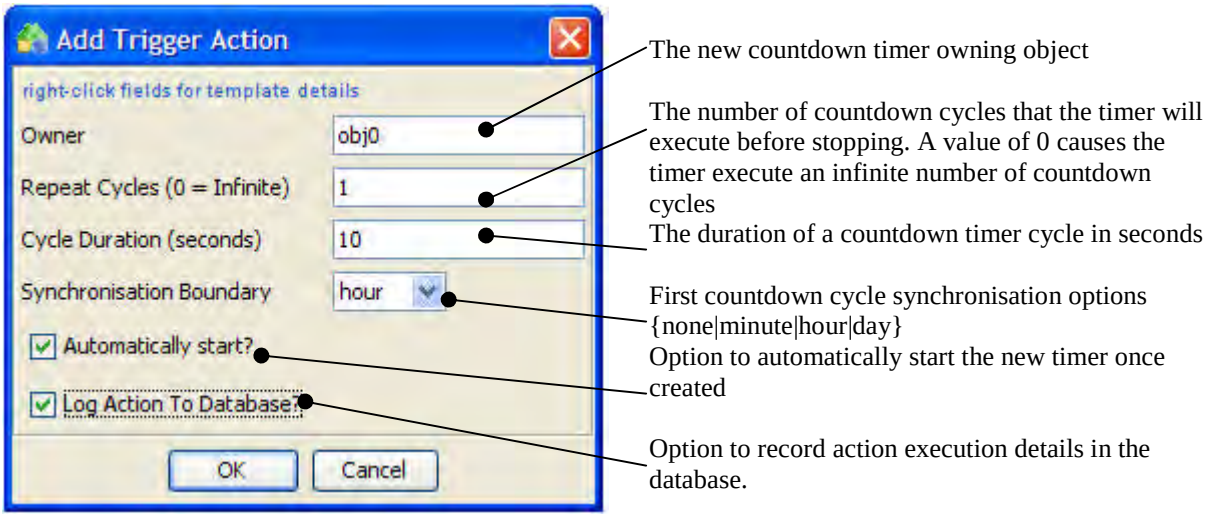
Fired Rules Viewer Mnemonics

trigCreateCountdownTimer
tearCreateCountdownTimer

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to create a new countdown timer object that is associated with the supplied owner object (may be a (mesh) object, a child group object, an associate group object, a notification object, a script object, a data object or a system object (reserved for use by the Resilience package). Only one timer is currently allowed per owner object and the timer resolution is 1 second.
The new timer may be created and optionally not started (state = INITIALISED) or automatically started (state = RUNNING). A running timer may be suspended (state = SUSPENDED), resumed (state = RUNNING) and re-initialised (state = INITIALISED) at any time.
When the countdown timer reaches the end of a cycle, it will inform the owning object that a cycle has completed (state = TIMEOUT).
When all cycles are completed, the timer will cease to operate (state = COMPLETED) unless re-initialised.

Scenario Manager Configuration Dialogue



Start Initialised/Restart Running Countdown Timer

State Mesh Model

N/A

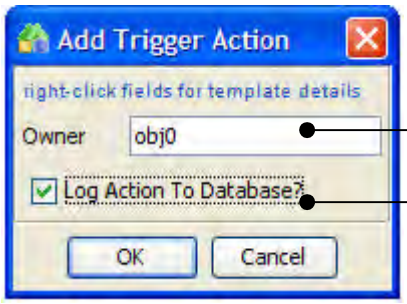
Fired Rules Viewer Mnemonics

trigStartCountdownTimer
tearStartCountdownTimer

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to start an INITIALISED or restart a RUNNING countdown timer object that is associated with the supplied owner object.
In each case, the synchronisation setting for the timer object is taken into account when determining the remaining time to the first or next timeout. Unless the resynchronisation = NONE option was chosen, this will result in the current cycle duration being less than or equal to the cycle duration as the system will synchronise the timer object cycle with the next synchronisation boundary.

Scenario Manager Configuration Dialogue



The countdown timer owning object

Option to record action execution details in the database.

Suspend Running Countdown Timer

State Mesh Model

N/A

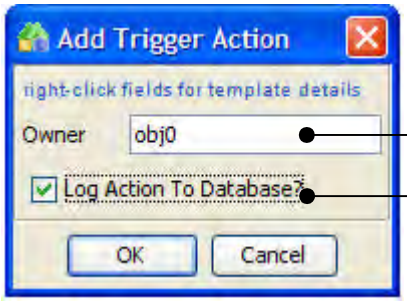
Fired Rules Viewer Mnemonics

trigSuspendCountdownTimer
tearSuspendCountdownTimer

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to suspend a RUNNING countdown timer object that is associated with the supplied owner object. If successful, the timer state is set to SUSPENDED and the countdown is stopped at the current point in the cycle.

Scenario Manager Configuration Dialogue



The countdown timer owning object

Option to record action execution details in the database.

Resume Suspended Countdown Timer

State Mesh Model

N/A

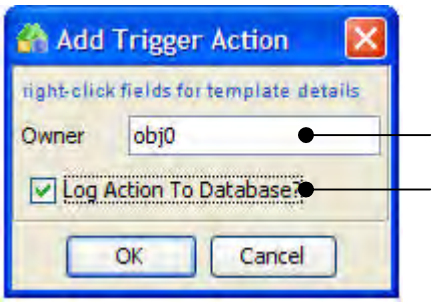
Fired Rules Viewer Mnemonics

trigSuspendCountdownTimer
tearSuspendCountdownTimer

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to resume a SUSPENDED countdown timer object that is associated with the supplied owner object. If successful, the timer state is set to RUNNING and the countdown is resumed at the current point in the cycle.

Scenario Manager Configuration Dialogue



The countdown timer owning object
Option to record action execution details in the database.

Re-Initialise Countdown Timer

State Mesh Model

N/A

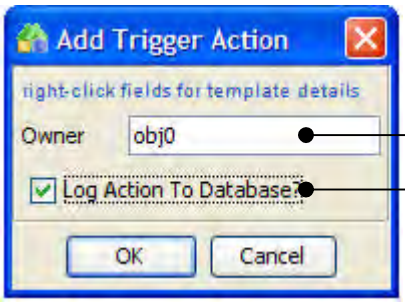
Fired Rules Viewer Mnemonics

trigReinitialiseCountdownTimer
tearReinitialiseCountdownTimer

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
The number of remaining cycles for the countdown timer object is reset to the original number of countdown cycles that were specified when it was first created.
This action attempts to set the timer state to INITIALISED i.e. not currently running.

Scenario Manager Configuration Dialogue



The countdown timer owning object

Option to record action execution details in the database.

Delete Countdown Timer

State Mesh Model

N/A

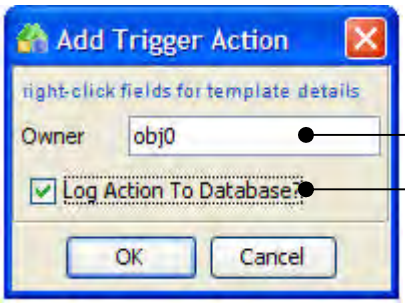
Fired Rules Viewer Mnemonics

trigDeleteCountdownTimer
tearDeleteCountdownTimer

Summary

If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
This action attempts to delete the timer object that is associated with the supplied owning object.

Scenario Manager Configuration Dialogue



The countdown timer owning object

Option to record action execution details in the database.

15.2.2.7 Analysis

Perform Standard Root Cause Analysis

State Mesh Model

Not applicable.

Fired Rules Viewer Mnemonics

trigPerformStandardRootCauseAnalysis
tearPerformStandardRootCauseAnalysis

Summary

This action provides a standard root cause analysis tool whose purpose is to identify and report those problems in a network that are the root cause(s) of a service impact. It operates on a state mesh, normally beginning at a (mesh) object that represents the impacted service. It offers standard root cause analysis with very few options; if greater flexibility is required the Perform Root Cause Analysis provides a much finer degree of control over the analysis.

The following description of the root cause analyser algorithm includes references to the various configuration options (in bold underlined type) at the points at which they affect the flow of processing.

The root cause analyser begins a first stage of discovery processing at the supplied (mesh) **object** that has suffered the service impact. It descends recursively through the state mesh, searching for (mesh) objects whose state has been affected directly or indirectly by underlying network problems. Objects that satisfy the following search criteria are added to a Non-Normal Objects List (NNOL):

- In Service Objects
- Degraded Objects
- Failed Objects

Objects that match the following criteria are excluded from the Non-Normal Objects List (NNOL):

- Commissioning Objects
- Out Of Service Objects
- In Maintenance Objects

The recursive search automatically descends through parent-child relationships and uncle-nephew (relative) relationships below the supplied (mesh) **object** until the lowest level of the state mesh is reached at which point it stops. If the **Follow Associate Links** option is checked, the analysis will traverse an associative relationship between peer (mesh) objects, before continuing down through the state mesh.

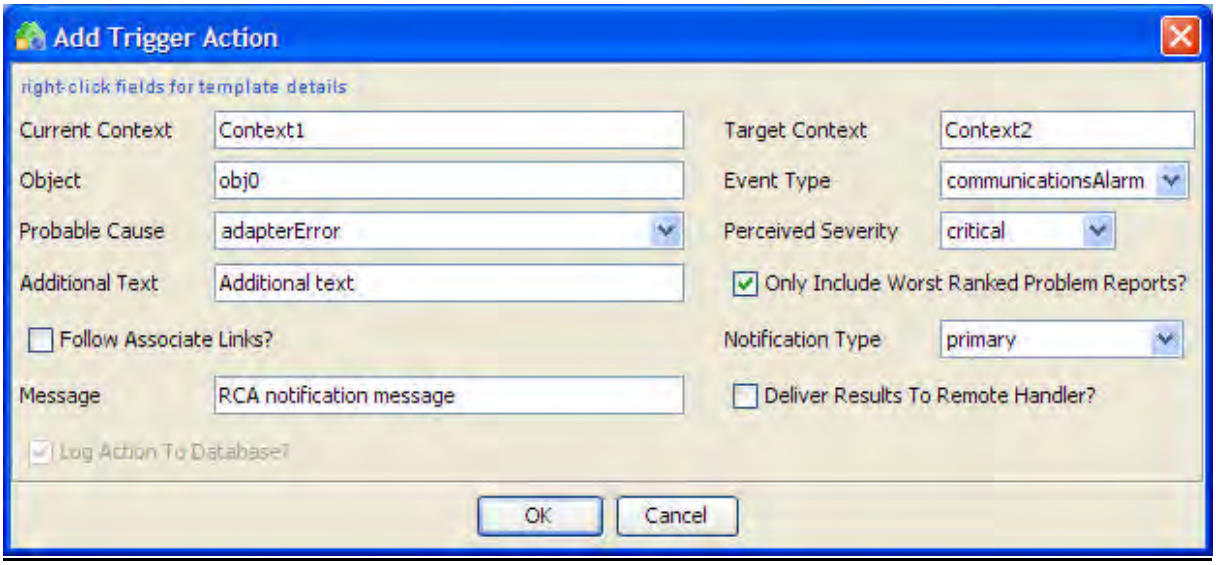
At the end of the search phase, the analyser has identified the set of non-normal(mesh) objects that may have directly or indirectly affected the state of the supplied (mesh) **object.** Once this phase of operation is complete, the root cause analyser begins a second analytical phase of processing:

- The Non-Normal Objects List (NNOL) entries are processed in turn and each NNOL object is compared with the list of current (non-marker) notifications. Where an NNOL object is also found to have an associated notification, the rank of the notification is compared with the worst notification rank seen so far and if it exceeds this value, it becomes the new worst rank.
- The NNOL entries are again processed in turn and each entry is compared with the list of current (non-marker) notifications.
    - If an NNOL object is found to have an associated notification:
        - If the **Only Include Worst Ranked Problem Reports** option is unchecked or the rank of the current notification is equal to the worst rank:
            - The notification is added to the Problem Reports List
            - The events in the notification's contributory events list are added to the Contributory Events List.
            - The associated master alarm external NMS alarm ID (if present) together with the notification ID are added to the Master Alarm List

- If an object in the NNOL does not have an associated notification or no contributory events were added to the Contributory Events List:
    - The NNOL object is added to the Affected Objects List
    - The NNOL object is added to the Markers List.
    - Any events attached to the NNOL object are added to the Sympathetic Events List.

- A root cause notification is built and:
    - A contributory events list is built in the Notification database from the Contributory Events List constructed previously.
    - A new notification is built in the Notification database using the contributory events list in the database.
    - The Affected Objects List constructed previously is added to the new notification in the Notification database.
    - A new notification object is constructed and if the **Deliver Results To Remote Handler** option is checked, the RCA Pending flag in the new notification object is set.
    - If there are entries in the Sympathetic Events List constructed previously, they are added to the sympathetic events list for the new notification in the Notification database.
    - The new notification object is inserted into the Working Memories (Contexts) defined in **Current Context & Target Context**
    - A marker notification object is created (tied to the new notification object created above) for each entry in the Markers List
- If the **Deliver Results To Remote Handler** option is checked, for the String based remote handler:
    - Details of the new notification object (Base Class, Unique Reference, **Message** and Notification ID) are added to Alarm Raise block
    - An entry is added to the Alarm Raise block for each entry in the Problem Reports List (Base Class, Unique Reference, Rank and the external NMS events IDs of each of the contributory events in the notification)
    - An entry is added to the Alarm Raise block for each entry in the Contributory Events List (the external NMS events ID)
    - An entry is added to the Alarm Raise block for each entry in the Sympathetic Events List (the external NMS events ID)
    - An entry is added to the Alarm Raise block for each entry in the Affected Objects List (Base Class, Unique Reference)
    - A Raise Alarm request is passed to the Notification Manager for delivery to all attached Remote Handlers
- If the **Deliver Results To Remote Handler** option is checked, for the XML based remote handler see the Remote Handler XML specification for details.

Scenario Manager Configuration Dialogue



Update Standard Root Cause Analysis

State Mesh Model

Not applicable.

Fired Rules Viewer Mnemonics

trigUpdateStandardRootCauseAnalysis
tearUpdateStandardRootCauseAnalysis

Summary

This action updates the results of a previous root cause analysis. It operates on a state mesh, normally beginning at a (mesh) object that represents the impacted service and updates the notification created by the previous analysis. It offers standard root cause analysis with very few options; if greater flexibility is required the Update Root Cause Analysis provides a much finer degree of control over the analysis. The following description of the root cause analyser algorithm in update mode includes references to the various configuration options (in bold underlined type) at the points at which they affect the flow of processing.

The root cause analyser begins a first stage of discovery processing at the supplied (mesh) **object** that has suffered the service impact. It descends recursively through the state mesh, searching for (mesh) objects whose state has been affected directly or indirectly by underlying network problems. Objects that satisfy the following search criteria (set in the configuration dialogue) are added to a Non-Normal Objects List (NNOL):

- In Service Objects
- Degraded Objects
- Failed Objects

Objects that match the following criteria are excluded from the Non-Normal Objects List (NNOL):

- Commissioning Objects
- Out Of Service Objects
- In Maintenance Objects

The recursive search automatically descends through parent-child relationships and uncle-nephew (relative) relationships below the supplied (mesh) **object** until the lowest level of the state mesh is reached at which point it stops. If the **Follow Associate Links** option is checked, the analysis will traverse an associative relationship between peer (mesh) objects, before continuing down through the state mesh.

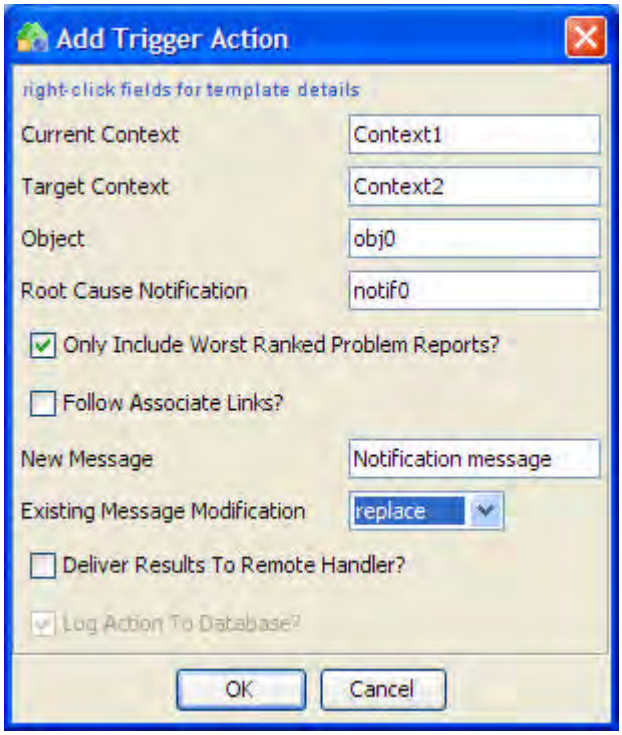
At the end of the search phase, the analyser has identified the set of non-normal (mesh) objects that may have directly or indirectly affected the state of the supplied (mesh) **object.** Once this phase of operation is complete, the root cause analyser begins a second analytical phase of processing:

- The Non-Normal Objects List (NNOL) entries are processed in turn and each NNOL object is compared with the list of current (non-marker) notifications. Where an NNOL object is also found to have an associated notification, the rank of the notification is compared with the worst notification rank seen so far and if it exceeds this value, it becomes the new worst rank.
- The NNOL entries are again processed in turn and each entry is compared with the list of current (non-marker) notifications.
    - If an NNOLobject is found to have an associated notification:
        - If the **Only Include Worst Ranked Problem Reports** option is unchecked or the rank of the current notification is equal to the worst rank :
            - The notification is added to the Problem Reports List
            - The events in the notification's contributory events list are added to the Contributory Events List.
            - The associated master alarm external NMS alarm ID (if present) together with the notification ID are added to the Master Alarm List
    - If an object in the NNOL does not have an associated notification or no contributory events were added to the Contributory Events List:

        - The NNOL object is added to the Affected Objects List
        - The NNOL object is added to the Markers List.
        - Any events attached to the NNOL object are added to the Sympathetic Events List.

- A root cause notification is built and:
    - A contributory events list is built in the Notification database from the Contributory Events List constructed previously.
    - A new notification is built in the Notification database using the contributory events list in the database.
    - The Affected Objects List constructed previously is added to the new notification in the Notification database.
    - A new notification object is constructed and if the **Deliver Results To Remote Handler** option is checked, the RCA Pending flag in the new notification object is set.
    - If there are entries in the Sympathetic Events List constructed previously, they are added to the sympathetic events list for the new notification in the Notification database.
    - The new notification object is inserted into the Working Memories (Contexts) defined in **Current Context & Target Context**
    - A marker notification object is created (tied to the new notification object created above) for each entry in the Markers List
- The Notification message is updated if required in the Working Memory contexts.
- The Root Cause Notification is updated in the database:
    - Any new events in the Contributory Events List constructed previously are added to the contributory events list attached to the existing **notification** in the Notification database
    - Any new entries in the Affected Objects List constructed previously are added to the affected objects list attached to the existing **notification** in the Notification database.
    - The message attached to the existing **notification** in the Notification database is updated is required
- If the **Deliver Results To Remote Handler** option is checked, for the String based remote handler:
    - Details of the new notification object (Base Class, Unique Reference, **Message** and Notification ID) are added to Alarm Raise block
    - An entry is added to the Alarm Raise block for each entry in the Problem Reports List (Base Class, Unique Reference, Rank and the external NMS events IDs of each of the contributory events in the notification)
    - An entry is added to the Alarm Raise block for each entry in the Contributory Events List (the external NMS events ID)
    - An entry is added to the Alarm Raise block for each entry in the Sympathetic Events List (the external NMS events ID)
    - An entry is added to the Alarm Raise block for each entry in the Affected Objects List (Base Class, Unique Reference)
    - A Raise Alarm request is passed to the Notification Manager for delivery to all attached Remote Handlers
- If the **Deliver Results To Remote Handler** option is checked, for the XML based remote handler see the Remote Handler XML specification for details.

Scenario Manager Configuration Dialogue



Perform Root Cause Analysis

State Mesh Model

Not applicable.

Fired Rules Viewer Mnemonics

trigPerformRootCauseAnalysis
tearPerformRootCauseAnalysis

Summary

This action encapsulates a very flexible root cause analysis tool whose purpose is to identify and report those problems in a network that are the root cause(s) of a service impact. It operates on a state mesh, normally beginning at a (mesh) object that represents the impacted service.
The detailed behaviour of the root cause analyser is highly configurable and uses a large number of options supplied by the Scenario Manager configuration dialogue. The following description of the root cause analyser algorithm includes references to the various configuration options (in bold underlined type) at the points at which they affect the flow of processing.
The root cause analyser begins a first stage of discovery processing at the supplied (mesh) **object** that has suffered the service impact. It descends recursively through the state mesh, searching for (mesh) objects whose state has been affected directly or indirectly by underlying network problems. Objects that satisfy the following search criteria (set in the configuration dialogue) are added to a Non-Normal Objects List (NNOL):

- **Include In Service Objects** (default true)
- **Include Commissioning Objects** (default false)
- **Include Out Of Service Objects** (default false)
- **Include In Maintenance Objects** (default false)
- **Include Degraded Objects** (default true)
- **Include Failed Objects** (default true)

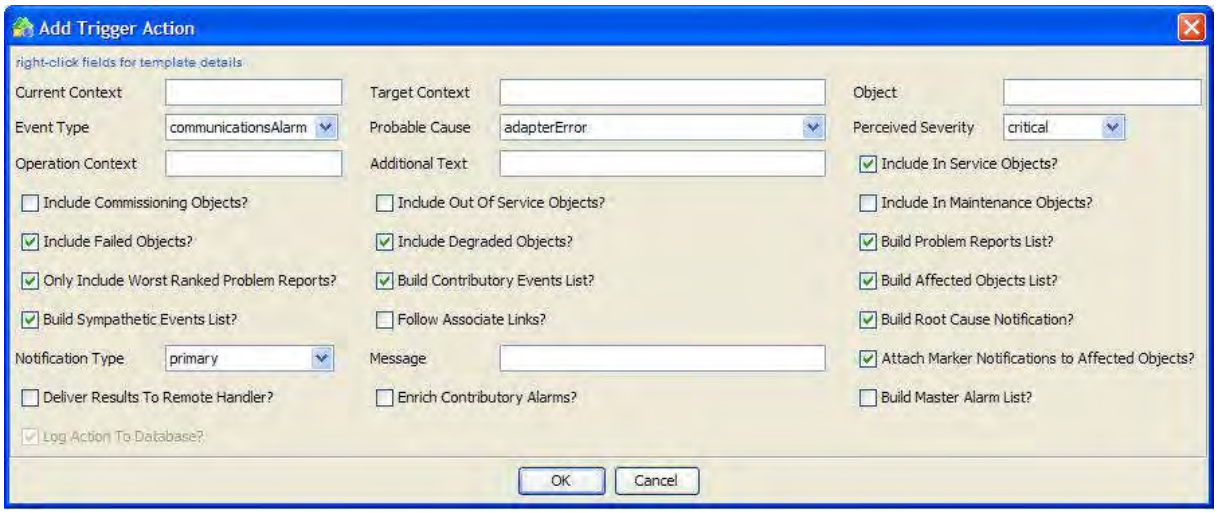
The recursive search automatically descends through parent-child relationships and uncle-nephew (relative) relationships below the supplied (mesh) **object** until the lowest level of the state mesh is reached at which point it stops. If the **Follow Associate Links** option is checked, the analysis will traverse an associative relationship between peer (mesh) objects, before continuing down through the state mesh.
At the end of the search phase, the analyser has identified the set of non-normal(mesh) objects that may have directly or indirectly affected the state of the supplied (mesh) **object.** Once this phase of operation is complete, the root cause analyser begins a second analytical phase of processing:

- The Non-Normal Objects List (NNOL) entries are processed in turn and each NNOL object is compared with the list of current (non-marker) notifications. Where an NNOL object is also found to have an associated notification, the rank of the notification is compared with the worst notification rank seen so far and if it exceeds this value, it becomes the new worst rank.
- The NNOL entries are again processed in turn and each entry is compared with the list of current (non-marker) notifications.
    - If an NNOL object is found to have an associated notification:
        - If the **Only Include Worst Ranked Problem Reports** option is unchecked or the rank of the current notification is equal to the worst rank :
            - If the **Build Problem Reports List** option is checked, the notification is added to the Problem Reports List
            - If the **Build Contributory Events List** option is checked, the events in the notification's contributory events list are added to the Contributory Events List.
            - If the **Build Master Alarms List** option is checked, the associated master alarm external NMS alarm ID (if present) together with the notification ID are added to the Master Alarm List

- If an object in the NNOL does not have an associated notification or no contributory events were added to the Contributory Events List:
    - If the **Build Affected Objects List** option is checked, the NNOL object is added to the Affected Objects List
    - If the **Attach Marker Notifications to Affected Objects** option is checked, the NNOL object is added to the Markers List.
    - If both the **Build Affected Objects List & Build Sympathetic Events List** options are checked, any events attached to the NNOL object are added to the Sympathetic Events List.

- If the **Build Root Cause Notification** option is checked:
    - A contributory events list is built in the Notification database from the Contributory Events List constructed previously.
    - A new notification is built in the Notification database using the contributory events list in the database.
    - If the **Build Affected Objects List** option is checked, the Affected Objects List constructed previously is added to the new notification in the Notification database.
    - A new notification object is constructed and if the **Deliver Results To Remote Handler** option is checked, the RCA Pending flag in the new notification object is set.
    - If the **Build Affected Objects List** option is checked and there are entries in the Sympathetic Events List constructed previously, they are added to the sympathetic events list for the new notification in the Notification database.
    - The new notification object is inserted into the Working Memories (Contexts) defined in **Current Context & Target Context**
    - .If the **Attach Marker Notifications to Affected Objects** option is checked, a marker notification object is created (tied to the new notification object created above) for each entry in the Markers List
- If the **Deliver Results To Remote Handler** option is checked:
    - Details of the new notification object (Base Class, Unique Reference, **Message** and Notification ID) are added to Alarm Raise block
    - An entry is added to the Alarm Raise block for each entry in the Problem Reports List (Base Class, Unique Reference, Rank and the external NMS events IDs of each of the contributory events in the notification)
    - An entry is added to the Alarm Raise block for each entry in the Contributory Events List (the external NMS events ID)
    - An entry is added to the Alarm Raise block for each entry in the Sympathetic Events List (the external NMS events ID)
    - An entry is added to the Alarm Raise block for each entry in the Affected Objects List (Base Class, Unique Reference)
    - A Raise Alarm request is passed to the Notification Manager for delivery to all attached Remote Handlers
- **Enrich Contributory Alarms** option – not yet supported
- If the **Build Master Alarm List** option is checked, all entries in the Master Alarms List are added to the Master Alarms block

Scenario Manager Configuration Dialogue


Add Trigger Action
right-click fields for template details
Current Context
Target Context
Object
Event Type
communicationsAlarm
Probable Cause
adapterError
Perceived Severity
critical
Operation Context
Additional Text
Include In Service Objects?
Include Commissioning Objects?
Include Out Of Service Objects?
Include In Maintenance Objects?
Include Failed Objects?
Include Degraded Objects?
Build Problem Reports List?
Only Include Worst Ranked Problem Reports?
Build Contributory Events List?
Build Affected Objects List?
Build Sympathetic Events List?
Follow Associate Links?
Build Root Cause Notification?
Notification Type
primary
Message
Attach Marker Notifications to Affected Objects?
Deliver Results To Remote Handler?
Enrich Contributory Alarms?
Build Master Alarm List?
Log Action To Database?
OK
Cancel

Update Root Cause Analysis

State Mesh Model

Not applicable.

Fired Rules Viewer Mnemonics

trigUpdateRootCauseAnalysis
tearUpdateRootCauseAnalysis

Summary

This action updates the results of a previous root cause analysis. It operates on a state mesh, normally beginning at a (mesh) object that represents the impacted service and updates the notification created by the previous analysis.

The detailed behaviour of the root-cause analyser is highly configurable and uses a large number of options supplied by a Scenario Manager configuration dialogue. The following description of the root cause analyser algorithm in update mode includes references to the various configuration options (in bold underlined type) at the points at which they affect the flow of processing.

The root cause analyser begins a first stage of discovery processing at the supplied (mesh) **object** that has suffered the service impact. It descends recursively through the state mesh, searching for (mesh) objects whose state has been affected directly or indirectly by underlying network problems. Objects that satisfy the following search criteria (set in the configuration dialogue) are added to a Non-Normal Objects List (NNOL):

- **Include In Service Objects** (default true)
- **Include Commissioning Objects** (default false)
- **Include Out Of Service Objects** (default false)
- **Include In Maintenance Objects** (default false)
- **Include Degraded Objects** (default true)
- **Include Failed Objects** (default true)

The recursive search automatically descends through parent-child relationships and uncle-nephew (relative) relationships below the supplied (mesh) **object** until the lowest level of the state mesh is reached at which point it stops. If the **Follow Associate Links** option is checked, the analysis will traverse an associative relationship between peer (mesh) objects, before continuing down through the state mesh.

At the end of the search phase, the analyser has identified the set of non-normal (mesh) objects that may have directly or indirectly affected the state of the supplied (mesh) **object.** Once this phase of operation is complete, the root cause analyser begins a second analytical phase of processing:

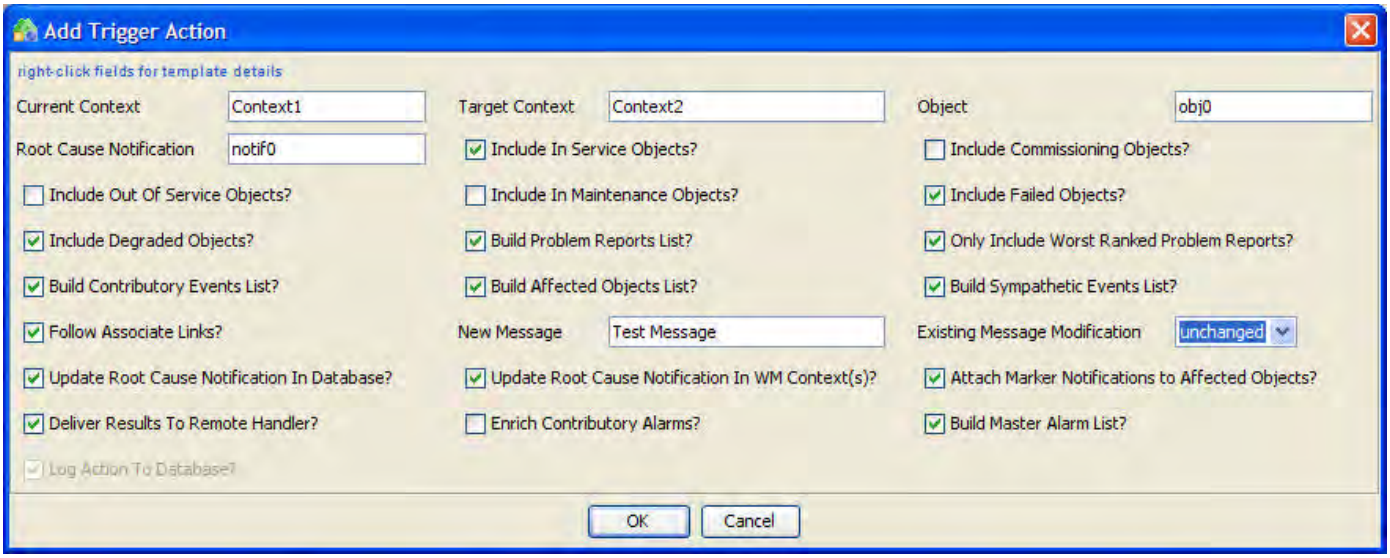
- The Non-Normal Objects List (NNOL) entries are processed in turn and each NNOL object is compared with the list of current (non-marker) notifications. Where an NNOL object is also found to have an associated notification, the rank of the notification is compared with the worst notification rank seen so far and if it exceeds this value, it becomes the new worst rank.
- The NNOL entries are again processed in turn and each entry is compared with the list of current (non-marker) notifications.
    - If an NNOLobject is found to have an associated notification:
        - If the **Only Include Worst Ranked Problem Reports** option is unchecked or the rank of the current notification is equal to the worst rank :
            - If the **Build Problem Reports List** option is checked, the notification is added to the Problem Reports List
            - If the **Build Contributory Events List** option is checked, the events in the notification's contributory events list are added to the Contributory Events List.

- If the **Build Master Alarms List** option is checked, the associated master alarm external NMS alarm ID (if present) together with the notification ID are added to the Master Alarm List

- If an object in the NNOL does not have an associated notification or no contributory events were added to the Contributory Events List:
  - If the **Build Affected Objects List** option is checked, the NNOL object is added to the Affected Objects List
  - If the **Attach Marker Notifications to Affected Objects** option is checked, the NNOL object is added to the Markers List.
  - If both the **Build Affected Objects List & Build Sympathetic Events List** options are checked, any events attached to the NNOL object are added to the Sympathetic Events List.

- If the **Update Root Cause Notification In WM** option is checked:
  - The Notification message is updated if required in the Working Memory contexts.
- If the **Update Root Cause Notification In Database** option is checked:
  - Any new events in the Contributory Events List constructed previously are added to the contributory events list attached to the existing **notification** in the Notification database
  - If the **Build Affected Objects List** option is checked, any new entries in the Affected Objects List constructed previously are added to the affected objects list attached to the existing **notification** in the Notification database.
  - The message attached to the existing **notification** in the Notification database is updated is required
  - .If the **Attach Marker Notifications to Affected Objects** option is checked, new marker notification objects are created (tied to the **notification** object) for each new entry in the Markers List
- If the **Deliver Results To Remote Handler** option is checked:
  - Details of the originating (mesh) **object** (Base Class, Unique Reference**)** and updated **notification** object (new **Message**, Notification ID, Notification Type, Notification Rank and master alarm external NMS event ID – if present) are added to Alarm Update block
  - An entry is added to the Alarm Update block for each entry in the Problem Reports List (Base Class, Unique Reference, Rank and the external NMS events IDs of each of the contributory events in the Notification)
  - An entry is added to the Alarm Update block for each entry in the Contributory Events List (the external NMS events ID)
  - An entry is added to the Alarm Update block for each entry in the Sympathetic Events List (the external NMS events ID)
  - An entry is added to the Alarm Update block for each entry in the Affected Objects List (Base Class, Unique Reference)
  - A Update Alarm request is passed to the Notification Manager for delivery to all attached Remote Handlers
  - **Enrich Contributory Alarms** option – not yet supported
  - If the **Build Master Alarm List** option is checked, all entries in the Master Alarms List are added to the Master Alarms block

Scenario Manager Configuration Dialogue



Perform Problem Extent Analysis

State Mesh Model

Not applicable.

Fired Rules Viewer Mnemonics

trigPerformProblemExtentAnalysis
tearPerformProblemExtentAnalysis

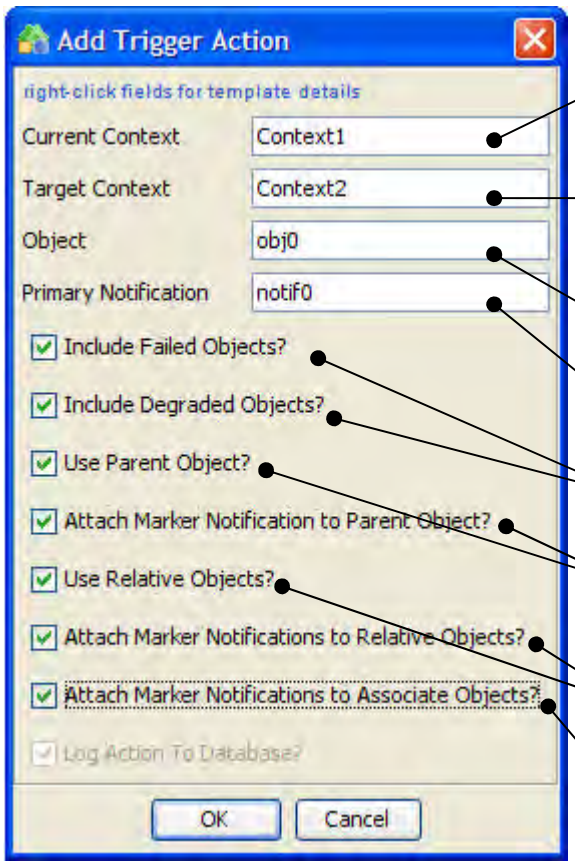
Summary

This action encapsulates a very flexible problem extent analysis tool whose purpose is to identify and report those (mesh) objects that are affected by a problem at a lower level in a layered network. It operates on a state mesh, normally beginning at a (mesh) object that has been previously identified as a problem source (and therefore already has a primary notification attached). It is particularly useful for analysing upwardly divergent network models with the purpose of identifying affected objects and annotating them with marker notifications for the purposes of gathering sympathetic alarms.
The detailed behaviour of the problem extent analyser is highly configurable and uses a number of options supplied by the Scenario Manager configuration dialogue. The following description of the problem extent analyser algorithm includes references to the various configuration options (in bold underlined type) at the points at which they affect the flow of processing.
The problem analyser begins search processing at the supplied problem source (mesh) **object**, on which the supplied primary **notification** also exists. It ascends recursively through the state mesh, beginning with its immediate parent and/or relative (mesh) objects, search for (mesh) objects whose state has been affected directly or indirectly by the originating problem. Objects that satisfy the following search criteria (set in the configuration dialogue) are added to an Affected Objects List (AOL):

- **Include Degraded Objects** (default true)
- **Include Failed Objects** (default true)
- **Use Parent Object** (default true)
- **Attach Marker Notification To Parent Object** (default true)
- **Use Relative Objects** (default true)
- **Attach Marker Notifications To Relative Objects** (default true)
- **Attach Marker Notifications To Associate Objects** (default true)

The search for affected objects starts at the supplied problem source (mesh) **object** and ascends recursively through the parent (if **Use Parent Object** is selected) and/or relative (mesh) objects (if **Use Relative Objects** option is selected).
If the (mesh) object currently being evaluated is not normal (and satisfies the **Include Degraded Objects** or **Include Failed Objects**test criteria) it is added to the AOL.
If a marker notification (linked to the supplied primary **notification**) is required (either from **Attach Marker Notification To Parent Object** or **Attach Marker Notifications To Relative Objects**), it is created and added to the (mesh) object.
If the **Attach Marker Notifications To Associate Objects** option is chosen, then all associate (mesh) objects of the supplied (mesh) **object** are added to the AOL and marker notifications are created and added to them, again linked to the supplied primary **notification.**
If state propagation from the current (mesh) object is enabled to its parent (mesh) object, the recursive analysis continues in this direction until the network extremity is reached.
If state propagation from the current (mesh) object is enabled to its relative (mesh) objects, the recursive analysis continues in this direction until the network extremity is reached.
At the end of the search phase, the analyser has identified those (mesh) objects that have been affected by the original problem and the resulting AOL is added to the supplied primary **notification.**

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and where any new marker notification objects will be inserted.

An alternative context in which any new marker notification objects may also be inserted (if un-used, set as Current Context).

The supplied problem source (mesh) object

The primary notification object attached to the problem source (mesh) object.

Options to include affected objects that are in the failed and degraded states.

Options to include parent (mesh) objects and attach marker notification objects to them.

Options to include relative (mesh) objects and attach marker notification objects to them

Option to include associate (mesh) objects and attach marker notifications to them.

Broadcast Analysis Refresh Request

State Mesh Model

Not applicable.

Fired Rules Viewer Mnemonics

trigBroadcastAnalysisRefreshRequest
tearBroadcastAnalysisRefreshRequest

Summary

This action encapsulates a very flexible tool whose purpose is to identify and deliver an analysis (refresh) request to those (mesh) objects that may be affected by a problem at a lower level in a layered network. It operates on a state mesh, normally beginning at a low level (mesh) object that has been previously identified as a problem source. It is particularly useful for identifying target (service) objects in higher network layers on which an initial or an updated root cause analysis needs to be performed. The detailed behaviour of the broadcast tool is highly configurable and uses a number of options supplied by the Scenario Manager configuration dialogue. The following description of the broadcast algorithm includes references to the various configuration options (in bold underlined type) at the points at which they affect the flow of processing.

The broadcast tool begins search processing at the supplied problem source (mesh) **object**. It ascends recursively through the state mesh, beginning with its immediate parent and/or relative (mesh) objects, and searches for target (mesh) objects that may have been affected directly or indirectly by the originating problem. Objects that satisfy the following search criteria (set in the configuration dialogue) have their Analysis Refresh Required attribute set to true:

- **Target Base Class** (required)
- **Target Sub Class** (optional)
- **Include Degraded Objects** (default true)
- **Include Failed Objects** (default true)
- **Use Parent Object** (default true)
- **Use Relative Objects** (default true)

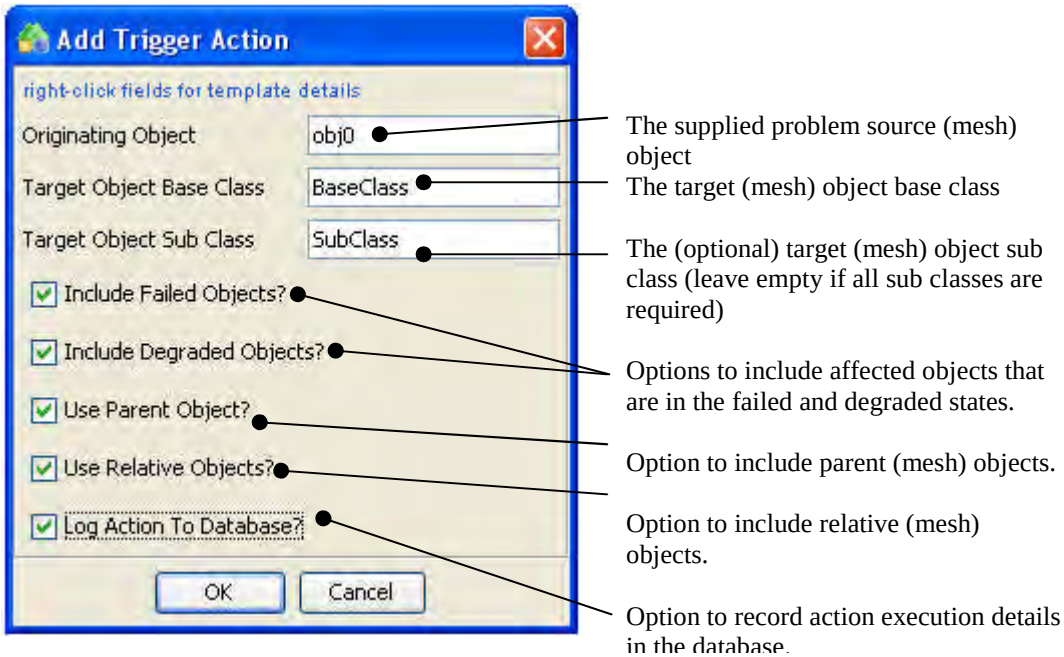
The search for affected objects starts at the supplied problem source (mesh) **object** and ascends recursively through the parent (if **Use Parent Object** is selected) and/or relative (mesh) objects (if **Use Relative Objects** option is selected).

If the (mesh) object currently being evaluated is not normal (and satisfies the **Include Degraded Objects** or **Include Failed Objects** test criteria) and is of the **Target Base Class** and optionally the **Target Sub Class**, then its Analysis Refresh Required attribute is set to true .

If state propagation from the current (mesh) object is enabled to its parent (mesh) object, the recursive analysis continues in this direction until the network extremity is reached.

If state propagation from the current (mesh) object is enabled to its relative (mesh) objects, the recursive analysis continues in this direction until the network extremity is reached.

Scenario Manager Configuration Dialogue



Acknowledge Analysis Refresh Request

State Mesh Model

Not applicable.

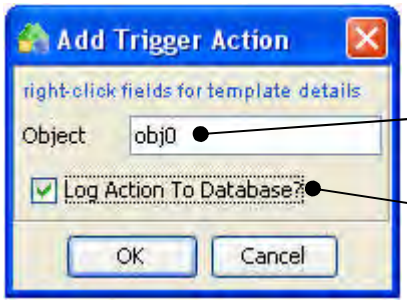
Fired Rules Viewer Mnemonics

trigAcknowledgeAnalysisRefreshRequest
tearAcknowledgeAnalysisRefreshRequest

Summary

This action provides a facility to set the Analysis Refresh Required attribute of a target (mesh) object to false. It is normally used once an initial or updated Root Cause Analysis on the target (mesh) object has been carried out (usually in response to the Analysis Refresh Required attribute having been previously set to true).

Scenario Manager Configuration Dialogue



The (mesh) object whose Analysis Refresh Required attribute requires setting to

Option to record action execution details in the database.

Ticket Handling

The Trouble Ticketing actions are specific to the HP UCA TeMIP Integration document.
Please refere to this documentation for full explanation and examples..

15.2.2.8 Measurement Handling

Create Data Object

State Mesh Model

To Be Completed

Fired Rules Viewer Mnemonic

trigCreateDataObject

Summary

To Be Completed

Scenario Manager Configuration Dialogue

To Be Completed

Refresh Data Object Raw Data

State Mesh Model

To Be Completed

Fired Rules Viewer Mnemonic

trigRefreshDataObject

Summary

To Be Completed

Scenario Manager Configuration Dialogue

To Be Completed

Perform Derived Data Calculation On Data Object

State Mesh Model

To Be Completed

Fired Rules Viewer Mnemonic

trigPerformCalculation

Summary

To Be Completed

Scenario Manager Configuration Dialogue

To Be Completed

Report Derived Data Calculation On Data Object Completed

State Mesh Model

To Be Completed

Fired Rules Viewer Mnemonic

trigReportCalculationFinished

Summary

To Be Completed

Scenario Manager Configuration Dialogue

To Be Completed

Remove Data Object

State Mesh Model

To Be Completed

Fired Rules Viewer Mnemonic

tearRemoveDataObject

Summary

To Be Completed

Scenario Manager Configuration Dialogue

To Be Completed

15.2.2.9 Statistics

Refresh Statistics Object Raw Data

State Mesh Model

To Be Completed

Fired Rules Viewer Mnemonic

trigStatisticsRefresh

Summary

To Be Completed

Scenario Manager Configuration Dialogue

To Be Completed

Perform Derived Data Calculation On Statistics Object

State Mesh Model

To Be Completed

Fired Rules Viewer Mnemonic

trigStatisticsPerformCalculation

Summary

To Be Completed

Scenario Manager Configuration Dialogue

To Be Completed

Report Derived Data Calculation On Statistics Object Completed

State Mesh Model

To Be Completed

Fired Rules Viewer Mnemonic

trigStatisticsCalculationsFinished

Summary

To Be Completed

Scenario Manager Configuration Dialogue

To Be Completed

15.2.2.10 User Defined

Notify Objects Affected By Site Failure

State Mesh Model

Not applicable.

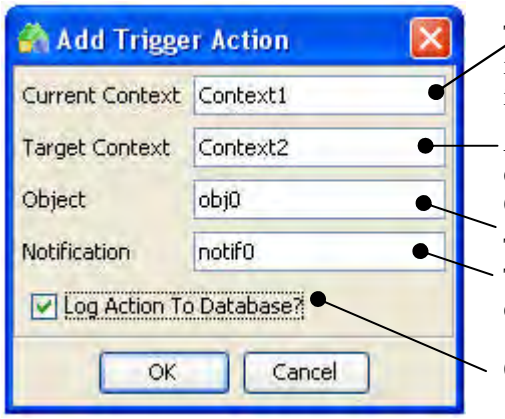
Fired Rules Viewer Mnemonic

trigNotObjSiteFailure

Summary

This action is an example of a user action and is used in the DTV example supplied with UCA.
If loop detection is active, the requested action is tested and if a loop is detected the action is aborted.
The action performs a recursive search, starting from the supplied failed Site (mesh) object, identifying potentially impacted DualReceiver, Receiver and child Site objects. Each located object is added to a list of impacted objects and a marker notification object is attached (with the 'originating' object reference set to the original failed Site).
The action recursively repeats the search for each child Site object located, thus it is able to follow chains of Sites.
When the search is completed, the impacted objects list is added to the failed notification record on the failed Site in the database (identified using the supplied marker notification), causing them to be displayed on the Notification Viewer GUI under the original Site failure notification report.

Scenario Manager Configuration Dialogue



The context (working memory) in which the triggering rule is deployed and where the mesh and notification objects are inserted.

An alternative context in which the mesh and notification objects may also be inserted (if un-used, set as Current Context).

The failed Site mesh object.

The notification object attached to the failed Site mesh object..

Option to record action execution details in the database.