# Netcool®/Precision for IP Networks™

3.6

## Monitoring and RCA Guide

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Part 3: Cambridge Broadband Ltd. copyright notice (BSD) Portions of this code are copyright (c) 2001-2003, Cambridge Broadband Ltd. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- The name of Cambridge Broadband Ltd. may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Part 4: Sun Microsystems, Inc. copyright notice (BSD) Copyright © 2003 Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, USA. All rights reserved. Use is subject to license terms below. This distribution may include materials developed by third parties. Sun, Sun Microsystems, the Sun logo and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of the Sun Microsystems, Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF DVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Part 5: Sparta, Inc. copyright notice (BSD) Copyright (c) 2003-2004, Sparta, Inc. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of Sparta, Inc. nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Part 6: Cisco/BUPTNIC copyright notice (BSD) Copyright (c) 2004, Cisco, Inc. and Information Network, Center of Beijing University of Posts and Telecommunications. All rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

- Neither the name of Cisco, Inc., Beijing University of Posts and Telecommunications, nor the names of their contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF

Micromuse Inc., 650 Townsend Street, San Francisco, USA CA 94103

www.micromuse.com

Document Version Number: 1.0

# Contents

# Preface

This guide describes how to administer, and use the monitoring and root cause analysis components of Netcool/Precision IP. The following chapters describe the functional areas and related concepts.

This preface contains the following sections:

# Audience

This guide is intended for both users and administrators, and provides detailed information about tools, functions, and capabilities. In addition, it is designed to be used as a reference guide to assist you in designing and configuring your environment. Much of the information contained in this document is also provided on-line within the help system.

Netcool/Precision IP works in conjunction with Netcool®/OMNIbus™ and it is assumed that you understand how Netcool/OMNIbus works. For more information on Netcool/OMNIbus, refer to the publications described in *Associated Publications* on page 4.

# About this Guide

This book is organized as follows:

- Chapter 1: *Overview of Monitoring and Root Cause Analysis* on page 11 describes Netcool/Precision IP network monitoring, the process of sending events to the Netcool/OMNIbus ObjectServer, and how Netcool/Precision IP performs topology-based Route Cause Analysis (RCA) on events held in the Netcool/OMNIbus ObjectServer.
- Chapter 2: *Network Polling* on page 21 describes how to use MONITOR and the polling agents to poll your network. It describes how to start MONITOR and the polling agents, and describes how to suspend polling.
- Chapter 3: *MONITOR Configuration Tool* on page 51 describes how to create, edit and browse the Netcool/Precision IP active object classes (AOCs) using the MONITOR Configuration tool. The AOC files include the polling definitions used by MONITOR.
- Chapter 4: *Stitchers Used for Polling* on page 89 describes the stitcher rules unique to the polling agents, complete with an explanation of the required input and output for each.
- Chapter 5: *The MONITOR Probe and Netcool/OMNIbus Probes* on page 127 describes the functionality of the MONITOR probe, its role in the monitoring process, and how to start and configure it.
- Chapter 6: *The Event Gateway* on page 135 describes how to start and configure the Netcool/Precision IP Event Gateway. It also includes descriptions of the gateway databases and descriptions of the event correlation rules used in the RCA-specific AOC extensions.
- Chapter 7: *Root Cause Analysis* on page 155 describes AMOS, the root cause analysis component of Netcool/Precision IP. It also describes the AMOS databases and the event correlation rules in the AOC extensions.

# Associated Publications

Netcool/Precision IP integrates with the Netcool/OMNIbus event management product. Netcool/Precision IP is also deployed within the Netcool GUI Foundation server application, which runs Netcool/Precision IP and other Netcool suite GUIs.

To efficiently administer Netcool/Precision IP, you must possess an understanding of the Netcool/OMNIbus technology. This section provides a description of the documentation that accompanies Netcool/OMNIbus, Netcool/Precision IP and the Netcool GUI Foundation.

## Netcool®/OMNIbus™ Installation and Deployment Guide

This book is intended for Netcool administrators who need to install and deploy Netcool/OMNIbus. It includes installation, upgrade, and licensing procedures. In addition, it contains information about configuring security and component communications. It also includes examples of Netcool/OMNIbus architectures and how to implement them.

## Netcool®/OMNIbus™ User Guide

This book is intended for anyone who needs to use Netcool/OMNIbus desktop tools on UNIX or Windows platforms. It provides an overview of Netcool/OMNIbus components, as well as a description of the operator tasks related to event management using the desktop tools.

## Netcool®/OMNIbus™ Administration Guide

This book is intended for system administrators who need to manage Netcool/OMNIbus. It describes how to perform administrative tasks using the Netcool/OMNIbus Administrator GUI, command line tools, and process control. It also contains descriptions and examples of ObjectServer SQL syntax and automations.

## Netcool®/OMNIbus™ Probe and Gateway Guide

This book contains introductory and reference information about probes and gateways, including probe rules file syntax and gateway commands. For more information about specific probes and gateways, refer to the documentation available for each probe and gateway on the Micromuse Support Site.

## Netcool®/Precision IP™ Installation and Deployment Guide

This book describes the automated installation process and minimum system requirements for Netcool/Precision IP. This book also describes post-installation configuration and troubleshooting.

## Netcool®/Precision IP™ Discovery Configuration Guide

This book describes how to configure and run discoveries. It contains reference information about the Precision Server, which performs network discovery. The book describes the components that make up the Precision Server, including helpers, agents, stitchers, and databases, and includes a detailed command line option reference. In addition, this book provides comprehensive information about the stitcher and OQL languages used within Netcool/Precision IP.

## Netcool®/Precision IP™ Monitoring and RCA Guide

This book describes how to customize monitoring and event correlation, and how to write and adapt monitoring stitchers. The book also describes the RCA Engine databases, and the additional components installed as part of the integration with Netcool/OMNIbus.

## Netcool®/Precision IP™ Desktop Guide

This book describes the operation of the Precision Desktop. The Precision Desktop is not available on Windows.

## Netcool®/Precision IP™ Topology Visualization Guide

This book describes how to visualize your topology using the Topoviz Hop View. The book also describes how to partition your view of the network using the Network Views, and how to view device information using the MIB Browser.

## Netcool GUI Foundation™ Administration Guide

This book describes how to administer the Netcool GUI Foundation, the central server application that runs web-based GUIs from different Netcool products. This guide describes how to configure the Netcool GUI Foundation server, manage users, create and provision pages, and administer security permissions.

## Netcool Licensing™ Administration Guide

This book is intended for Netcool administrators who need to install and administer Netcool Licensing. It provides an overview of the generic Netcool Licensing component, as well as instructions for installing, upgrading, and configuring one or more license servers to dispense licenses to Netcool Licensing clients.

## Online Help

Netcool/Precision IP web-based GUIs contain context-sensitive online help with index and search capabilities. Online documentation, HTML versions of the associated guides, are also available.

# Typographical Notation

Table 1 shows the typographical notation and conventions used to describe commands, SQL syntax, and graphical user interface (GUI) features. This notation is used throughout this book and other Netcool® publications.

Table 1: Typographical Notation and Conventions (1 of 2)

| Example | Description |
|---|---|
| `Monospace` | The following are described in a monospace font:<br>• Commands and command line options<br>• Screen representations<br>• Source code<br>• Object names<br>• Program names<br>• SQL syntax elements<br>• Filenames, paths, and directory names<br>Italicized monospace text indicates a variable that the user must populate. For example, `-password` *`password`*. |
| **Bold** | The following application characteristics are described in a bold font style:<br>• Buttons<br>**Note:** Text in the pop-up tooltips is used to name buttons with icons. These button names are described in plain text.<br>• Frames<br>• Text fields<br>• Menu entries<br>A bold arrow symbol indicates a menu entry selection. For example, **File→Save**. |
| *Italic* | The following are described in an italic font style:<br>• An application window name; for example, the *Login* window<br>• Information that the user must enter<br>• The introduction of a new term or definition<br>• Emphasized text<br>• References to external documents |
| [1] | Code or command examples are occasionally prefixed with a line number in square brackets. For example:<br>[1] `First command...`<br>[2] `Second command...`<br>[3] `Third command...` |

Table 1: Typographical Notation and Conventions (2 of 2)

<table>
<tr><th>Example</th><th>Description</th></tr>
<tr><td>{ a | b }</td><td>In SQL syntax notation, curly brackets enclose two or more required alternative choices, separated by vertical bars.</td></tr>
<tr><td>[ ]</td><td>In SQL syntax notation, square brackets indicate an optional element or clause. Multiple elements or clauses are separated by vertical bars.</td></tr>
<tr><td>|</td><td>In SQL syntax notation, vertical bars separate two or more alternative syntax elements.</td></tr>
<tr><td>...</td><td>In SQL syntax notation, ellipses indicate that the preceding element can be repeated. The repetition is unlimited unless otherwise indicated.</td></tr>
<tr><td>,...</td><td>In SQL syntax notation, ellipses preceded by a comma indicate that the preceding element can be repeated, with each repeated element separated from the last by a comma. The repetition is unlimited unless otherwise indicated.</td></tr>
<tr><td><u>a</u></td><td>In SQL syntax notation, an underlined element indicates a default option.</td></tr>
<tr><td>( )</td><td>In SQL syntax notation, parentheses appearing within the statement syntax are part of the syntax and should be typed as shown unless otherwise indicated.</td></tr>
</table>

Many Netcool commands have one or more command line options that can be specified following a hyphen (`-`).

Command line options can be `string`, `integer`, or `BOOLEAN` types:

- A `string` can contain alphanumeric characters. If the string has spaces in it, enclose it in quotation (`"`) marks.
- An `integer` must contain a positive whole number or zero (`0`).
- A `BOOLEAN` must be set to `TRUE` or `FALSE`.

SQL keywords are not case-sensitive, and may appear in uppercase, lowercase, or mixed case. Names of ObjectServer objects and identifiers are case-sensitive.

## Note, Tip, and Warning Information

The following types of information boxes are used in the documentation:

**Note:** Note is used for extra information about the feature or operation that is being described. Essentially, this is for extra data that is important but not vital to the user.

**Tip:** Tip is used for additional information that might be useful for the user. For example, when describing an installation process, there might be a shortcut that could be used instead of following the standard installation instructions.

**Warning:** Warning is used for highlighting vital instructions, cautions, or critical information. Pay close attention to warnings, as they contain information that is vital to the successful use of our products.

## Syntax and Example Subheadings

The following types of constrained subheading are used in the documentation:

### Syntax

Syntax subheadings contain examples of ObjectServer SQL syntax commands and their usage. For example:

```
CREATE DATABASE database_name;
```

### Example

Example subheadings describe typical or generic scenarios, or samples of code. For example:

```
<body>
   <img src="ChartView?template=barchart&format=PNG
    &request=image&chart=quote&width=800&height=400" border="0" height="400"
    width="800" alt="Events by Severity"
   >
</body>
```

# Operating System Considerations

Unless otherwise specified, command files are located in the `NCHOME/precision/bin` directory, where `NCHOME` is the environment variable that contains the path to the Netcool Suite home directory.

The precise formulation of this path depends on your platform:

- On UNIX platforms, replace `NCHOME` with `$NCHOME`. All command line formats and examples are for the standard UNIX shell. UNIX is case-sensitive. You must type commands in the case shown in the book.
- On Microsoft Windows platforms, replace `NCHOME` with `%NCHOME%` and the forward slash (`/`) with a backward slash (`\`).

# Chapter 1: Overview of Monitoring and Root Cause Analysis

This chapter describes Netcool/Precision IP network monitoring, the process of sending events to the Netcool/OMNIbus ObjectServer, and how Netcool/Precision IP performs topology-based Root Cause Analysis (RCA) on events held in the Netcool/OMNIbus ObjectServer.

This chapter contains the following sections:

# 1.1 Introduction

Having discovered a network topology, Netcool/Precision IP can use it to monitor the status of the devices in it, and determine which of them are experiencing problems. The Netcool/Precision IP component that stores network topology is MODEL.

The Netcool/Precision IP component that controls network polling is MONITOR. MONITOR uses polling agents to gather data. This data is passed as alerts to the ObjectServer through the MONITOR probe. The Netcool/OMNIbus ObjectServer is the master repository for all event information across the Netcool suite. Alerts from many other sources are stored in the ObjectServer.

Events are passed to the Netcool/Precision IP gateway. The gateway enriches some types of event with information held in MODEL and updates the events in the ObjectServer. Some types of event are passed through to AMOS.

The Netcool/Precision IP component that performs Root Cause Analysis (RCA) on events is AMOS. It calculates the elements which are root cause events and suppresses events which are downstream (symptoms) of this event. Figure 1 shows a high level view of the interactions between Netcool/OMNIbus and the polling and RCA components of Netcool/Precision IP.

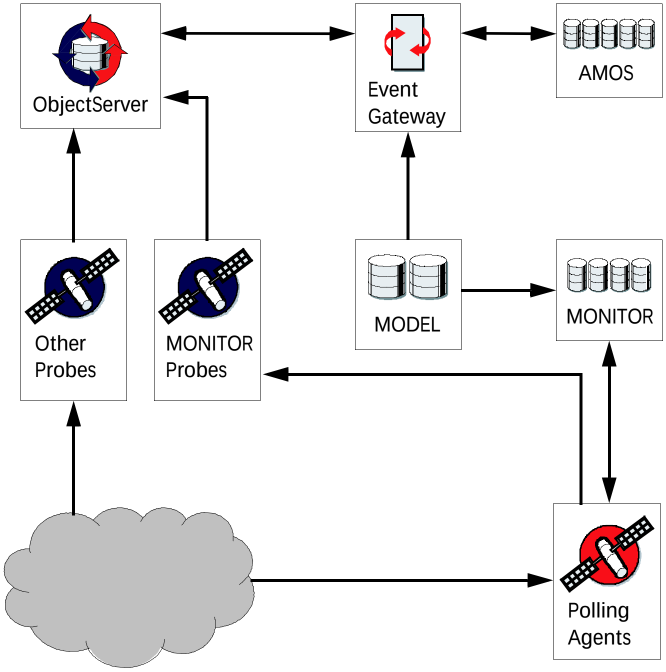


Figure 1: Overview of Netcool/Precision IP and Netcool/OMNIbus Integration

## 1.2 The Polling Process

The polling process is illustrated in Figure 2.

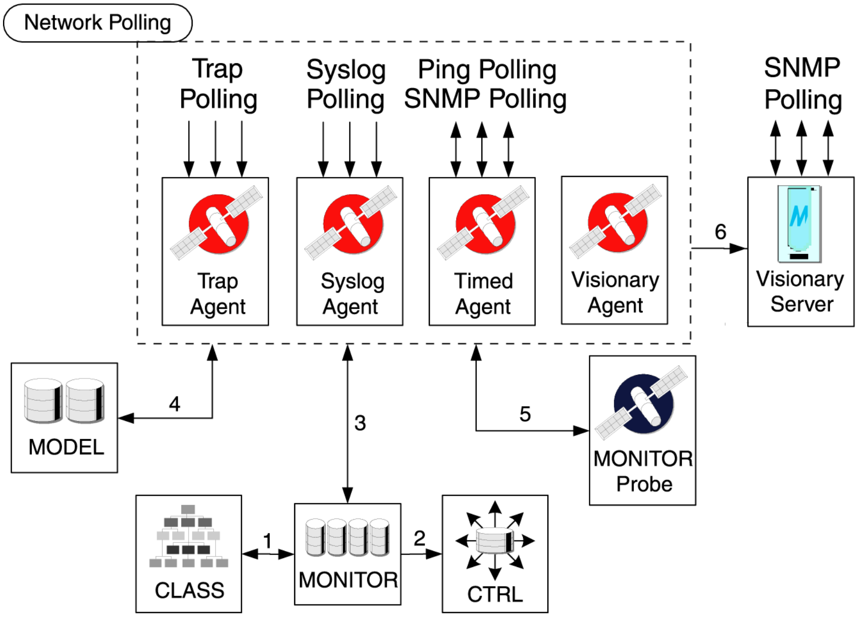


Figure 2: The Polling Process

MODEL contains the discovered network topology. For every entity in MODEL the instantiate rules have an Active Object Class (AOC) assigned. The AOC defines the type of polling required for each class of device.

When MONITOR starts, it connects to the CLASS component and downloads the poll definitions for every class (item 1 in Figure 2). Each poll definition specifies one of the following polling agent types:

- Timed
- Trap
- Syslog
- Visionary

The Trap and Syslog polling agents are collectively known as *Triggered* polling agents.

The polling agents perform monitoring, filtering and timing functions, and also launch and control stitchers.

Poll definitions specify the stitcher to run. A variety of stitchers are supplied with Netcool/Precision IP. See Chapter 4: *Stitchers Used for Polling* on page 89 for instructions on how to write text-based stitchers.

The poll definition can be configured to run separate types of polling by entering a value in the `AgentKey` field. For example:

- `PING`
- `SNMP`

The poll definition also includes any scope filters which have been applied to the class.

For each poll type, MONITOR instructs the CTRL component to start the specified polling agent (item 2 in Figure 2). If the polling agent is already running, the poll type is added to existing poll types that the polling agent is processing.

When the polling agent starts, and when it receives the new poll type, it sends a query to MONITOR to obtain a list of polls that match the `AgentKey` field in the poll definition (item 3 in Figure 2).

The polling agent then queries MODEL to obtain a list of entities that match the poll type (item 4 in Figure 2). The scope filter in the poll definition is used to filter the results from MODEL.

The polling agent polls the entities in the network. If the conditions of the poll are met, as specified by the rules in the stitcher, the polling agent sends the results as an alert to the ObjectServer. All alerts pass through the MONITOR probe (item 5 in Figure 2).

If Netcool/Visionary is installed on the network, the Visionary polling agent can be utilised to send SNMP polling data to the Netcool/Visionary server through a broker (item 6 in Figure 2).

**Note:** The Netcool/Knowledge Library is a set of rules files written to a common standard. It enables Netcool/OMNIbus probes to work seamlessly with Netcool/Precision IP without any need for configuration. The Netcool/Knowledge Library is available with your Netcool/OMNIbus installation. It is also available as a download on the Micromuse Support Site.

## Triggered Polling Agents

Triggered polling agents do not actively poll the network. They monitor the network continuously, and are activated by the receipt of either an SNMP trap or a Syslog message. The following types of triggered polling agent are available:

- Trap
- Syslog

The polling agent monitors the network. When a trap or syslog message is detected it invokes the stitcher specified in the poll definition to send the appropriate alert to the ObjectServer.

**Note:** By default, Netcool/Precision IP uses the Netcool/OMNIbus MT Trapd probes and Syslog probes to poll the network. The Netcool/Precision IP Trap and Syslog polling agents are switched off, by default, and in future releases these polling agents will be deprecated. Netcool/OMNIbus probes work seamlessly with Netcool/Precision IP without any need for configuration if you have the Netcool/Knowledge Library installed. The Netcool/Knowledge Library is available with your Netcool/OMNIbus installation. It is also available as a download on the Micromuse Support Site.

### Trap Polling Agent

When the Trap polling agent runs, it listens for the types of trap specified in the poll definition. If a trap is detected, the stitcher sends the appropriate alert to the ObjectServer through the MONITOR probe.

To specify the Trap polling agent in a poll definition, enter the executable `ncp_m_trapstitcher` in the `AgentName` field. The poll definition also specifies the stitcher that the polling agent needs to run.

The Trap polling agent listens on port `162` by default. This can be configured in the `trapAgent.configuration` table which is defined in the `MonitorTrapStitcherAgent.cfg` file.

### Syslog Polling Agent

When the Syslog polling agent runs, it monitors the system log (Syslog) files listening for updates. If an update occurs, the Syslog polling agent parses the updated sections, extracts the required information, and processes it to see if an alert needs to be generated. Alerts are sent to the ObjectServer through the MONITOR probe.

To specify the Syslog polling agent in a poll definition, enter the executable `ncp_m_syslogstitcher` in the `AgentName` field. The poll definition also specifies the stitcher that the polling agent needs to run.

## Timed Polling Agents

Timed polling agents poll the network at intervals specified within poll definitions. Based on the response, the polling agent sends the appropriate alert to the ObjectServer through the MONITOR probe.

To specify the Timed polling agent in a poll definition, enter the executable `ncp_m_timedstitcher` in the `AgentName` field.

There are two types of timed polling:

- Ping
- SNMP

These types are specified by the `AgentKey` field in the poll definition. The poll definition also specifies the stitcher that the polling agent needs to run.

### Ping Polling

The ping process is the most common way to check that a device is available from another location in the network. Ping polling checks that a device is still present, live, and contactable, by sending a packet of information on a periodic basis to an IP address and waiting for a response. Ping polling uses ICMP (Internet Control Message Protocol). On many network devices, pings are typically run as a low priority and often time out. For this reason, Ping polling may be configured to send one or more pings to a device before generating an event indicating loss of connectivity. The number of retries is configurable by the user.

### SNMP Polling

SNMP polling is used to send SNMP requests, defined in various MIBs, to devices on the network that use SNMP (Simple Network Management Protocol).

## Visionary Polling Agent

The Visionary polling agent is a monitor agent that configures Netcool/Visionary, based on the topology discovered by Netcool/Precision IP. The Visionary polling agent reacts to topology updates saved to MODEL that are within the scope of the polling agent poll definition by forwarding them to Netcool/Visionary. This polling agent is disabled by default.

The Visionary polling agent provides Netcool/Visionary 2.7 with the class-driven approach used by Netcool/Precision IP. The scope of the Visionary polling agent is configured in the same manner as the `ncp_m_timedstitcher` and can be further refined by the use of the scaling functionality contained in Netcool/Visionary. See the *Netcool/Visionary Administration Guide* for further details.

**Note:** Netcool/Visionary 2.7 supports monitoring of SNMPv1 and SNMPv2 compatible devices only. The Visionary polling agent does not forward details of SNMPv3 compatible devices to Netcool/Visionary.

**Note:** Netcool/Visionary requires a separate DSM instance to monitor devices outside of the network topology discovered by Netcool/Precision IP. See the *Netcool/Visionary Administration Guide* for instructions on how to create an additional DSM instance.

**Note:** If you stop the `ncp_m_visionary` process, Netcool/Visionary will continue monitoring the network topology.

## User-Defined Polling

You can create your own polling processes using one of the three types of polling agent. To create a new polling process you must create one or more text based stitchers. A poll definition must be written to call this new stitcher.

For information about writing stitchers and editing poll definitions, see Chapter 4: *Stitchers Used for Polling* on page 89.

## Poll Suspension

Polling is conducted on a class-by-class basis. All devices which belong to a particular class run the poll definitions assigned to that class. A class can either explicitly name the poll definitions or it can inherit the poll definitions from its parent class.

For example, the class `Cisco` contains two poll definitions and inherits one more. Every device which is instantiated to the class `Cisco` runs these three polls.

You may want to suspend polling on individual devices without changing the instantiate rules of the AOCs. It is possible to suspend polls on one or more devices by making inserts into the `polls.suspended` table within MONITOR. For information on manual poll suspension, see *Manually Suspending Polling* on page 25.

## 1.3 The Event Enrichment Process

The gateway enriches events in the ObjectServer with network topology information stored in MODEL. Event enrichment is also provided for specific event types which originate from the Netcool/Precision IP polling agents, Netcool/OMNIbus probes and other Netcool products.

Events from the Netcool/Precision IP polling agents are sent to the ObjectServer through the MONITOR probe. The gateway sends these events from the Netcool/OMNIbus ObjectServer to the Netcool/Precision IP component AMOS where root cause analysis is performed.

The gateway also sends root cause events from AMOS to the ObjectServer, and updates existing events which have been identified as symptoms of a root cause event.

For more information on the gateway, see Chapter 6: *The Event Gateway* on page 135.

## 1.4 The Root Cause Analysis Process

The Netcool/Precision IP component AMOS performs root cause analysis (RCA). It does this by correlating events with each other, and with the network topology, to determine which ones are the root causes, and which are symptoms that disappear when the root cause is resolved.

Because AMOS knows how devices in the network are connected, it can use a technique called downstream suppression to determine which devices are temporarily inaccessible due to other network failures. It suppresses the events on these temporarily inaccessible devices. Suppressed events are still visible to the user, however, they are marked as symptomatic, rather than root cause.

The way in which AMOS performs RCA is controlled by rules that are specified in the AOCs (Active Object Classes). These rules, known as the event correlation rules, allow a high degree of customization of how AMOS works, enabling it to be tailored to specific customer requirements.

For information on correlation rules, see Chapter 7: *Root Cause Analysis* on page 155.

# Chapter 2: Network Polling

This chapter describes how to use MONITOR and the polling agents to poll your network. It describes how to start MONITOR and the polling agents, both manually and automatically. It also describes how the databases of MONITOR and the polling agents can be used to suspend polling on certain devices and retrieve specific information about the polling process.

This chapter contains the following sections:

# 2.1 Starting MONITOR and Polling Agents

This section describes the process of starting MONITOR and the polling agents.

## Prerequisites

The following must be in place before MONITOR and the polling agents can be started:

- CLASS, the class loader component of the Precision Server, must be running, to ensure that MONITOR can download the poll definitions and distribute them to the polling agents.
- DISCO, the network discovery component of the Precision Server, must have successfully completed a discovery and sent it to MODEL.
- MODEL must be running in order to pass the network topology to the polling agents.
- CTRL must be running. MONITOR uses CTRL to start the polling agent executables.
- MONITOR probe must be running to transfer events to the ObjectServer.

## Starting MONITOR

Micromuse recommends that MONITOR is started using the domain process controller CTRL. The use of CTRL to automatically manage processes is described in the *Netcool/Precision IP Discovery Configuration Guide*.

**Warning:** If you are using Netcool/Precision IP with failover, you must start MONITOR using CTRL. The CTRL process checks the status of the MONITOR component and uses this information to generate the Health Check events used by the failover process. For more information on failover, see the *Netcool/Precision IP Installation and Deployment Guide*.

### Manually Starting MONITOR

On Microsoft Windows, Netcool/Precision IP components can be run as processes or as services. Components run as processes are started from a command prompt in the same way as on UNIX platforms. For more information on running components as services, see the *Netcool/Precision IP Discovery Configuration Guide*.

Run the command `ncp_monitor` to manually start MONITOR.

The command line options for `ncp_monitor` are:

```
ncp_monitor -domain DOMAIN_NAME -service SERVICE_NAME [-latency LATENCY] [-debug DEBUG]
[-backup] [-help] [-version]
```

The command line options are described in Table 2.

Table 2: Explanation of command line attributes available with MONITOR

| Command Line Option | Description |
|---|---|
| `-domain DOMAIN_NAME` | The name of the domain under which MONITOR is running. |
| `-service SERVICE_NAME` | The service to which events should be sent. This must be set to `Monitor2ObjServ` to send events to the ObjectServer.<br><br>Setting this value to `Events` sends events directly to AMOS. Micromuse does not recommend performing root cause analysis on alerts that have not been processed by the Netcool/OMNIbus ObjectServer.<br><br>The service specified when starting MONITOR is automatically passed to any polling agents that MONITOR starts. |
| `-latency LATENCY` | The maximum time in milliseconds (ms) that MONITOR waits to connect to another Precision Server process via the messaging bus. This option is useful for large and busy networks where the default settings can cause the process to assume that there is a problem when in fact the communication delay is a result of network traffic. |
| `-debug DEBUG` | The level of debugging output. Possible values are `1-4`, where `4` represents the most detailed output. |
| `-backup` | Configures MONITOR to operate in backup mode. For information on failover, see the *Netcool/Precision IP Installation and Deployment Guide*. |
| `-help` | Prints out a synopsis of all command line options for `ncp_monitor`, then exits. |
| `-version` | Prints the version number of `ncp_monitor`, then exits. |

# Starting Polling Agents

Micromuse recommends that the triggered and timed agents are started automatically by MONITOR, using CTRL, as defined in the poll definitions.

## Manually Starting Polling Agents

On Microsoft Windows, Netcool/Precision IP components can be run as processes or as services. Components run as processes are started from a command prompt in the same way as on UNIX platforms. For more information on running components as services, see the *Netcool/Precision IP Discovery Configuration Guide*.

The polling agents can be started manually by running the polling agent executables:

- `ncp_m_timedstitcher`, controls ping and SNMP polling.
- `ncp_m_visionary`, controls Netcool/Precision IP integration with Netcool/Visionary.
- `ncp_m_trapstitcher`, controls trap monitoring.
- `ncp_m_syslogstitcher`, controls syslog monitoring.

The command line options for all agents are:

```
AGENT_NAME -key AGENT_TYPE -domain DOMAIN_NAME -service SERVICE_NAME [-debug DEBUG]
[-help] [-latency LATENCY] [-version]
```

Where *AGENT_NAME* is the polling agent executable. The command line options for the agents are described in Table 3.

Table 3: Explanation of command line attributes for the Polling Agents

| Command Line Option | Description |
|---|---|
| -key *AGENT_TYPE* | The key used to ensure that the correct type of polling is carried out, for example, PING. This is the link between the AOC definition, the polling agents and the stitchers. The key used must exactly match the key specified in the poll definition which the polling agent is to use. |
| -debug DEBUG | The level of debugging output. Possible values are 1-4, where 4 represents the most detailed output. |
| -domain *DOMAIN_NAME* | The name of the domain under which MONITOR is running. |
| -service *SERVICE_NAME* | The service to which events should be sent. This must be set to Monitor2ObjServ to send events to the ObjectServer.<br><br>Setting this value to Events sends events directly to AMOS. Micromuse no longer supports root cause analysis from alerts that have not been processed by the Netcool/OMNIbus ObjectServer.<br><br>The service specified when starting MONITOR is automatically passed to any polling agents that monitor starts. |
| -help | Prints out a synopsis of all command line options for the polling agent then exits. |
| -latency *LATENCY* | The maximum time in milliseconds (ms) that MONITOR waits to connect to another Precision Server process via the messaging bus. This option is useful for large and busy networks where the default settings can cause the process to assume that there is a problem when in fact the communication delay is a result of network traffic. |
| -version | Prints the version number of the polling agent then exits. |

## 2.2 Manually Suspending Polling

You can suspend polling on individual devices using the database table `polls.suspended`, which is defined in the `NCHOME/etc/precision/MonitorSchema.cfg` file.

**Note:** `NCHOME` is the environment variable that contains the path to the Netcool Suite home directory. For information on how this environment variable varies with platform, see *Operating System Considerations* on page 9.

The columns of the `polls.suspended` table are described in Table 4.

Table 4: The polls.suspended table

| Column name | Constraints | Data type | Description |
|---|---|---|---|
| `EntityName` | Not NULL<br>PRIMARY KEY | Text | The name of the entity for which to suspend the specified poll. This should correspond to the EntityName of the entity as defined in the MODEL topology database. |
| `ClassName` | Not NULL | Text | The ClassName of the MODEL entity as defined in the MODEL topology database. |
| `PollName` | Not NULL<br>PRIMARY KEY | Text | The name of the poll to suspend. This must correspond to the PollName attribute of the poll as defined in the poll definitions of the relevant AOC file. |
| `AuditData` | | Object type vblist | An optional free-form field into which any audit data, such as the time of, and user responsible for, a particular poll suspension may be stored. |
| `ActionType` | | Int type actions | This column is for internal use only, and you should not insert values into this field. |

You must ensure that the `EntityName` and `ClassName` that you specify in the `polls.suspended` table entry exactly match the `EntityName` and `ClassName` in the MODEL `master.entityByName` database table entry for the entity.

The `PollName` that you specify in the `polls.suspended` table entry must match the name of the poll that you are suspending. Polls are named in the AOC files in which they are defined.

If the attributes do not match, the suspension has no effect.

# Logging in to the OQL service provider

In order to suspend or resume polls, log in to the service `Monitor` using the OQL (Object Query Language) service provider using the command:

```
ncp_oql -domain DOMAIN_NAME -service Monitor -username USERNAME
```

Where *`USERNAME`* is the user name and *`DOMAIN_NAME`* is the domain name to connect to. For more information on using the OQL service provider, see the *Netcool/Precision IP Discovery Configuration Guide*. You need to log into the OQL service provider in order to access the `polls.suspended` table.

# Suspending Polling

After you have logged in to the OQL service provider, you can suspend a poll on a device by making an insert into the `polls.suspended` table. The following example suspends a default ping poll on `router01`.

```
1.> insert into polls.suspended
2.> (EntityName, ClassName, PollName)
3.> values
4.> ("router01", "Cisco72xx", "defaultPing");
5.> send;
```

The poll is immediately suspended. If this poll is currently in the process of executing when it is suspended, it completes that execution before the suspension becomes effective. It is therefore possible for a poll to generate an event after it has been suspended. This is most likely to happen with polls which have a long timeout period, for example, ping polls which wait for several seconds to receive a response.

The `polls.suspended` table is persistent and the information it contains is still present if `ncp_monitor` or Netcool/Precision IP is restarted. To resume a poll you must delete the relevant entry in the `polls.suspended` table.

## Entering Audit Data

You can enter audit data as name-value pairs in the `AuditData` field. The `AuditData` field can be empty, can contain a single name-value pair, or can contain multiple, comma separated, name-value pairs. For example:

```
1.> insert into polls.suspended
2.> (EntityName, ClassName, PollName, AuditData)
3.> values
4.> ("router01", "Cisco72xx", "defaultPing", { user = "admin", date = "20/01/2003", time
= "11:56:27",reason = "Disable poll to allow routine maintenance."});
5.> send;
```

### Suspending All Polling

To suspend all polls on a specific device, insert an asterisk `"*"`, in the `PollName` field. For example, to suspend all polls on `router01`, log in to the OQL service provider and enter:

```
1.> insert into polls.suspended
2.> (EntityName, ClassName, PollName)
3.> values
4.> ("router01", "Cisco72xx", "*");
5.> send;
```

To suspend all polls on all devices, insert an asterisk in each field. For example, to suspend all polls enter:

```
1.> insert into polls.suspended
2.> (EntityName, ClassName, PollName)
3.> values
4.> ("*", "*", "*");
5.> send;
```

## Resuming Polling

To resume a poll which has been suspended, delete the relevant entry from the `polls.suspended` table. For example, to resume a default ping poll on `router01`, log in to the OQL service provider and enter:

```
1.> delete from polls.suspended where
2.> EntityName = "router01"
3.> and PollName = "defaultPing";
4.> send;
```

The poll is immediately resumed, and executes again at the appropriate time as defined in the poll definitions.

### Resuming All Polling on a Specific Device

If you have suspended all polls on a specific device using the `"*"` wildcard, you can resume normal polling on that device by deleting this entry in the `polls.suspended` table. For example, to resume normal polling on the device `router01`, log in to the OQL service provider and enter:

```
1.> delete from polls.suspended where
2.> EntityName = "router01"
3.> and PollName = "*";
4.> send;
```

**Note:** Deleting a wildcard entry in the `polls.suspended` table leaves any other entries for that device unchanged. Specifically, if you have suspended any polls for that device using inserts which explicitly identify polls by name, these suspensions remain in effect.

If you wish to resume *all* polling on the device `router01`, do not specify the PollName value. For example, to resume all polling on the device `router01`, log in to the OQL service provider and enter:

```
|1.> delete from polls.suspended where
|2.> EntityName = "router01"
|3.> send;
```

## 2.3 Default Polling Process Descriptions

This section describes the default polling process. The process flow descriptions are intended for administrators of Netcool/Precision IP who may need to customize the operation of these polling agents.

You can configure the polling process by creating and applying new stitchers. You should understand all aspects of Precision Server and RCA Engine functionality, and have an in-depth knowledge of the polling process, before attempting to write your own stitchers. For detailed information on the functionality of the stitchers, see Chapter 4: *Stitchers Used for Polling* on page 89.

For a basic description of each polling agent, see *The Polling Process* on page 13.

### Ping Polling

Ping polling is used to ensure that a device is still present, live and contactable in the network by periodically sending a packet of information to an IP address and waiting for a response.

The possible results of a poll are:

- Success
- Fail
- Restore

The results are described in the following sections.

#### Poll Success

A device in the network continues to be contactable. The agent process is:

1. The agent executable `ncp_m_timedstitcher` runs based on the `Frequency` attribute of the poll definition.
2. The executable starts the stitcher specified in the poll definition, which sends a ping to the appropriate network device.
3. The device sends a response within the `TimeOut` period specified in the poll definition.
4. The stitcher checks the internal test flag to determine if not the device was reachable the last time it was polled. Since it was (the flag was set to `pass`), a restore event is not generated.
5. The poll restarts when next initiated by the `Frequency` attribute.

## Poll Failure

A device is no longer contactable. The agent process is:

1. The agent executable `ncp_m_timedstitcher` runs based on the `Frequency` attribute of the poll definition.

**Note:** You must enter a time period in the `Frequency` attribute of the poll definition.

2. The executable starts the stitcher specified in the poll definition, which sends a ping to the appropriate network device.
3. The device fails to respond within the `TimeOut` period specified in the poll definition.
4. The stitcher generates an event and sends it to the MONITOR probe with an appropriate name assigned to the `EventName` column.
5. The internal test flag is set to `fail` to acknowledge the fact that the device could not be contacted.
6. The poll restarts when next initiated by the `Frequency` attribute.

The column names and values of the ping fail event depend on the poll definitions and the configuration of the stitcher used to perform the ping poll. A typical example is given below in Table 5.

Table 5: The Column Names of an Example Ping Fail Event

| Column Name | Example Value |
|---|---|
| EventId | 1227 |
| EntityName | Router4500.1234 |
| ClassName | Cisco |
| Description | Ping fail for 192.168.34.56 |
| EventName | pingFail |
| RuleSet | pingFailToCorrelatedRootCause |
| EventType | Event |
| Severity | Major |
| AssignedTo | Joe |
| Acknowledged | Unacknowledged |
| AgentAddress | 192.168.123.146 |
| EventGroupID | 1227 |

### Poll Restore

A device that was previously unreachable (on the last ping attempt), becomes reachable again. The agent process is:

1. The agent executable `ncp_m_timedstitcher` runs based on the `Frequency` attribute of the poll definition.
2. The executable starts the stitcher specified in the poll definition, which sends a ping to the appropriate network device.
3. The device sends a response within the `TimeOut` period specified in the poll definition.
4. The stitcher checks the internal test flag to determine if not the device was reachable the last time it was polled. Since it was not (the flag was set to `fail`).
5. The stitcher generates a restore condition. It sends an event to the MONITOR probe indicating the device is now reachable and resets the internal test flag to `pass`.
6. The poll restarts when next initiated by the `Frequency` attribute.

## SNMP Polling

SNMP polling is used to acquire MIB-related information from particular network devices. It is associated with the SNMP protocol and the SNMP polling agent. The possible outcomes from an SNMP poll are:

- Success
- Fail

These polling outcomes are described in the following sections. The concept of delta polling, a more advanced type of SNMP polling, is described in *Delta Polling* on page 33.

Some stitchers also support restore events, using an internal test flag in a similar way to the Ping process flow.

### Poll Success

An SNMP poll is successful if the device is contactable within a timeout period. However, if the event generation conditions are not met, the associated stitcher does not generate an alert.

Several event generation conditions can be specified in the stitcher, which may be simple or complex, and may even be logically dependant on each other.

For example, the following process describes a simple event generation condition that checks whether a contactable device has restarted in the last two minutes. The agent process is:

1. The agent executable `ncp_m_timedstitcher` runs an SNMP poll every two minutes, based on the setting `Frequency=120` in the poll definition.
2. The executable starts the stitcher specified in the poll definition.
3. The stitcher contacts the device, within the `TimeOut` period, and retrieves the MIB variable `sysUpTime`. This MIB variable provides the time in hundredths of a second since the device was last initialized. The variable can be specified in the `Threshold` attribute of the poll definition or written into the stitcher.
4. The first time a device is polled, the value of `sysUpTime` is retrieved and stored. In this case the stitcher does not check the event generation condition.
5. On the next poll, the agent compares the `sysUpTime` MIB variable values from the last two polls.

   If the device has been continually running, the difference between the two values is greater than zero and no event is generated.

   If the device has restarted, the difference is less than zero and the stitcher sends an event to the MONITOR probe to indicate the device has restarted.
6. The poll restarts when next initiated by the `Frequency` attribute.

## Poll Failure

The SNMP polling fails when the device is unreachable within timeout period. The event generation condition is not evaluated.

For example, the following process describes a SNMP poll to a device that is no longer contactable. The agent process is:

1. The agent executable `ncp_m_timedstitcher` runs an SNMP poll based on the `Frequency` attribute of the poll definition.
2. The stitcher attempts to retrieve the required MIB variable from the network device. The device is not contactable and fails to respond within the `TimeOut` period.
3. The stitcher sends an event to the MONITOR probe with an appropriate name assigned in the `EventName` column name. The event generation condition is not evaluated.
4. The poll restarts when next initiated by the `Frequency` attribute.

### Delta Polling

A delta poll generates events based on the differences between the previous and current values received by one or more MIB variables. The example in *Poll Success* on page 29, that compares values of the MIB variable `sysUpTime`, is an example of delta polling.

Stitcher rules which use delta polling functionality are available in some stitchers and for construction of user-defined stitchers. For more information on the stitcher rules, see Chapter 4: *Stitchers Used for Polling* on page 89.

The process flow for delta polling is the same as for other SNMP polling, but the event generation condition is likely to be more complicated.

### Polling Multiple MIB Variables

SNMP polling can also be used to retrieve several SNMP variables at once from a device.

## Trap Monitoring

Traps are asynchronous notifications that enable a network entity to report a condition to a management station. Trap agents are reactive and have a much simpler mechanism than the timed polling agents. Trap monitoring is defined using the poll definition `AgentControl` and `StitcherInfo` attributes. The `AgentControl` attribute specifies which traps should be handled and which should be discarded.

The traps that are commonly used with the SNMP protocol are listed in Table 6.

Table 6: Trap Type Values and Descriptions (1 of 2)

| Trap Type | Trap Name | Description |
|---|---|---|
| 0 | `coldStart` | Signifies that the sending device is reinitializing itself and may have been altered. |
| 1 | `warmStart` | Signifies that the sending device is reinitializing itself and has not been altered. |
| 2 | `linkDown` | Signifies that the sending device recognizes the failure of a communication link. |
| 3 | `linkUp` | Generated when a recognized communication link comes up or gets restored. |
| 4 | `authenticationFailure` | Generated when the sending device receives a message that is not properly authenticated, for example, an incorrect login attempt. |

Table 6: Trap Type Values and Descriptions (2 of 2)

| Trap Type | Trap Name | Description |
|---|---|---|
| 5 | egpNeighborloss | Signifies that an Exterior Gateway Protocol (EGP) neighbor, for whom the sending device was an EGP peer, has been marked down and the relationship no longer exists. |
| 6 | Enterprise Specific Trap Name | Signifies that the sending device recognizes some enterprise-specific event has occurred. This value is used for any trap that does not match trap type values 1 to 5. |

A trap that is defined in a MIB in the NCHOME/precision/mibs directory is called a *known* trap. A trap that is not defined in a MIB is called an *unknown* trap.

## Known Trap

The following steps describes the polling agent process when a known trap is detected:

1. The agent executable ncp_m_trapstitcher is already running, having been initiated at startup. It listens on the default port 162 for incoming traps.
2. Using the TrapType value in the trap PDU (Packet Data Unit) the agent attempts to resolve the trapname from the MIBs in the NCHOME/precision/mibs directory.

   If TrapType=6 the agent attempts to resolve the trapname using the Enterprise value and the SpecificTrapType value.

   If the trapname cannot be resolved the trap is unknown. For information on the agent process for an unknown trap, see *Unknown Trap* on page 35.
3. If the trapname can be resolved, the agent identifies which device the trap originated from using the IP address contained in the trap. The agent then checks the network topology to find out the class of the device.
4. The class determines which poll definitions are loaded. This is done for every trap. In each poll definition the trap is managed if any of the following conditions are met:
   - The trap is listed in the KnownTraps section of the AgentControl field.
   - The value ALL is listed in the KnownTraps section of the AgentControl field.
   - The value Unhandled is listed in the KnownTraps section of the AgentControl field and the trap is not listed in the KnownTraps section of any other poll definition for this class.

   The agent then checks that the trap does not appear in the OmitTraps field.
5. If the poll definition conditions are met, the agent executable starts the stitcher. The stitcher constructs an event describing the trap and where it came from.
6. The agent waits for another trap to be received.

### Unknown Trap

The following steps describes the polling agent process when an unknown trap is detected:

1. The agent executable `ncp_m_trapstitcher` is already running, having been initiated at startup. It listens on the default port `162` for incoming traps.
2. The agent attempts to resolve the `trapname` from the MIBs in the `NCHOME/precision/mibs` directory. If the `trapname` cannot be resolved the trap is unknown.
3. The agent checks the unknown trap against the attribute `UnknownTrapHandling` in each poll definition.
   - If `UnknownTrapHandling` is set to `true`, the agent starts the stitcher and sets `trapname` to `unknown trap`. The stitcher constructs and sends an event.
   - If `UnknownTrapHandling` is set to `false`, the agent does not start the stitcher.
4. The agent waits for another trap to be received.

## Syslog Polling

Syslog polling parses syslog files and analyzes the results to detect new messages. Syslogs allow a device to deliver messages to another device. The syslog message format commonly includes values for date, time, and the service or process that generated the message. Messages can indicate the occurrence many different events, for example, communication links going up or down, or failed root login attempts.

Syslog polling, like trap monitoring, is reactive, and it has an even simpler process flow. It does not have fail, restore or success outcomes as such. The possible outcomes from an Syslog poll are:

- Message found
- No message found

These polling outcomes are described in the following sections.

## Syslog Message Found

This section describes the polling agent process when the agent detects a message and the message matches the values in the AgentControl field. The poll definition Filename attribute specifies the location of the file containing the syslog messages. The default value is Filename=["/var/adm/messages"]. The agent process is:

1. The agent executable ncp_m_syslogstitcher parses the file looking for messages.
2. The agent finds the message and checks its fields against those specified in the poll definition AgentControl field.
3. If one or more of the fields match, the agent starts the stitcher and passes the message to it.
4. The stitcher processes the message, extracts the information in the relevant fields and generates an event from it.
5. The agent continues to monitor the files.

## No Syslog Message Found

This section describes the polling agent process when the agent does not detect a message or the message does not match the values in the AgentControl field. The agent process is:

1. The agent executable ncp_m_syslogstitcher parses the file looking for messages.
2. There are no new messages there, or the messages do not match the values specified in AgentControl. The agent does not start a stitcher.
3. The agent continues to monitor the files.

# 2.4 MONITOR Database Reference

This section describes the database tables used by MONITOR (the `ncp_monitor` executable). The poll definitions, AOC definitions and agent status are stored in the MONITOR databases.

**Note:** This section is intended for advanced users who want to interrogate the Netcool/Precision IP MONITOR databases.

## The polldefCache Database Schema

The summary information for the `polldefCache` database schema is shown in Table 7.

Table 7: polldefCache Database Summary

| | |
|---|---|
| **Database name** | `polldefCache` |
| **Defined in** | `NCHOME/etc/precision/MonitorSchema.cfg` |
| **Fully qualified database table name** | `polldefCache.polldefs` |

### The polldefs Table

The `polldef` table holds the poll definitions that are loaded from CLASS. The columns are described in Table 8.

Table 8: polldefCache.polldefs Table Descriptions (1 of 2)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `PollName` | PRIMARY KEY<br>NOT NULL | Text | The unique name of the poll definition. |
| `PollStatus` | NOT NULL | Integer | The poll status. Possible values are:<br>• `1` - poll is active<br>• `0` - poll is inactive |
| `AgentName` | NOT NULL | Text | The name of the polling agent used to conduct this poll. |
| `AgentKey` | | Text | The agent key, which links the AOC definition with the polling agent executable and the stitchers it employs.<br>The agent key distinguishes between polls that use the same agent, for example, the timed stitcher agent which runs both Ping and SNMP polls. Using the agent key ensures that two separate instances of the executable `ncp_m_timedstitcher` are run (one for each type of poll). |

Table 8: polldefCache.polldefs Table Descriptions (2 of 2)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `HostName` | | Text | The host machine from which polling is being conducted. |
| `Frequency` | | Integer | How often the poll is conducted. |
| `Threshold` | | Text | A threshold condition for the poll. |
| `Scope` | | Text | A filter that constrains poll execution to certain devices, classes or instances if necessary. |
| `ClassName` | PRIMARY KEY<br>NOT NULL | Text | The name of the class within which this poll is defined. |
| `StitcherName` | NOT NULL | Text | The name of the stitcher that the agent calls. |
| `AgentControl` | Externally defined vblist data type | Object | A list of agent control information, for example, to define how to handle trap and syslog monitoring. |
| `StitcherInfo` | Externally defined vblist data type | Object | A list of stitcher Information. |
| `ActionType` | NOT NULL | Integer | The type of action the event represents. Possible values are:<br>• `0` - Create<br>• `1` - Change<br>• `2` - Delete |

# The class Database Schema

The summary information for the `class` database schema is shown in Table 9.

Table 9: class Database Summary

| | |
|---|---|
| **Database name** | `class` |
| **Defined in** | `NCHOME/etc/precision/MonitorSchema.cfg` |
| **Fully qualified database table name** | `class.activeClasses` |

## The activeClasses Table

The `activeClasses` table holds a copy of the full definition of every AOC active in Netcool/Precision IP. The columns are described in Table 10.

Table 10: class.activeClasses Table Descriptions

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `ClassName` | PRIMARY KEY<br>NOT NULL<br>UNIQUE | Text | Name of the AOC. |
| `SuperClass` | NOT NULL | Text | Name of the parent AOC. |
| `Dictionary` | | List of text | List of data dictionaries used by the AOC. |
| `Instantiate` | NOT NULL | Text | Rules for instantiating the AOC. |
| `Extensions` | Externally defined extension data type | Object of extension data type | List of extensions contained within the AOC. |
| `VisualIcon` | NOT NULL | Text | The icon associated with this AOC. |
| `MenuRules` | Externally defined `menurule` data type | List of object data types (the object is of the `menurule` data type) | A list of menu rules associated with the AOC. |
| `Menu` | Externally defined menu data type | List of object data types (object is of the menu data type) | List of menu options available in the GUI for this AOC. |
| `ActionType` | Externally defined actions data type | Integer | The type of action the event represents. Possible values are:<br>• `0` - Create<br>• `1` - Change<br>• `2` - Delete |

# The agentInfo Database Schema

The summary information for the `agentInfo` database schema is shown in Table 11.

Table 11: agentInfo Database Summary

| | |
|---|---|
| **Database name** | `agentInfo` |
| **Defined in** | `NCHOME/etc/precision/MonitorSchema.cfg` |
| **Fully qualified database table name** | `agentInfo.master` |

### The master Table

The `agentInfo.master` table holds information about the active polling agents so that MONITOR can track their status and also launch them as appropriate by sending inserts to the `services.inTray` table of CTRL. The columns are described in Table 12.

Table 12: agentInfo.master Table Description

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `AgentName` | NOT NULL | Text | The name of the polling agent. |
| `AgentKey` | | Text | The agent key. |
| `HostName` | | Text | The name of the host where the polling agent is running. |

## The polls Database Schema

Table 13 shows the summary information for the `polls` database schema.

Table 13: polls Database Summary

| | |
|---|---|
| **Database name** | `polls` |
| **Defined in** | `NCHOME/etc/precision/MonitorSchema.cfg` |
| **Fully qualified database table name** | `polls.suspended` |

### The suspended Table

The `polls.suspended` table is used to suspend specified polls on specific entities. The columns are described in Table 14.

Table 14: polls.suspended Table Description (1 of 2)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `EntityName` | NOT NULL<br>PRIMARY KEY | Text | The name of the entity for which to suspend the specified poll. This should correspond to the `EntityName` of the entity as defined in the topology database. |
| `ClassName` | NOT NULL | Text | The class name of the AOC associated with this entity. |
| `PollName` | NOT NULL<br>PRIMARY KEY | Text | The name of the poll to suspend. This should correspond to the `PollName` attribute of the poll as defined in the relevant AOC file.<br>* indicates that all polls should be suspended. |

Table 14: polls.suspended Table Description (2 of 2)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `AuditData` | Externally defined vblist data type | Object | An optional field into which any audit data may be stored. For example, audit data might indicate the user responsible for a particular poll suspension, or the time of the suspension. |
| `ActionType` | Externally defined actions data type | Integer | This column is for internal use only. |

# The config Database Schema

Table 15 shows the summary information for the `config` database schema.

Table 15: config Database Summary

| | |
|---|---|
| **Database name** | `config` |
| **Defined in** | `NCHOME/etc/precision/MonitorSchema.cfg` |
| **Fully qualified database table name** | `config.failover` |

## The failover Table

The `config.failover` table contains the failover configuration and current failover state of the MONITOR component. The columns are described in Table 16.

Table 16: config.failover Table Description

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `BackupMonitor` | NOT NULL | Boolean | This value is true if MONITOR is started using the `-backup` command line option. Possible values are:<br>• `0` - Not configured as the backup system<br>• `1` - Configured as the backup system |
| `Failedover` | NOT NULL | Boolean | The failover state. Possible values are:<br>• `0` - Not in a failover state<br>• `1` - In a failover state |

## 2.5 Polling Agent Database Reference

When the polling agents are launched, they load their configuration files to create their databases and tables.

**Note:** This section is intended for advanced users who want to interrogate the Netcool/Precision IP polling agent databases.

The polling agent databases are defined in the files listed in Table 17.

Table 17: Polling Agent Configuration Files

| Polling Agent | Configuration File |
|---|---|
| Timed | `NCHOME/etc/precision/MonitorTimedStitcherAgent.cfg` |
| Syslog | `NCHOME/etc/precision/MonitorSysLogStitcherAgent.cfg` |
| Trap | `NCHOME/etc/precision/MonitorTrapStitcherAgent.cfg` |
| Visionary | `NCHOME/etc/precision/MonitorVisionaryAgent.cfg` |

The Timed polling agent is used for ping polling and SNMP polling. The polling agent databases can be divided into generic and specialized groups.

Generic databases are defined in every polling agent configuration file. These databases hold the network topology that is to be polled and the polling methodology. The generic databases are:

- `topoCache`
- `polldefCache`

Specialized databases are polling agent specific. These databases are handled internally and you do not normally need to configure inserts into them. The specialized databases are:

- `triggers` (for Syslog polling)
- `trapAgent`
- `triggers` (for Trap polling)

These database tables are described in the following sections.

# The topoCache Database Schema

The summary information for the topoCache generic database schema is shown in Table 18.

Table 18: topoCache Database Summary

| | |
|---|---|
| **Database name** | topoCache |
| **Defined in** | NCHOME/etc/precision/MonitorTimedStitcherAgent.cfg<br>NCHOME/etc/precision/MonitorSysLogStitcherAgent.cfg<br>NCHOME/etc/precision/MonitorTrapStitcherAgent.cfg |
| **Fully qualified database table name** | topoCache.entityByName |

## The entityByName Table

The entityByName table holds a simplification of the topology information in the master database of MODEL. The columns are described in Table 19.

Table 19: topoCache.entityByName Table Description (1 of 2)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| ObjectId | PRIMARY KEY<br>NOT NULL<br>UNIQUE | Long integer | Unique Object ID of the network entity. |
| EntityName | PRIMARY KEY<br>NOT NULL<br>UNIQUE | Text | Unique descriptive name of a network entity. |
| Address | | List of text | List of OSI model layer 1-7 addresses for the entity. |
| Description | | Text | Value of sysDescr MIB variable or other suitable description of the entity. |

Table 19: topoCache.entityByName Table Description (2 of 2)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| EntityType | Externally defined entityTypes data type | Integer | Element type of the entity. Possible values are:<br>• 0 - Unknown<br>• 1 - Chassis<br>• 2 - Interface<br>• 3 - Logical interface<br>• 4 - Vlan object<br>• 5 - Card<br>• 6 - PSU<br>• 7 - Subnet<br>• 8 - Module |
| ClassName | | Text | The class name of the network entity (if applicable). |
| EntityOID | | Text | Value of the sysOID MIB variable of the entity. |
| Status | Externally defined status data type | Integer | Flag showing status of the network entity. |
| Security | | Text | Password to access network entity (if applicable). |
| RelatedTo | | List of text | List of connections to the network entity. |
| Contains | | List of text | List of elements or other containers contained within the current network entity. |
| UpwardConnections | | List of text | List of containers that contain this entity. |
| IsActive | Externally defined boolean data type | Integer | Flag indicating whether an Active Object Class is needed. |
| CreateTime | | Time | Creation time of network entity record in table. |
| ChangeTime | | Time | Time of last modification to the network entity record. |
| ActionType | Externally defined actions data type | Integer | Type of record. |

# The polldefCache database schema

The summary information for the `polldefCache` generic database schema is shown in Table 20.

Table 20: polldefCache Database Summary

| | |
|---|---|
| **Database name** | `polldefCache` |
| **Defined in** | `NCHOME/etc/precision/MonitorTimedStitcherAgent.cfg`<br>`NCHOME/etc/precision/MonitorSysLogStitcherAgent.cfg`<br>`NCHOME/etc/precision/MonitorTrapStitcherAgent.cfg` |
| **Fully qualified database table name** | `polldefCache.polldefs` |

## The polldefs table

The `polldefs` table stores the full definition of the polling methodology, loaded from CLASS through MONITOR. The columns are described in Table 21.

Table 21: polldefCache.polldefs Table Description (1 of 2)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `PollName` | PRIMARY KEY<br>NOT NULL | Text | The unique name of the poll. |
| `PollStatus` | NOT NULL | Integer | The status of the poll. |
| `AgentName` | NOT NULL | Text | The name of the polling agent executable used to conduct this poll. |
| `AgentKey` | | Text | The agent key, which links the AOC definition with the polling agent executable and the stitchers it employs.<br>The agent key distinguishes between polls that use the same agent, for example, the timed stitcher agent which runs both Ping and SNMP polls. Using the agent key ensures that two separate instances of the executable `ncp_m_timedstitcher` are run (one for each type of poll). |
| `HostName` | | Text | The host machine from which Polling is being conducted. |
| `Frequency` | | Integer | How often the poll is conducted. This column is only relevant to the timed agents and has no effect on the non-timed agents (Trap and Syslog). |
| `Threshold` | | Text | A threshold condition for the poll. |

Table 21: polldefCache.polldefs Table Description (2 of 2)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `Scope` | | Text | A filter that constrains poll execution to certain devices, classes or instances. |
| `ClassName` | PRIMARY KEY<br>NOT NULL | Text | The name of the class to which the event belongs. |
| `StitcherName` | NOT NULL | Text | The name of the stitcher that the agent will call. |
| `AgentControl` | Externally defined vblist data type | Object | A list of agent control information, for example, to define how to handle specific traps. |
| `StitcherInfo` | Externally defined vblist data type | Object | A list of stitcher Information. |
| `ActionType` | NOT NULL | Integer | The type of action the event represents. Possible values are:<br>• `0` - Create<br>• `1` - Change<br>• `2` - Delete |

# The triggers Database Schema for Syslog Polling

The `triggers` database is created for the Syslog polling agent. The summary information for the `triggers` database schema is shown in Table 22.

Table 22: triggers Database Summary

| | |
|---|---|
| **Database name** | `triggers` |
| **Defined in** | `NCHOME/etc/precision/MonitorSysLogStitcherAgent.cfg` |
| **Fully qualified database table name** | `triggers.despatch` |

## The despatch Table

The `despatch` table stores information about the device from which a syslog message has been received. The columns are described in Table 23.

Table 23: triggers.despatch Table Description

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `FieldNames` | Externally defined vblist data type | Object | A list of field names. |
| `Mappings` | | List of text | A list of mappings. |

# The trapAgent Database Schema

The `trapAgent` database is created for the trap polling agent. The summary information for the `trapAgent` database schema is shown in Table 24.

Table 24: trapAgent Database Summary

| | |
|---|---|
| **Database name** | `trapAgent` |
| **Defined in** | `NCHOME/etc/precision/MonitorTrapStitcherAgent.cfg` |
| **Fully qualified database table name** | `trapAgent.configuration` |

## The configuration Table

The `configuration` table holds configuration information for the Trap Agent. The columns are described in Table 25.

Table 25: trapAgent.configuration Table Description

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `TrapPort` | Default = 162 | Integer | The port on which to listen for traps. |
| `SourceByPayload` | Default = 1 | Integer | Configures the way in which the trap source address is retrieved. Possible values are:<br>• `0` - Retrieve the trap source address from the IP header.<br>• `1` - Retrieve the trap source address from the payload, if possible, or the IP header if there is no address in the payload. |
| `UnknownDeviceClass` | | Text | A text string to be used for handling devices for which there is no AOC definition. |

## Example Configuration of the Trap Agent

The Trap agent can retrieve the source address of the trap from either the payload or the header.

Inserting a value of 1 (the default setting) into the `trapAgent.configuration.SourceByPayload` column configures the Trap polling agent to attempt to retrieve the trap source address from the trap payload. If there is no address in the payload, the Trap polling agent uses the header source address instead.

Inserting a value of 0 into the `trapAgent.configuration.SourceByPayload` column configures the Trap polling agent to retrieve the trap source address from the header.

The following example insert shows how you might configure the Trap polling agent.

```
insert into trapAgent.configuration
(
                TrapPort, SourceByPayload, UnknownDeviceClass
)
values
(
                162, 1, "UnknownDevice"
);
```

The above example insert configures the Trap polling agent to:

- Listen for traps on port 162.
- Attempt to retrieve the trap source address from the trap payload.
- Use the string `UnknownDevice` to handle traps from devices with no AOC.

# The triggers Database Schema for Trap Polling

The `triggers` database is created for the trap polling agent. The summary information for the `triggers` database schema is shown in Table 26.

Table 26: triggers Database Summary

| | |
|---|---|
| **Database name** | `triggers` |
| **Defined in** | `NCHOME/etc/precision/MonitorTrapStitcherAgent.cfg` |
| **Fully qualified database table name** | `triggers.despatch` |

## The despatch Table

The `despatch` table contains information about the device on which a trap has been received, such as the class to which that device belongs (and the monitoring policies that should therefore be applied to that device).

The Trap agent inserts the received details into the `triggers.despatch` table which initiates the necessary stitcher. The stitcher and trigger record combination define how the trap is handled. An event record is constructed in the stitcher and inserted into the `mojo.events` database for use by an event correlation engine such as the RCA Engine. The columns are described in Table 27.

Table 27: triggers.despatch Table Description (1 of 2)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `Community` | | Text | A community string. |
| `Enterprise` | | Text | The type of enterprise. |

Table 27: triggers.despatch Table Description (2 of 2)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| AgentAddress | | Text | The address of the agent. |
| TrapType | | Integer | The type of trap received. |
| SpecificTrapType | | Integer | The specific type of trap. |
| TrapName | | Text | The name of the trap. |
| TrapDescription | | Text | A description of the trap. |
| TimeTicks | | Long integer | The time ticks (in hundredths of a second) since the system was last initialized. |
| ResolvedVarBinds | Externally defined text data type | Object | Resolved varbinds. |
| UnResolvedVarBinds | Externally defined text data type | List type text | Unresolved varbinds. |

# Chapter 3: MONITOR Configuration Tool

This chapter describes how to create, edit and browse the Netcool/Precision IP active object classes (AOCs) using the MONITOR Configuration tool. The AOC files include the polling definitions used by MONITOR.

This chapter contains the following sections:

# 3.1 Overview of the MONITOR Configuration Tool

The MONITOR Configuration tool is a user interface used to create, edit and browse active object classes (AOCs). It is the recommended method of making adjustments to the AOCs. An explanation of the structure and function of the AOCs can be found in the *Netcool/Precision IP Discovery Configuration Guide*. The event correlation rules, and extensions to the AOCs that control root cause analysis (RCA), are described in Chapter 7: *Root Cause Analysis* on page 155.

The MONITOR Configuration tool allows you to customize all attributes of the AOCs using dedicated editor windows. All these editors are described in detail in this chapter.

**Note:** The MONITOR Configuration tool is not available on Windows. For more information on customizing the AOCs on Windows, see *Customizing the AOCs Manually* on page 52.

## Poll Definitions

The poll definitions have many important functions. They define how, when and where MONITOR and the polling agents poll the network. For example, they specify how often a device is polled, the type of polling agent employed to do the polling, and the information that is collected during the polling process.

The poll definitions are part of the extensions to the AOCs and can be constructed or customized using the *Poll Editor* window in the MONITOR Configuration tool.

Like other attributes of the AOCs, the poll definitions are inherited by child classes from their parent classes unless locally overriden. For more information on AOC syntax, see the relevant chapter in the *Netcool/Precision IP Discovery Configuration Guide*.

## Event Correlation Rules

The event correlation rules (also known as *Event Methods*) within an AOC cannot be edited using the MONITOR Configuration tool. You must manually create or edit these rules using a text editor. For information on the event correlation rule syntax, see *The Event Correlation Rules* on page 173.

## Customizing the AOCs Manually

The MONITOR Configuration tool is not available on Windows. All aspects of the AOCs can be customized manually, by editing the AOC text files.

For more information on editing the AOCs manually, including AOC architecture and syntax, backing up the AOC files, and an overview of the components of the AOCs, see the *Netcool/Precision IP Discovery Configuration Guide*.

For detailed information on the attributes that can be used in the poll definitions, see *Attributes of the Poll Definitions* on page 75.

For detailed information on the event correlation methods, see *Event Rule Attributes* on page 174.

# 3.2 Starting the MONITOR Configuration Tool

Before starting the MONITOR Configuration tool you must ensure the following components are running:

- CLASS: the component that contains the AOC definitions.
- AUTH: the component that authenticates the users.

This section describes the process of starting the MONITOR Configuration tool. It also describes the configuration of CLASS and the creation of users in AUTH.

## Configuring CLASS for the MONITOR Configuration Tool

CLASS contains the AOC definitions. The MONITOR Configuration tool downloads the AOC definitions from CLASS and returns any changes. CLASS broadcasts the changes to the other Precision Server components that require the AOC definitions. For more information about CLASS, see the *Netcool/Precision IP Discovery Configuration Guide*.

Before starting the MONITOR Configuration tool, you can start CLASS with the `-read_aocs_from` command line option. This option determine which AOC definitions CLASS reads and subsequently sends to the MONITOR Configuration tool.

If you intend to create new class definitions using the MONITOR Configuration tool, you should create a directory to store the new AOCs and instruct CLASS to write the AOC files to this directory at regular intervals using the `-write_aocs_to` command line option. If CLASS terminates, there is an up-to-date version of the AOCs in a text format in addition to the cache. The output AOC text files can be used as the input for CLASS when it is restarted.

You should *not* write new AOCs back to the directory from which they are read.

The CLASS command line options are described in full in the *Netcool/Precision IP Discovery Configuration Guide*

## Configuring AUTH for the MONITOR Configuration Tool

AUTH is needed to authenticate your MONITOR Configuration tool session.

All users of the Netcool/Precision IP are assigned user profiles which specify the actions that they are allowed to perform. Setting up user profiles is described in the *Netcool/Precision IP Discovery Configuration Guide*. As the MONITOR Configuration tool only interacts directly with AUTH and CLASS, only those permissions relating to access to the service `Class` are relevant to users of the MONITOR Configuration tool. In this context, a user can have three possible permission configurations:

- OQL Read/Write access
- OQL Read access
- No OQL access

All of the operations described in this chapter require OQL Read/Write access.

If you are using a user profile which has only OQL Read access, the MONITOR Configuration tool functionality is unchanged, but you can only view the information. All functions which change any information are disabled.

If you are using a user profile which has no OQL access, a message is displayed notifying you that you do not currently have permission to use the MONITOR Configuration tool, and the MONITOR Configuration tool does not start.

## MONITOR Configuration Tool User Modes

The MONITOR Configuration tool can be run in two modes; high level (the default) and low level. The editors which are available in the MONITOR Configuration tool are defined by the mode of operation.

The mode is set by the `-usermode` command line option.

If you need to edit the poll definitions, you must run the MONITOR Configuration tool in low level mode. In this mode each MONITOR Configuration tool Class icon includes a Poll Editor button. MONITOR Configuration tool command line options are described in the next section.

# Starting the MONITOR Configuration Tool

To start the MONITOR Configuration tool enter the command `ncp_monitorconfig` and the required command line options. The command line options for the MONITOR Configuration tool are:

```
ncp_monitorconfig -domain DOMAIN_NAME [-usermode USERMODE] [-latency LATENCY]
[-username USERNAME] [-password PASSWORD] [-debug DEBUG] [-help] [-version]
```

The command line options are described in Table 28.

Table 28: ncp_monitorconfig Command Line Options (1 of 2)

| Option | Description | Required/Optional |
|---|---|---|
| `-domain` *DOMAIN_NAME* | The name of the domain under which the MONITOR Configuration tool is running. | Required. |
| `-usermode` *USERMODE* | This option can be used to start the MONITOR Configuration tool in `highlevel` (simplified) or `lowlevel` (more complex) mode. If no value is specified, `highlevel` mode is used. | Optional. |
| `-latency <LATENCY>` | The maximum time in milliseconds (ms) that the component waits to connect to another Precision Server process via the messaging bus. You may need to specify a latency, using the `-latency` command line option, to increase the maximum time in milliseconds (ms) that the MONITOR Configuration tool waits to connect to CLASS via the messaging bus. As the AOC definitions can be very large, they may cause the MONITOR Configuration tool to timeout when downloading them from CLASS. | Optional. |
| `-username` *USERNAME* | The username you wish to use to log into the domain. | If the username is not supplied on the command line it must be entered at the login prompt instead. |
| `-password` *PASSWORD* | The password to access the MONITOR Configuration tool.<br><br>For security reasons you should use this option carefully, as other users may be able to see your password. Micromuse recommends that you enter your password when prompted rather than at the command line. | A password is required for the MONITOR Configuration tool, but should be specified at the login prompt rather than on the command line. |
| `-debug` *DEBUG* | The level of debugging output (1-4, where 4 represents the most detailed output). | Optional. |

Table 28: ncp_monitorconfig Command Line Options (2 of 2)

| Option | Description | Required/Optional |
|---|---|---|
| `-help` | Prints out a synopsis of all command line options for the component.<br><br>If specified, the component is not started even if –help is used in conjunction with other arguments. | Optional. |
| `-version` | Prints the version number of the component.<br><br>If specified, the component is not started even if –version is used in conjunction with other arguments. | Optional. |

# 3.3 Navigating the MONITOR Configuration Tool

This section describes the basic operation of the MONITOR Configuration tool.

## Logging into the MONITOR Configuration Tool

When the MONITOR Configuration tool starts, the *Login* window is displayed, as shown in Figure 3. Enter your username and password and select **Login**.

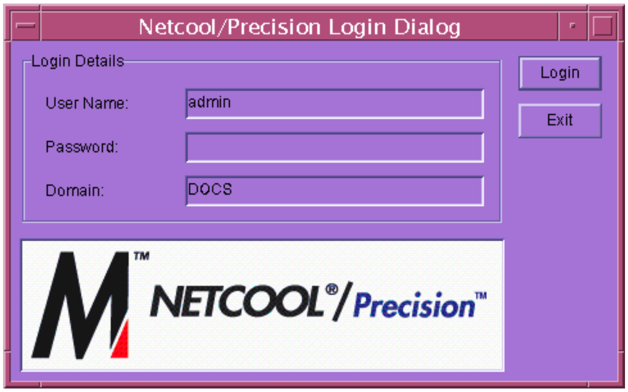


Figure 3: Login Window

In order to log into the MONITOR Configuration tool you must have permission to access the CLASS databases.

The default username and password combination is `admin` and no password. Micromuse recommends that the password for `admin` is changed during installation.

# The Main View

After you have successfully logged in, the MONITOR Configuration tool displays the main view, as shown in Figure 4.

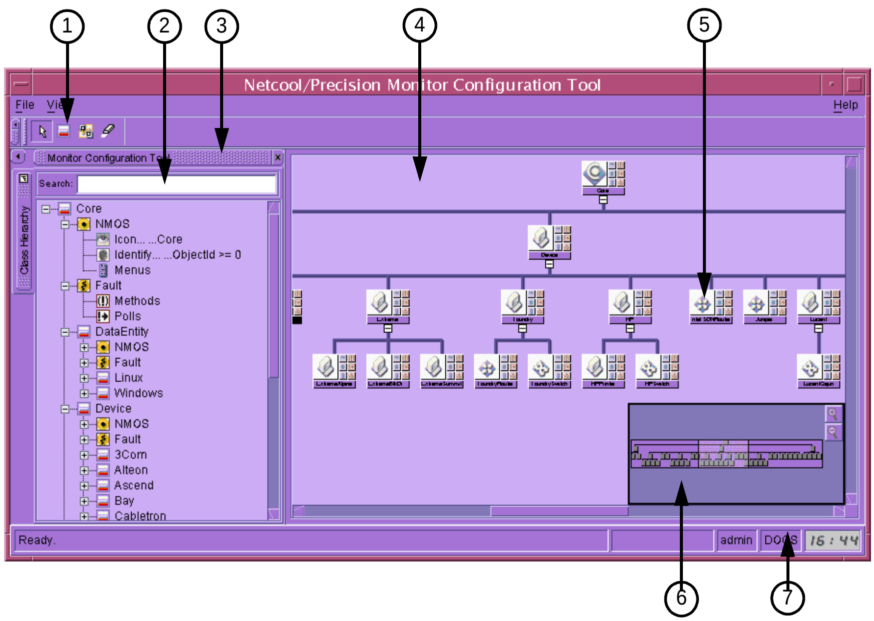


Figure 4: MONITOR Configuration Tool Main View

The areas within the main view are described in Table 29.

Table 29: Main View Descriptions (1 of 2)

| Item | Name | Description |
|---|---|---|
| 1 | Toolbar | Contains the MONITOR Configuration tool mode buttons. These set the mode of operation for the main view, as described in *MONITOR Configuration Tool Buttons* on page 60. |
| 2 | Search bar | Enter an AOC name (case sensitive) in the search box. If the class is found, it is shown and highlighted in the main work area |
| 3 | MONITOR Configuration Tool area | Displays the AOCs in a directory-like structure.<br><br>This area can be shown or hidden using the menu option **View→MONITOR Configuration Tool**. |
| 4 | Main work area | Displays a graphical representation of the AOC hierarchy. |

Table 29: Main View Descriptions (2 of 2)

| Item | Name | Description |
|---|---|---|
| 5 | Class Icon | The representation of a class in the main work area. The editor buttons displayed depend on the `usermode` command line option and the and the mode of operation set on the toolbar. |
| 6 | Panner tool | The panner tool contains a minimized overall representation of all the classes. Contained within the overall representation is a small box that represents the currently visible part of the class hierarchy.<br><br>To display the panner tool, select the menu option **View→Show Panner**. The position of the panner can be changed using the menu option **View→ Panner Location**. |
| 7 | Status bar | Displays the status of the current operation. It also displays the name of the domain and your username. |

## Using the Panner and Zoom Functions

The menu options available in the Main View are:

- View→Show Panner shows or hides the panner tool. The panner tool contains a minimized overall representation of all the classes. Contained within the overall representation is a small box that represents the currently visible part of the class hierarchy.

  You can drag the small box around to scroll around the classes. You can use the magnifying glass buttons on the panner tool to zoom in and out of the class hierarchy.

- View→Panner Location selects the position of the panner tool within the Window. Select one of the four corner options.
- View→Zoom to selects the level of zoom for the Main Work Area.

## MONITOR Configuration Tool Buttons

The MONITOR Configuration tool mode buttons are displayed in the toolbar, as shown in Figure 5.

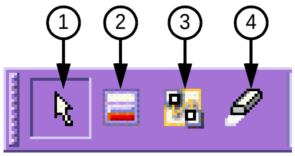


Figure 5: Toolbar

The toolbar buttons are:

1. Select Class tool: changes the cursor to *Select* mode, which is the default. In select mode you can click the dialog buttons attached to each class and click the class names to rename them. Select mode is the only mode in which the class dialog buttons are enabled.
2. Create New Class tool: changes the cursor to *Add* mode, in which you can click once on any class to add a child class below the existing class in the hierarchy.
3. Re-parent Class tool: changes the cursor to *Move* mode, in which you can move a class to a new location in the hierarchy. To move a class, click once on the class to be moved and click once on the new parent.
4. Delete Class tool: changes the cursor to *Delete* mode, in which you can delete a class by clicking once on the class to be deleted.

**Warning:** There is no undo function if you delete, move or add classes using the MONITOR Configuration tool.

## Class Icons

The class icons in the main work area contain a number of edit buttons, as shown in Figure 6.

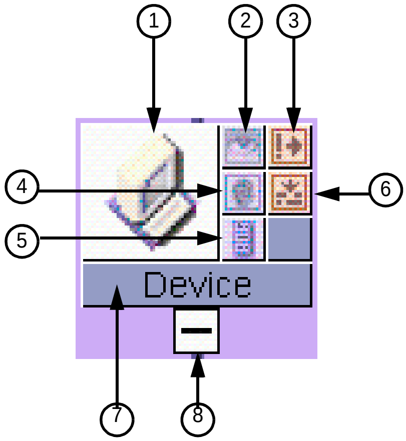


Figure 6: Class Icon

Each class icon edit button has one or more dependencies. Table 30 describes the items and buttons in the class icon and identifies the conditions that must be met for the item to appear. The name of the button is displayed as a tooltip when the cursor is placed over the button.

Table 30: Main View Descriptions

| Item | Name | Description |
|---|---|---|
| 1 | Visual icon | Icon representing the class type. |
| 2 | **Change Icon** button | Selects an icon to display in the Precision Desktop for the class, as described in *Changing the Visual Icon* on page 62. The Precision Server must be installed for this button to be displayed. |
| 3 | **Change Polls** button | Customizes the poll definitions, as described in *Editing Poll Definitions* on page 72. The monitoring and RCA components must be installed, and the command line option `usermode` must be set to `lowlevel` for this button to be displayed. |
| 4 | **Change Identity** button | Displays the Filter Builder window, as described in *Modifying the Instantiate Rule for a Class* on page 64. The Precision Server must be installed for this button to be displayed. |
| 5 | **Change Menus** button | Displays the Menu Builder window, as described in *Editing Menus in the Precision Desktop* on page 68. The Precision Server must be installed for this button to be displayed. |
| 6 | **Change Policies** button | Allows high-level configuration of monitoring policies, as described in *Managing Policies* on page 70. The monitoring and RCA components must be installed for this button to be displayed. |
| 7 | Class name | The name of the class.<br><br>You can rename the class by clicking the class name. Class names may not include spaces or non alphanumeric characters. |
| 8 | Show or Hide class | Shows or hides the child classes for the class. |

## Changing the Visual Icon

To change the icon that the Precision Desktop displays for the class:

1. Click the **Change Icon** button. The *Open* window is displayed.
2. Browse through the directory structure to the location of the icons. By default the images are stored in the `NCHOME/precision/images` directory.

   If you select an image in a different directory, the images is copied to the `NCHOME/precision/images` directory.

**Note:** `NCHOME` is the environment variable that contains the path to the Netcool Suite home directory. For information on how this environment variable varies with platform, see *Operating System Considerations* on page 9.

3. Select any image file with the `.xmp` file extension. The `.xpm` file is a UNIX-X11 PixMap image (256 color image for X Windows).

4. Click **Select** to select the image and close the *Open* window.

# 3.4 Modifying the Instantiate Rule for a Class

The *Filter Builder* window is used to define or modify the instantiate rules for a class. To open the *Filter Builder* window, click the **Change Identity** button on the class icon. This button is only displayed for each class icon when the Precision Server is installed.

Each time the network discovery locates a new device, the device is compared to the instantiate rules. The instantiate rule is a filter and any device that matches the filter is stored in the network topology database MODEL with a link to the class. This process is called *Instantiation*.

The *Filter Builder* window is shown in Figure 7.

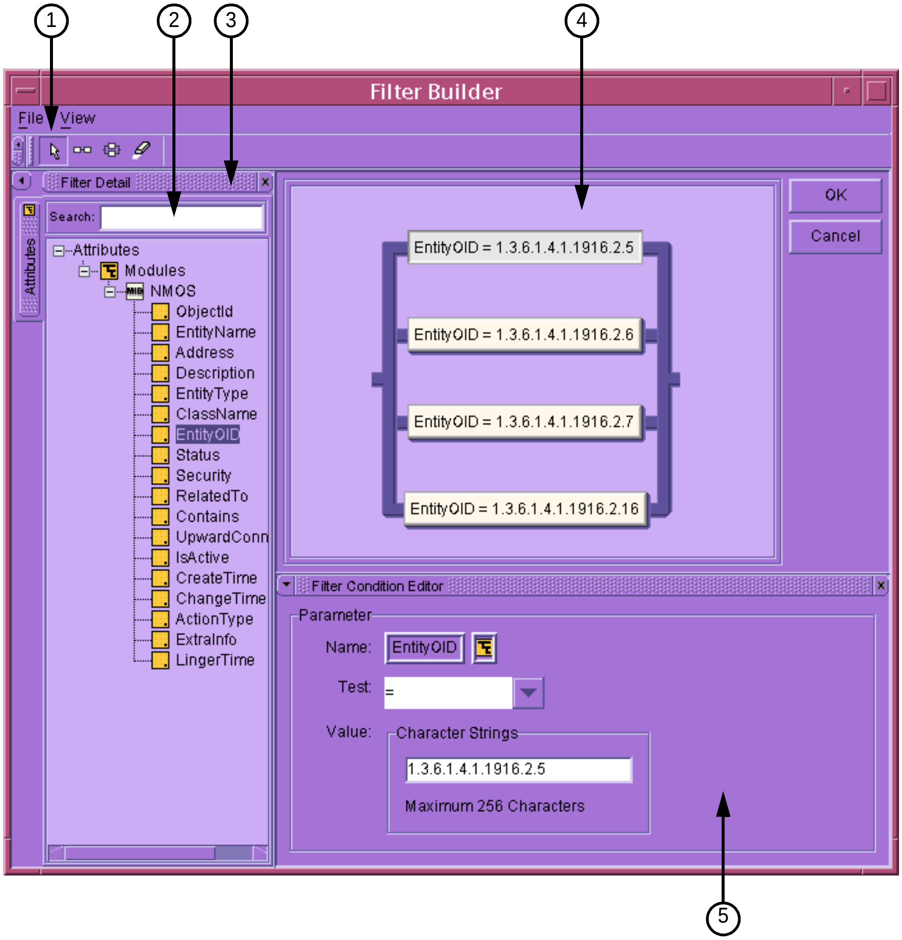


Figure 7: Filter Builder Window

The *Filter Builder* window is described in Table 31.

Table 31: Filter Builder Window Description

| Item | Component | Description |
|---|---|---|
| 1 | Toolbar | Selects the modes of operation. |
| 2 | Search Box | Used to search for an attribute in the hierarchy. If you enter partial attribute names, the first matching attribute is selected. Press Enter again to move to the next matching attribute. |
| 3 | Filter detail area | Displays all of the attributes that can be used in the instantiate rule. |
| 4 | Main work area | Displays the conditions of the instantiate rule and shows how the conditions are logically linked. |
| 5 | Filter condition editor area | This area is used for creating or editing the details of the conditions. The area displays the details for the condition selected in the main work area. |

## Filter Builder Modes of Operation

The *Filter Builder* window modes of operation are selected using the toolbar shown in Figure 8.

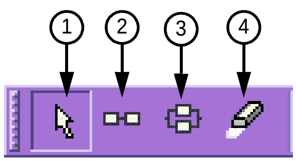


Figure 8: Filter Builder Toolbar

The toolbar buttons and the modes of operation are described in Table 32

Table 32: Filter Builder Toolbar Descriptions

| Item | Button | Description |
|---|---|---|
| 1 | **Select** | When enabled, you can edit the attribute test for the selected condition in the filter condition editor area. |
| 2 | **AND** | When enabled, you can create an AND condition in the main work area. Return to Select mode to edit the condition in the filter condition editor area. |
| 3 | **OR** | When enabled, you can create an OR condition in the main work area. Return to Select mode to edit the condition in the filter condition editor area. |
| 4 | **Delete** | When enabled, you can delete a condition in the main work area by clicking the condition. This operation cannot be undone. |

## Constructing Complex Rules

The logical linking of conditions to create complex instantiate rules is graphically displayed in the main work area.

To construct complex instantiate rules, join conditions using the AND and OR operators. A conditions joined using AND is shown adjacent to the others. Conditions joined using OR are shown parallel to other conditions. You can create as many nested AND and OR conditions as necessary.

To add a new condition:

1. Move the cursor within the main work area to select one or more pre-existing conditions. These conditions are bounded by a rectangle.
2. If OR mode is selected, clicking the mouse button encloses all selected conditions within a rectangle. The new condition is added to the flow in parallel to these conditions.
3. If AND mode is selected, clicking the mouse button encloses all selected conditions within a rectangle if required. The new condition is added to the flow adjacent to these conditions.

Any given instantiate rule is considered to have been passed if there is a path that can be taken from the left hand side of the display to the right that satisfies each condition in between.

## The Filter Condition Editor

When a condition is selected in the main work area, the details of the condition are displayed, and can be edited, in the filter condition editor area.

The fields in the filter condition editor area are described in Table 33.

Table 33: Filter Condition Editor Area Fields

| Field | Description |
|---|---|
| **Name** | The name of the device attribute to be tested. |
| **Test** | Select an operator from this drop down list. Possible values include:<br>• `=`<br>• `>`<br>• `like`<br>• `not like` |
| **Value** | The value for the condition. See note below this table. |

**Note:** The acceptable values for this attribute (such as, character strings or integers) are indicated above the Value field. Any other requirement (for example, a maximum number of characters) is shown below the Value field. You must use the backslash character (\) to escape the dot when entering values such as an Entity OID.

# 3.5 Editing Menus in the Precision Desktop

The *Menu Builder* window is used to construct the menu items that appear in the Precision Desktop. To open the *Menu Builder* window, click the **Change Menus** button on the class icon.

You can define a context-sensitive menu option that appears in the Precision Desktop when there is an event associated with a device that has been instantiated to this class. The *Menu Builder* window is shown in Figure 9.

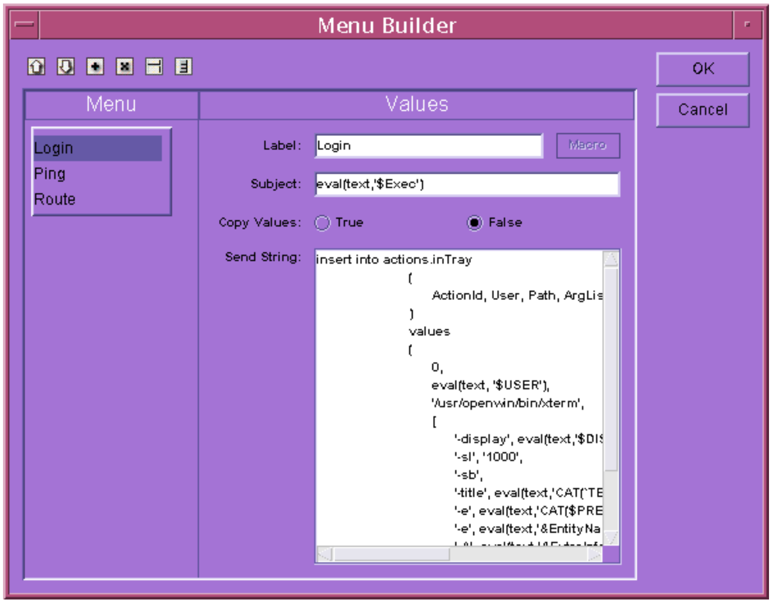


Figure 9: Menu Builder Window

In order to add a menu, click the + button and type a name in the **Label** field. As you type the name, it automatically appears in the **Menu** column. This is how it ultimately appears in the Precision Desktop. The fields for the menu are described in Table 34.

Table 34: Menu Builder Field Descriptions (1 of 2)

| Field | Description |
|---|---|
| **Label** | The name of the option that is to be listed on the Precision Desktop menu. |
| **Subject** | identifies the service or Precision Server component with which the OQL command string is associated. Typically specifies an **eval** statement such as `eval(text, '$Exec')`. |

Table 34: Menu Builder Field Descriptions (2 of 2)

| Field | Description |
|---|---|
| **Copy Values** | Select the **true** or **false** radio button. If true, the values in the fields of the triggering event overwrite the values in the database referenced in the send string field. If false, the values in the target database are those specified by the send string attribute. This attribute is only used when using a run directive to insert an event into a database that has exactly the same fields, such as the AMOS Events database. |
| **Send String** | The OQL insert to be sent to the component specified as the subject. |

When you have finished creating your menu, click OK to save them and exit the *Menu Builder* window.

You can create any number of menus. You can also use the up and down arrows to move the position of a an item in the Precision Desktop menu and add horizontal dividers and submenus that can contain as many menu methods as you wish.

The *Menu Builder* window, shown in Figure 9, contains the following menus defined for the current class:

- Login: allows you to telnet into a device associated with this AOC where an event or alert has been generated.
- Ping: allows you to run an extra ping on the device.
- Route: runs a trace route application on the device.

# 3.6 Managing Policies

Network monitoring is controlled by enabling and disabling polling policies in the *Policy Editor* window.

To open the *Policy Editor* window, click the **Change Policies** button on the class icon. This button is only displayed when the monitoring and RCA components of Netcool/Precision IP are installed.

The *Policy Editor* window, shown in Figure 10 is used to configure and implement the monitoring policies supplied with Netcool/Precision IP. The monitoring policies contain both polling and event correlation functionality.

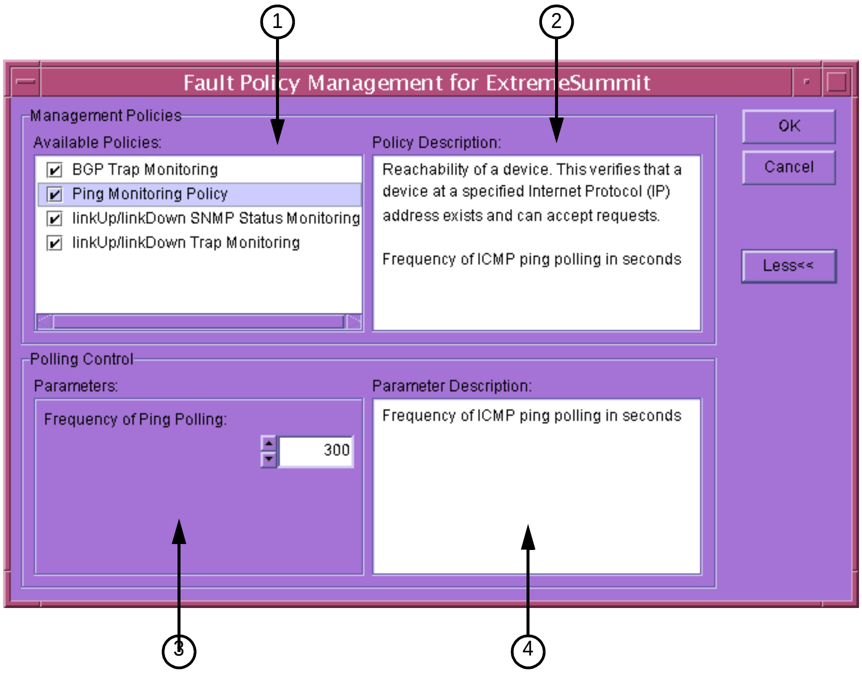


Figure 10: Policy Editor Window

Figure 10 shows the expanded *Policy Editor* window. The expanded section is displayed by selecting the MORE>> button. When expended this button changes to LESS<<.

The components of the *Policy Editor* window are described in Table 35.

Table 35: Policy Editor Window Component Descriptions

| Item | Component | Description |
|---|---|---|
| 1 | Available Policies | Displays the available monitoring policies. |
| 2 | Policy Description area | Displays the description of the selected policy. |
| 3 | Parameters area | Displays the parameters for the selected polling policy. This area is available when the *Policy Editor* window is expanded. |
| 4 | Parameter Description area | Displays information about the parameter you are currently changing. This area is available when the *Policy Editor* window is expanded. |

**Warning:** The term `Frequency`, when displayed in the *Policy Editor* window, always refers to a time period in seconds.

## Selecting and Configuring Polling Policies

To enable a polling policy, select the checkbox to the left of the policy. To disable a policy, deselect the checkbox.

To configure a polling policy, select the policy in the Available Policies area and expand the *Policy Editor* window. The Parameters area displays the parameters for the selected policy.

Set the parameter to the required value. For example, you may want to change the polling frequency parameter.

To save changes and implement the policies, click the OK button. When you close the *Policy Editor* window, the MONITOR Configuration tool sends an update to CLASS.

The following error message is received if CLASS does not respond to the update:

```
Process ncp_class did not respond before the -latency timeout delay. Your changes may
not have been applied.
```

If CLASS is running, you should restart the MONITOR Configuration tool with the `-latency` command line option set to a larger value. The default latency value is `60000` ms).

# 3.7 Editing Poll Definitions

The *Poll Editor* window, shown in Figure 11, allows you to customize the poll definitions or create new poll definitions. To open the *Poll Editor* window, click the **Change Polls** button of the class you want to edit.

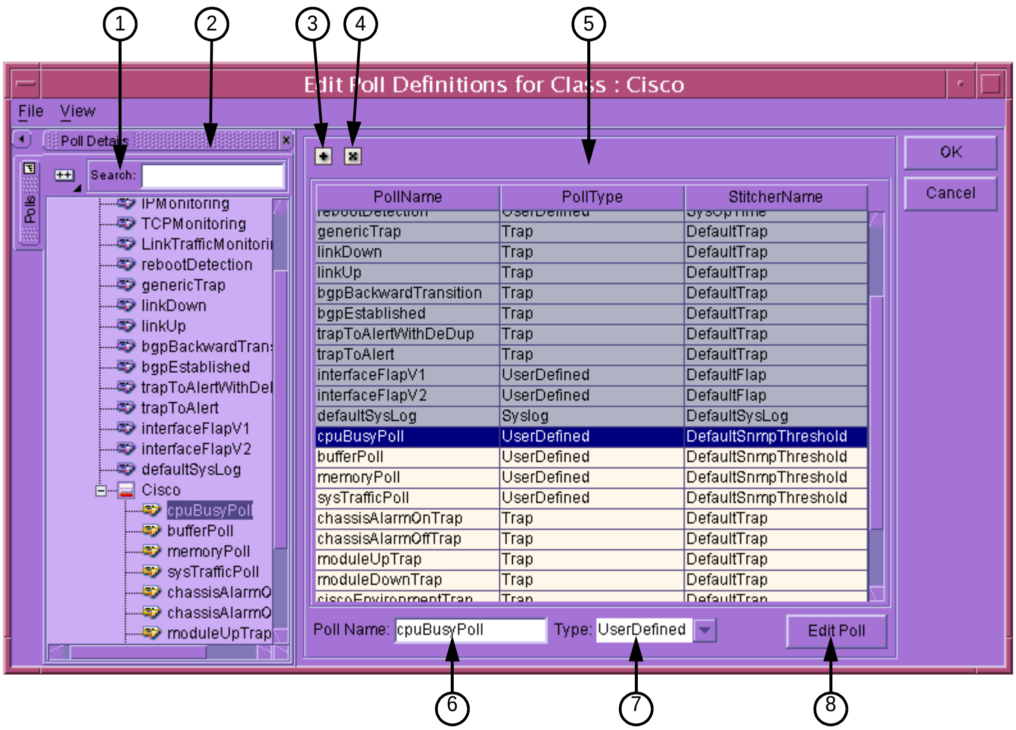


Figure 11: The Poll editor main window

The areas within the *Poll Editor* window are described in Table 36.

Table 36: Poll Editor Window Description (1 of 2)

| Item | Area | Description |
|---|---|---|
| 1 | Search box | Used to search for an poll definition in the hierarchy. If you enter partial poll definition names, the first matching entry is selected. Press Enter again to move to the next matching poll definition. |
| 2 | Poll console | Used for navigating through the hierarchy of AOCs to examine the poll definitions. You can navigate upwards through the hierarchy, but not downwards from the class you are in. |

Table 36: Poll Editor Window Description (2 of 2)

| Item | Area | Description |
|---|---|---|
| 3 | **Add Poll** button | Adds a new poll definition.<br><br>Clicking this button creates a new definition with a name based on the current class, and the values are set to `=""` or to the default values. Customize this new definition to create the required poll definition. |
| 4 | **Delete Poll** button | Deletes the selected poll definition. |
| 5 | Main window | Displays a list of poll definitions for the class.<br><br>Poll definitions are shown in different colors according to where they are defined. Inherited definitions are purple, locally defined definitions are gold, and inherited definitions which have been locally overridden are shown in a light purple box. |
| 6 | **Poll Name** field | Changes the poll definition name.<br><br>Note that you cannot rename an inherited poll definition. |
| 7 | **Poll Type** field | Change the poll type to change several linked attributes at the same time.<br><br>For example, changing the poll type from `Trap` to `Ping`, automatically changes the stitcher to `DefaultPing.stch` and the agent executable to `ncp_m_timedstitcher`.<br><br>Note that you cannot rename an inherited poll definition. |
| 8 | **Edit Poll** button | Opens the *Values Editor* window for the selected poll definition. |

**Note:** Editing a definition inherited from a class above the current class only overrides that definition in the class you are editing. It does *not* change that definition in the class in which it was defined.

## Editing a Poll Definition

The poll definition name and the poll type are edited using the **Poll Name** and **Poll Type** buttons in the *Poll Editor* window, as described in Table 36. All other attributes are edited using the *Values Editor* window.

To display the *Values Editor* window, click the **Edit Poll** button in the *Poll Editor* window. In the *Values Editor* window the **PollName** and **PollType** fields are always disabled. If the poll type is not set to **UserDefined**, the **AgentName** and **StitcherName** fields are also greyed out. All other compulsory attributes are present in the *Values Editor* window.

The *Values Editor* window is shown in Figure 12.

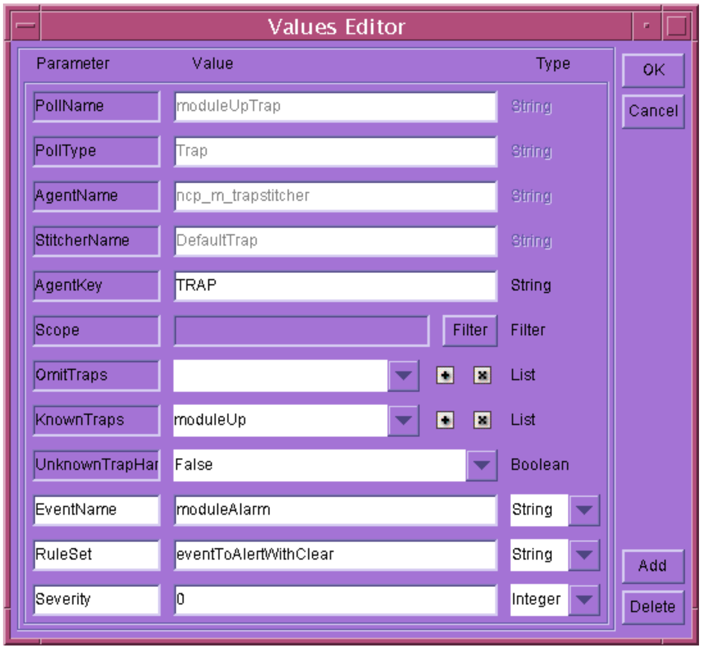


Figure 12: The Values Editor

To change the parameters of the attributes, use the drop down lists and text boxes.

To add an attribute, click the **Add** button. The new field can refer to any of the attributes of a poll definition.

To delete an optional attribute, select the attribute by clicking in the parameter field then click the **Delete** button.

To add an item to a list, type the item you wish to add in the text box and click the + button.

To delete items from a list, select an item from the list and click the X button.

All the attributes of the poll definition are edited using the procedure above, with the exception of fields containing a filter. Editing the filter for a field is describe in *Editing Attribute Containing Filters* on page 85.

## Attributes of the Poll Definitions

The attributes available for constructing poll definitions are described in Table 37. All attributes are compulsory unless otherwise stated. If you write your own poll definitions, you must make sure that you use the appropriate attributes for the stitcher that you intend to use. See Chapter 4: *Stitchers Used for Polling* on page 89 for a list of the stitchers, including the poll definition attributes that each stitcher requires.

Table 37: Poll Definition Attribute Descriptions (1 of 2)

| Attribute | Type | Description |
|---|---|---|
| `PollName` | String | Defines the name of the poll definition. It must be unique within the specified AOC. For example:<br>`PollName = "defaultPing",` |
| `PollStatus` | Boolean | Used by the Policy Editor to turn a particular poll definition on (`=1`) or off (`=0`). If any other value is specified, `PollStatus` defaults to `1`. For example:<br>`PollStatus = 1,` |
| `AgentName` | String | Defines the agent executable that is used. For Ping and SNMP polls this is `ncp_m_timedstitcher`, for syslog polling it is `ncp_m_syslogstitcher`, for trap monitoring it is `ncp_m_trapstitcher`, and for Visionary integration it is ncp_m_visionary. For example:<br>`AgentName = "ncp_m_timedstitcher",` |
| `AgentKey` | String | Allows multiple and distributed polling agents to be run (on separate machines if required).<br>By default, the timed agent only polls elements with valid IP addresses. Adding the plus symbol (+) enables the timed agent to poll non-IP devices, such as, local interfaces which can be interrogated through their owning device.<br>For example:<br>`AgentKey = "PING",`<br>`AgentKey = "LINK+",`<br>`AgentKey = "NCV",` |
| `StitcherName` | String | The name of the stitcher file that the agent executable runs. Text-based stitchers are stored in `NCHOME/precision/monitor/stitchers`, and precompiled stitchers are stored in `NCHOME/precision/monitor/lib`.<br>`StitcherName` should not include filename extensions. Additionally, when referring to precompiled stitchers, the `libPollerStitcher` prefix should be omitted.<br>For example:<br>`StitcherName = "DefaultPing",` |

Table 37: Poll Definition Attribute Descriptions (2 of 2)

| Attribute | Type | Description |
|---|---|---|
| `Frequency` | Integer | Specifies the interval at which the agent executable runs the stitcher, in seconds. Only applies only to the timed polling agent. For example:<br>`Frequency = 300,` |
| `Scope` | String | Specifies the types of devices the poll is run on. The scope attribute is applied to the `master.entityByName` database in MODEL. If the scope condition evaluates to `false` the poll is not run on that device.<br>If this attribute is not included in the poll definition, the default option is to pass all types of device.<br>This attribute only applies to stitchers that are executed by `ncp_m_timedstitcher` and ncp_m_visionary.<br>The following example, restricts polling to interfaces and main nodes without interfaces. Since the IP address of a main node is the same as the IP address of one of its interfaces, this ensures that the same address is not pinged twice.<br>`Scope = "((Contains is NULL AND EntityType=1)`<br>`          OR EntityType=2)",` |
| `AgentControl` | String | This optional attribute applies to non-timed polling agents only. It is an object that contains a list of sub-attributes, as shown below:<br>`AgentControl = {`<br>`<sub-attribute1>,`<br>`<sub-attribute2>,`<br>`<sub-attribute..>,`<br>`<sub-attributeN>,`<br>`}`<br>The available sub-attributes are described in Table 38. |
| `StitcherInfo` | String | The mandatory attribute, `StitcherInfo`, contains a list of optional sub-attributes. The sub-attributes of `StitcherInfo` are configurable and can contain any name-value pair. You can also leave `StitcherInfo` unassigned.<br>The data types of the sub-attributes of `StitcherInfo` do not have to be defined in the poll definitions. The data types can be specified in the stitcher file. A number of common sub-attributes used by the stitchers are described in Table 39. |

Table 38 describes the sub-attributes of the `AgentControl` attribute.

Table 38: Sub-Attributes of the AgentControl Attribute (1 of 2)

| Sub-Attribute | Type | Description |
|---|---|---|
| `KnownTraps` | String | A list of traps that the trap agent considers for processing. Names of traps can be specified, or you can use the following special values:<br>• `ALL` - the agent processes all traps it receives, with the exception of those named in the `OmitTraps` section.<br>• `UNHANDLED` - the agent uses this poll definition to process any traps which are not in the `KnownTraps` section of any poll definitions in scope, and which are not named in the `OmitTraps` section.<br>This sub-attribute only applies to the trap polling agent. For example:<br>`KnownTraps = [ "linkDown","linkUp" ],` |
| `OmitTraps` | String | Lists the traps that the trap agent must not consider for processing. If a trap named in `OmitTraps` is received, the trap agent does not start the stitcher and discards the trap. The following example lists two traps to omit:<br>`OmitTraps = [ "coldStart","warmStart" ],`<br>This sub-attribute only applies to the trap polling agent. |
| `UnknownTrap Handling` | string | A boolean expression that defines whether a trap which is not defined in any of the MIBs available to the trap agent should be processed in this poll definition. The MIB definitions are stored in `NCHOME/precision/mibs`.<br>A trap which is not defined in a MIB cannot have its enterprise OID and trap number resolved to an equivalent textual identifier.<br>This sub-attribute only applies to the trap polling agent. For example:<br>`UnknownTrapHandling = False,` |
| `FileNames` | String | Specifies which system files are parsed for messages. The following example specifies that the `/var/adm/messages` file should be parsed:<br>`FileNames = ["/var/adm/messages"],`<br>This sub-attribute only applies to the syslog polling agent. |
| `FieldNames` | String | Specifies which fields of the system message are extracted and can therefore be used by the stitcher.<br>This sub-attribute only applies to the syslog polling agent. For example:<br>`FieldNames = ["Date","Time","NodeName","Service","Message"],` |

Table 38: Sub-Attributes of the AgentControl Attribute (2 of 2)

| Sub-Attribute | Type | Description |
|---|---|---|
| `Reread` | Boolean | Defines the location to start reading a file from. Possible values are:<br>• `0` - False - specifies that the system file is parsed from the place it was last read.<br>• `1` - True - specifies that the file is parsed again from the start.<br>This sub-attribute only applies to the syslog polling agent. For example:<br>`Reread = 0,` |
| `RegExps` | String | A list of regular expressions, which has a one-to-one correspondence to the field names defined in the `FieldName` attribute. A typical assignment is given below:<br>`RegExps = [`<br>`     "[A-Z][a-z][a-z]  [0-9][0-9]*",`<br>`     "[0-9][0-9]*:[0-9][0-9]*:[0-9][0-9]",`<br>`     "[a-zA-Z0-9\.]*",`<br>`     "[A-Za-z0-9]*\[[0-9]*\]:",`<br>`     ".*:"`<br>`     ],`<br>This sub-attribute only applies to the syslog polling agent. |
| `Delimiter` | String | Used in parsing the regular expressions. A delimiter defines the separation of the regular expressions. The following example specifies that a whitespace is considered the break between two fields in a syslog message:<br>`Delimiter = " ",`<br>You can also specify a comma-separated list of delimiters. In the following example, any of the delimiters in the list is used as the separation of the regular expressions:<br>`Delimiter = ["TD: ", "PP: "],`<br>This sub-attribute only applies to the syslog polling agent. |
| `Mappings` | String | Specifies which of the fields listed in the `FieldNames` attribute identifies the originator of the system message. For example:<br>`Mappings = ["NodeName"],`<br>This sub-attribute only applies to the syslog polling agent. |

Table 39 describes a number of the sub-attributes of the `StitcherInfo` attribute.

Table 39: Sub-Attributes of the StitcherInfo Attribute (1 of 5)

| Sub-Attribute | Type | Description |
|---|---|---|
| `CorrelateBy` | String | This attribute is used in calculating whether a series of `linkUp` and `linkDown` traps constitutes a *flapping* device. If `CorrelateBy` is not assigned, incoming traps are only correlated by IP address.<br><br>A certain number of traps from one device in a certain time period constitutes flapping. The following example ensures the traps are additionally correlated by interface number:<br><br>`CorrelateBy = [‘ifIndex’]` |
| `Description` | String | The description that is included in the event generated by the stitcher. For example:<br><br>`Description = "My Threshold event",`<br><br>When used in poll definitions using the `AocDefinedThreshold` stitcher, the event description can include SNMP data retrieved from the device. For example:<br><br>`Description = 'Problem with interface eval(text,"&SNMP.VALUE.ifName").',`<br><br>For more information on using the eval language with the `AocDefinedThreshold` stitcher, see *Extended Eval Language Support for SNMP Threshold Polling* on page 92. |
| `DsmAddress` | String | The IP address of the Netcool/Visionary Distributed Status Monitor (DSM). For example:<br><br>`DsmAddress="127.0.0.1",`<br><br>For further information about DSMs, see the Netcool/Visionary 2.7 documentation set. |
| `DsmName` | String | The name of the Netcool/Visionary DSM. For example:<br><br>`DsmName="dsm1",` |
| `EventName` | String | The name of the event generated by the stitcher. For example:<br><br>`EventName = 'threshBreach',` |
| `Retries` | Integer | The number of times that a stitcher attempts to re-poll a device if a poll fails. For example:<br><br>`Retries = 3,` |
| `RuleSet` | String | Defines the ruleset that is used to correlate this event. For example:<br><br>`RuleSet = 'eventToAlertWith24HourClear',` |

Table 39: Sub-Attributes of the StitcherInfo Attribute (2 of 5)

| Sub-Attribute | Type | Description |
|---|---|---|
| `Severity` | Integer | Defines the severity of the event generated by the stitcher. For example:<br>`Severity = 3,` |
| `TimeOut` | Integer | Used to determine how long a stitcher waits for a response from a polled device before considering the request to have timed out. For example, in `libPollerStitcherDefaultPing.so` the following value is measured in milliseconds:<br>`TimeOut = 5000,` |
| `MibVariable` | String | `MibVariable` specifies the non-repeating MIB variable to be polled for. For example:<br>`MibVariable = "freeMem",`<br>Attributes `MibVariable`, `Comparison` and `Threshold` must be used together to form an SNMP threshold condition. The format is:<br>`MibVariable =` *`Value`*<br>`Comparison =` *`Value`*<br>`Threshold =` *`Value`* |
| `Comparison` | String | This attribute must follow `MibVariable`. Possible values are:<br>• `">"` - greater than<br>• `">="` - greater than or equal to<br>• `"<="` - less than or equal to<br>• `"=="` - equal to<br>• `"<>"` - not equal to<br>For example:<br>`Comparison = ">"` |
| `Threshold` | String | This sub-attribute of `StitcherInfo` is used in poll definitions using the `DefaultSnmpThreshold` stitcher, as well as in poll definitions using the `AocDefinedThreshold` stitcher.<br>When used in poll definitions using the `DefaultSnmpThreshold` stitcher, this attribute must follow `Comparison`.<br>For example:<br>`Threshold= 100000,`<br>When used in poll definitions using the `AocDefinedThreshold` stitcher, `Threshold` defines the condition that, if breached, causes an event to be raised. For more information on this attribute, see *Defining a Threshold Condition for SNMP Polling* on page 83. |

Table 39: Sub-Attributes of the StitcherInfo Attribute (3 of 5)

| Sub-Attribute | Type | Description |
|---|---|---|
| RestoreEvent | Boolean | Defines whether a restore event is generated if the result of the SNMP conditional test, defined by MibVariable, Comparison and Threshold, falls below the threshold condition, having previously exceeded it. Possible values are:<br>• 1 - True - a restore event is generated.<br>• 0 - False (Default) - no restore event is generated.<br>For example:<br>RestoreEvent = 0, |
| PollFailEvent | Boolean | Defines whether a poll fail event is generated if no result is returned for an SNMP query of a device. Possible values are:<br>• 1 - True - no event is generated.<br>• 0 - False (Default) an event is generated with a severity of 3.<br>For example:<br>PollFailEvent=0, |
| ThresHold | String | Specifies the value of the Service field defined in the FieldNames attribute of AgentControl which the stitcher is looking for in syslog polling. The following example tells the stitcher to only process a message if the Service field of the message contains the text unix:<br>ThresHold = 'unix:',<br>ThresHold is used in syslog monitoring. |
| Varbinds | List | A list of the SNMP variables to be initially gathered by the poll. For a table poll all the entries should have the same index, ideally being from the same table. The SNMP variables retrieved will be included in the event sent by the stitcher. The variables are also available for use in other attributes of the poll definition by using the eval language.<br>This sub-attribute of StitcherInfo is only used in poll definitions using the AocDefinedThreshold stitcher. |
| TablePoll | Boolean | A boolean flag indicating whether the poll is to gather single instances of the specified SNMP variable or walk a whole table, where<br>0=Single instance<br>1=Table poll<br>For example:<br>TablePoll = 1,<br>This sub-attribute of StitcherInfo is only used in poll definitions using the AocDefinedThreshold stitcher. |

Table 39: Sub-Attributes of the StitcherInfo Attribute (4 of 5)

| Sub-Attribute | Type | Description |
|---|---|---|
| `ClearThreshold` | String | This sub-attribute of `StitcherInfo` is only used in poll definitions using the `AocDefinedThreshold` stitcher.<br><br>Once the `Threshold` has been broken for a particular entity, subsequent polls evaluate the `ClearThreshold`, if present. If the `ClearThreshold` is breached (evaluates true), then a Clear event will be sent for the entity.<br><br>If no `ClearThreshold` is specified, a Clear event will be sent as soon as the `Threshold` evaluates false for a subsequent poll on the same entity.<br><br>This attribute uses the same syntax as the `Threshold` attribute. For more information on the `Threshold` attribute, see *Defining a Threshold Condition for SNMP Polling* on page 83. |
| `ClearDescription` | String | This sub-attribute of `StitcherInfo` is only used in poll definitions using the `AocDefinedThreshold` stitcher.<br><br>The description that is included in the Clear event generated by the stitcher. For example:<br><br>`ClearDescription = 'Problem was with interface eval(text,"&SNMP.VALUE.ifNAME").',` |

Table 39: Sub-Attributes of the StitcherInfo Attribute (5 of 5)

| Sub-Attribute | Type | Description |
|---|---|---|
| `AdditionalVarbinds` | List | This sub-attribute of `StitcherInfo` is only used in poll definitions using the `AocDefinedThreshold` stitcher.<br><br>Defines further SNMP variables to be retrieved in the event of the `Threshold` being breached. The definition must include the name of the variable to be retrieved. Optionally, the index of the previously retrieved SNMP variable to be used to define the specific piece of data required can be included. These further SNMP variables, once retrieved, are available for use in other attributes of the poll definition by using the eval language.<br><br>For more information on using the eval language with the `AocDefinedThreshold` stitcher, see *Extended Eval Language Support for SNMP Threshold Polling* on page 92. |
| `IndexExtractions` | List | This sub-attribute of `StitcherInfo` is only used in poll definitions using the `AocDefinedThreshold` stitcher.<br><br>Assigns specified bits from the index of an SNMP request to a variable. `IndexExtractions` contains the following variables:<br><br>• `Name`. The name of the variable to which the bits are assigned.<br>• `Varbind`. The SNMP varbind from the index of which the bits are extracted.<br>• `IndexDigit`. Where an SNMP request index is multiple digits long, `IndexDigit` specifies which digit the bits are extracted from.<br>• `Bits`. Specifies either a single bit or a bit range.<br><br>The following example creates a variable called `RouteInfo`, and assigns it route information from bits five to eight of the SNMP variable `ipRouteNextHop`.<br><br>`IndexExtractions = [{ Name = "RouteInfo" ,  Varbind = 'ipRouteNextHop', IndexDigit = 1, Bits = '5-8' }]` |

## Defining a Threshold Condition for SNMP Polling

When used with the `AocDefinedThreshold` stitcher, the `Threshold` field defines a condition that, if it evaluates true, results in an event being sent. The `Threshold` field accepts simple arithmetic rules, boolean operators, and IP to Long datatype conversion. Table 40 shows all the available operators and gives examples of their use.

Table 40: Operators Available in the Threshold Field (1 of 2)

| Operator | Example |
|---|---|
| Plus | `(1 + 2)` |
| Minus | `(4 - 2)` |

Table 40: Operators Available in the Threshold Field (2 of 2)

| Operator | Example |
|---|---|
| Multiplication | `(5 * 3)` |
| Division | `( 10 / 2 )` |
| Modulus | `( 8 % 3 )` |
| Power | `(10 POW 3)` |
| Log | `(Ln 5)` |
| IP to Long datatype conversion | `( IpToLong("1.2.3.4"))` |
| Bitwise AND | `(5 & 3)`<br>Note that bitwise operations can only be applied to integer values. |
| Bitwise | `(5 \| 3)` |
| Bitwise Exclusive OR | `(5 ^ 3)` |
| Boolean OR | `((eval(int, '&SNMP.VALUE.ifSpeed') > 10000) OR (eval(int, '&SNMP.VALUE.ifSpeed') < 100))` |
| Boolean AND | `((eval(int, '&SNMP.VALUE.ifSpeed') > 10000) AND (eval(int, '&SNMP.VALUE.ifOperStatus') !=2 ))` |
| Boolean NOT | `(NOT((eval(int,'&SNMP.VALUE.ifOperStatus') = 1))` |
| Equal | `( eval(int, '&SNMP.VALUE.ifOperStatus') = 1 )` |
| Not equal | `( eval(int, '&SNMP.VALUE.ifOperStatus') != 1 )` |
| Less than | `( eval(int, '&SNMP.VALUE.ifSpeed') < 100 )` |
| Greater than | `( eval(int, '&SNMP.VALUE.ifSpeed') >100 )` |
| Less than or equal | `( eval(int, '&SNMP.VALUE.ifSpeed') <= 100 )` |
| Greater than or equal | `( eval(int, '&SNMP.VALUE.ifSpeed') >= 100 )` |
| Like | `( eval(text, '&RECORD.ExtraInfo->m_IfDescr') LIKE 'Gigabit.*' )` |
| Not Like | `( eval(text, '&RECORD.ExtraInfo->m_IfDescr') NOT LIKE 'Loopback.*' )` |

## Editing Attribute Containing Filters

To add a filter to an attribute, click the **Filter** button. The *Filter Builder* window is displayed. The *Filter Builder* window for the **Scope** field is shown in Figure 13.

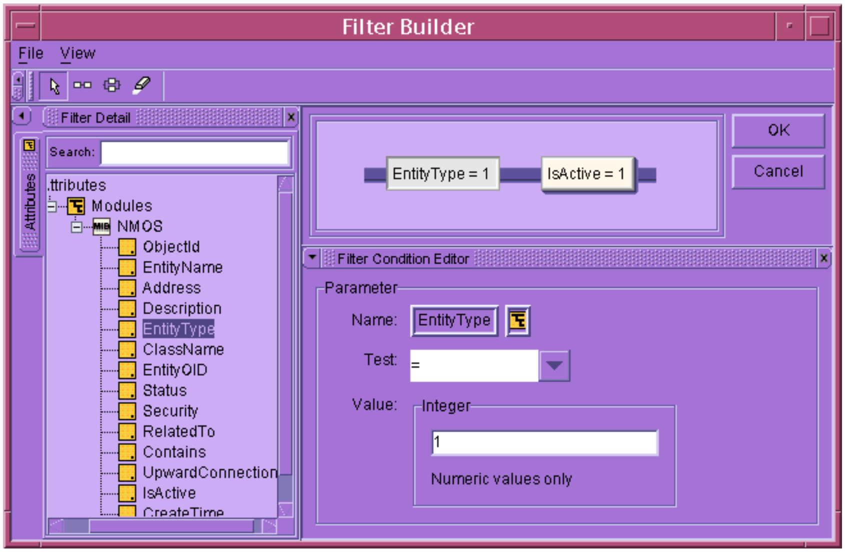


Figure 13: The Filter Builder as it Appears when Editing the Scope Attribute

If you are editing an attribute other than Scope the *Filter Builder* window has the same appearance, except that it does not have the filter detail area on the left. An example *Filter Builder* window for the Threshold attribute is shown in Figure 14.

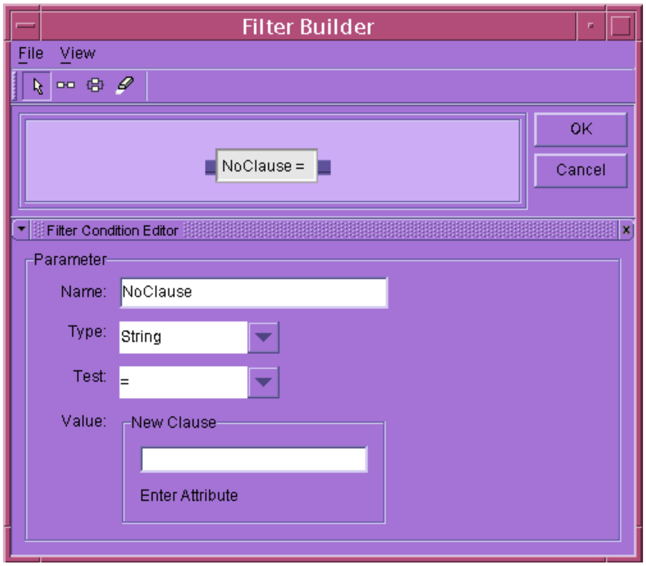


Figure 14: The Filter Builder Without the Filter Detail Console

The operation of the *Filter Builder* window for poll definitions is the same as the *Filter Builder* window used to modify the instantiate rules for a class. For a description of the operation of the *Filter Builder* window, see *Modifying the Instantiate Rule for a Class* on page 64.

## 3.8 Planning your Classes

Although scalability and extensibility are fundamental principles of the AOCs, it is still important that you give some thought to the structure and hierarchy of your classes before you begin to create additional AOCs. Some key points to consider when planning your classes are:

- A child class automatically inherits the instantiate rules of its parent class when it is created by the MONITOR Configuration tool. If the instantiate rules for that parent class change, however, the children are not dynamically updated (because inherited instantiation rules can be locally overridden). Therefore, if you want instantiation rules to be inherited beneath a given level in the hierarchy you must attribute them to the parent class before you create any child classes.
- Classes do not inherit the instantiate rules of their new parents if they are moved to a different location in the hierarchy. It is therefore important to plan your class structure before you begin to define your classes.
- Should a device match the instantiate rule of more than one AOC, it instantiates to the class in the hierarchy that is deepest and furthest left. The position of a given AOC in the hierarchy is determined alphabetically. For example, `Bay` comes before `Cisco`.

There is a risk that more than one instantiate rule at the same level can match. Where possible you should try to use instantiate rules that are mutually exclusive at any given level in your AOC hierarchy to avoid this problem.

# Chapter 4: Stitchers Used for Polling

This chapter describes the stitcher rules unique to the polling agents, complete with an explanation of the required input and output for each. It also gives an explanation of scope within monitoring stitchers and deconstructs an example stitcher file to show the relationship between the poll definitions, stitcher rules and scope.

This appendix contains the following sections:

## 4.1 Introduction to Stitchers

Stitchers are versatile programs used throughout Netcool/Precision IP to retrieve and manipulate data. A large number of stitchers are used in the discovery process, and stitchers are also vital to the polling process. This section describes the monitoring stitchers used in the polling process, and all references here to stitchers should be understood as referring to monitoring stitchers. The discovery stitchers are described in the *Netcool/Precision IP Discovery Configuration Guide*.

Stitchers can be precompiled or text-based. Precompiled stitchers cannot be modified by the user, however, their functioning can be controlled by defining their input through the poll definitions. Precompiled stitchers are stored in `NCHOME/precision/monitor/lib` and have the filename suffix `.so`.

**Note:** `NCHOME` is the environment variable that contains the path to the Netcool Suite home directory. For information on how this environment variable varies with platform, see *Operating System Considerations* on page 9.

Text-based stitchers can be edited. They are located in the `NCHOME/precision/monitor/stitchers` directory and have the filename suffix `.stch`. Precompiled stitchers may have alternative text-based versions. These are contained in `NCHOME/precision/monitor/stitchers` and have the filename extension `.txt`. If you want an alternative text-based stitcher to be used instead of a precompiled stitcher, you need to change the filename extension of the text-based stitcher from `.txt` to `.stch`, and change the name to match the name specified in the poll definition.

For example, if you wanted `DefaultPing.txt` to be run instead of `libPollerStitcherDefaultPing.so`, you must rename `DefaultPing.txt` to `DefaultPing.stch`.

Stitcher rules are procedures which can be called from within the stitcher. The stitchers are written in Netcool/Precision IP's own stitcher language, and use OQL (a programming language unique to Netcool/Precision IP, based on SQL) to interact with component databases.

# 4.2 Monitoring Stitchers

The polling definitions supplied with Netcool/Precision IP use the monitoring stitchers. This section describes each of the stitchers.

## Poll Definition Attributes

The following attributes of the poll definitions are mandatory for all stitchers:

- `PollName`
- `PollStatus`
- `AgentName`
- `AgentKey`
- `StitcherName`
- `StitcherInfo`

The `StitcherInfo` attribute is compulsory and must be present in any poll definition, however, it may not necessarily have any sub-attributes defined.

The `Scope` attribute is optional. If it is excluded the poll definition passes all types of device. The `Scope` attribute only applies to stitchers that have been executed by `ncp_m_timedstitcher` and `ncp_m_visionary`.

The poll definition attribute descriptions are not included in the following sections. For more information on poll definition attributes, see *Attributes of the Poll Definitions* on page 75.

The following sections describe the stitchers, including the poll definition attributes they use.

**Note:** Mandatory poll definition attributes are not listed.

## Precompiled Stitchers

This section describes the precompiled stitchers. This section also identifies any alternative text-based versions of precompiled stitchers.

## SNMP Threshold Polling

The `libPollerStitcherAocDefinedThreshold.so` stitcher is used to perform SNMP threshold polling. The stitcher is described in Table 41.

Table 41: Overview of the libPollerStitcherAocDefinedThreshold Stitcher

| | |
|---|---|
| Name | `libPollerStitcherAocDefinedThreshold.so` |
| Type | Precompiled |
| Poll definition attributes used | `Frequency`<br>`Scope`<br>`StitcherInfo`<br>• `RuleSet`<br>• `EventName`<br>• `Varbinds`<br>• `TablePoll`<br>• `Threshold`<br>• `Description`<br>• `ClearThreshold`<br>• `ClearDescription`<br>• `AdditionalVarbinds`<br>• `IndexExtractions` |
| Agent | `ncp_m_timedstitcher` |

This stitcher uses unique extensions to the eval statement within some of the poll definition attributes, as described below.

## Extended Eval Language Support for SNMP Threshold Polling

When used with the `AocDefinedThreshold` stitcher, the following poll definition attributes can make use of an extended subset of the eval language:

- `Threshold`
- `Description`
- `ClearThreshold`
- `ClearDescription`

You can use eval statements in the above attributes in the same way as normal eval statements, however, you can evaluate a greater range of information. For more information on the syntax and use of the eval statement, see the *Netcool/Precision IP Discovery Configuration Guide*. Use of the extended subset of the eval statement is described in the examples below.

**Note:** The information you are trying to evaluate must have already been retrieved by the `Varbinds` or `AdditionalVarbinds` poll definition attribute.

### Example Evaluation of SNMP Values

The following example returns the value of the SNMP variable `sysName`.

```
eval(text, '&SNMP.VALUE.sysName')
```

### Example Evaluation of SNMP Indices

The following example returns the value of the index of the SNMP request for the variable `ipRouteNextHop`. In a table poll, this will be evaluated for every index in the table list.

```
eval(text, '&SNMP.INDEX.ipRouteNextHop')
```

### Example Evaluation of Previously-Retrieved SNMP Values

The following example returns the value of the SNMP variable `sysName`, which was retrieved when this poll was last run.

```
eval(text, '&SNMP.VALUE.OLD.sysName')
```

### Example Evaluation of Old SNMP Indices

The following example returns the value of the index of the SNMP request for the variable `ipRouteNextHop, which was retrieved when this poll was last run`. In a table poll, this will be evaluated for every index in the table list. Note that the old index is likely to be the same as the new index.

```
eval(text, '&SNMP.INDEX.OLD.ipRouteNextHop')
```

### Example Evaluation of Entity Values

The following example returns the ObjectId from the MODEL record for the entity being polled.

```
eval(text, '&ENTITY.ObjectId')
```

Instead of `ObjectId`, you can specify any of the column names in the `master.entityByName` database table.

### Example Evaluation of Poll Definition Values

The following example returns the frequency of the poll.

```
eval(text, '&POLL.Frequency')
```

Instead of `Frequency`, you can specify any of the poll definitions attribute of this poll.

**Note:** If you restrict the scope of this poll to a specific interface, the device is polled through that interface, but SNMP values for each interface on the device are retrieved. If you are only interested in specific interfaces, include a filter for the relevant interfaces within the `Threshold` attribute, as described in the example below.

### Example Interface Filter

If you are polling a device with many interfaces, but you are only interested in generating events for interfaces which have 'BRI' in the name, append a condition similar to the following to the `Threshold` attribute of your poll definition:

```
AND eval(text,"&SNMP.VALUE.ifName") like "^BRI"
```

### Example Polling Only Managed Interfaces

You can perform SNMP polls which exclude from polling interfaces automatically tagged for exclusion at discovery time.

The discovery component which tags interfaces for exclusion from polling is the `TagManagedEntities` stitcher, and is described in the *Netcool/Precision IP Discovery Configuration Guide*. This discovery stitcher stores interfaces to be excluded from polling in the `m_ExtraInfo` field for the main node, within `m_UnmanagedInterfaces`, using the format: `[ <ifIndex1>, <IfIndex2>, ..... <IfIndexN> ]`, where the IfIndices are the ifIndices of the interfaces you do not want the system to monitor.

Use the `snmpLinkState2` poll definition together with the `AocDefinedThreshold` stitcher to exclude from polling interfaces automatically tagged for exclusion at discovery time.

**Note:** The `snmpLinkState2` poll definition provides an alternative to the the `snmpLinkState` poll definition.

The `snmpLinkState2` poll definition is off by default. To enable this poll definition, set `PollStatus` stitcher to `1`.

By default, the poll definition to operate on all chassis (main node) entities. Scope is defined as follows:

```
Scope = 'EntityType = 1 and Contains is not null and IsActive = 1',
```

The `Threshold` attribute references entity attributes populated by the `TagManagedEntities` discovery stitcher. The Threshold attribute is evaluated once for each instance of the varbinds encountered. In this case, an instance is an interface. The section of interest follows:

```
[1]  (
[2]  ( eval(text,"&ENTITY.ExtraInfo->m_UnmanagedInterfaces") = NULL )
[3]  OR
[4]  ( NOT eval(int,"&SNMP.INDEX.ifAdminStatus") IN ( eval(list type
     int,"&ENTITY.ExtraInfo->m_UnmanagedInterfaces") ) )
[5]  )
```

For reasons of efficiency, in a typical deployment, the *unmanaged* interfaces are listed. If this list is not present (line 2), then Netcool/Precision IP assumes that events can be generated for all interfaces. If the list is present (line 4), then Netcool/Precision IP can only generate events for those interfaces *not* present in the list.

The `Threshold` test works as follows: an event is only generated when an interface is not up (`ifOperStatus!= up(1)`), but should be (`ifAdminStatus = up(1)`). When this occurs an event is generated with `Severity` set to `3`, and `Description` set as follows:

```
Description = 'Link down: eval(text,"&SNMP.VALUE.ifName") (
"eval(text,"&SNMP.VALUE.ifDescr")" )',
```

**Note:** You may wish to remove the reference to ifName, as this MIB object is not supported on all device types.

## Detecting Interface Flapping

The `libPollerStitcherDefaultFlap.so` stitcher is used to detect flapping. Flapping is a condition where a device or interface connects to and then disconnects from the network repeatedly in a short space of time. The stitcher sends an event if more than 5 `linkUp` and `linkDown` trap pairs have been received in a 60 second time period. Only `LinkUp` and `LinkDown` traps should be named in `KnownTraps` in the poll definition. The stitcher is described in Table 42.

Table 42: Overview of the libPollerStitcherDefaultFlap Stitcher (1 of 2)

| | |
|---|---|
| Name | `libPollerStitcherDefaultFlap.so` |
| Type | Precompiled |

Table 42: Overview of the libPollerStitcherDefaultFlap Stitcher (2 of 2)

| | |
|---|---|
| Poll definition attributes used | AgentControl<br>• OmitTraps<br>• KnownTraps<br>• UnknownTrapHandling<br>StitcherInfo<br>• RuleSet<br>• EventName<br>• CorrelateBy |
| Agent | _timedstitcher |

The AlternativeDefaultFlap.txt stitcher is functionally identical to libPollerStitcherDefaultFlap.so.

## Generic Trap Reporting

The libPollerStitcherDefaultTrap.so stitcher provides generic trap reporting. It takes the trap name and description from the trap, and all other information from the poll definitions, and sends an event when a trap is received. The agent executable ncp_m_trapstitcher, which runs this stitcher, uses the attributes in AgentControl to determine whether or not to run the stitcher for a particular trap.

Any varbinds sent as part of the trap are added to the ExtraInfo field of the event. If an appropriate definition exists in the MIB files, the varbind name is resolved and its text representation is used. If the definition does not exist, the OID value is used. The stitcher is described in Table 43.

Table 43: Overview of the libPollerStitcherDefaultTrap Stitcher (1 of 2)

| | |
|---|---|
| Name | libPollerStitcherDefaultTrap.so |
| Type | Precompiled |

Table 43: Overview of the libPollerStitcherDefaultTrap Stitcher (2 of 2)

| | |
|---|---|
| Poll definition attributes used | AgentControl<br>• OmitTraps<br>• KnownTraps<br>• UnknownTrapHandling<br>StitcherInfo<br>• EventName<br>• RuleSet<br>• Severity<br>• CorrelateBy |
| Agent | ncp_m_trapstitcher |

The AlternativeDefaultTrap.txt stitcher is functionally identical to libPollerStitcherDefaultTrap.so.

## Ping Polling

The libPollerStitcherDefaultPing.so stitcher runs ICMP polls on devices to check their availability. By default, the poll times out after five seconds, and a device is not repolled if a poll fails. These values can be overridden using the TimeOut and Retries attributes of the poll definitions. The stitcher sends an event if the poll fails, or if the poll succeeds, having previously failed for a particular device. The stitcher is described in Table 44.

Table 44: Overview of the libPollerStitcherDefaultPing Stitcher

| | |
|---|---|
| Name | libPollerStitcherDefaultPing.so |
| Type | Precompiled |
| Poll definition attributes used | FrequencyStitcherInfo<br>RuleSet<br>TimeOut<br>Retries<br>EventName |
| Agent | ncp_m_timedstitcher |

The AlternativeDefaultPing.txt stitcher is functionally identical to libPollerStitcherDefaultPing.so.

## Polling for Administrative or Operational Status Mismatches

The `libPollerStitcherSnmpLinkStatus.so` timed stitcher polls at intervals of five minutes by default. It compares the administrative and operational status of interfaces between polls. This stitcher is used as a backup to the monitoring of `linkUp` and `linkDown` traps. The stitcher is described in Table 45.

Table 45: Overview of the libPollerStitcherSnmpLinkStatus Stitcher

| | |
|---|---|
| Name | `libPollerStitcherSnmpLinkStatus.so` |
| Type | Precompiled |
| Poll definition attributes used | `Frequency`<br>`StitcherInfo`<br>• `RuleSet`<br>• `EventName` |
| Agent | `ncp_m_timedstitcher` |

Table 46 shows the events which are generated as a result of the changes in interface status. Additionally, an event is generated when a poll fails to return any data, and a Clear event is generated when a poll to the same device subsequently succeeds. The stitcher is described in Table 46.

Table 46: Events Generated by the libPollerStitcherSnmpLinkStatus Stitcher (1 of 2)

| Previous poll | | Current poll | | Event generated |
|---|---|---|---|---|
| Administrative status | Operational status | Administrative status | Operational status | |
| Up | Up | Up | Down | `"The interface has gone down."`<br>`Severity = Minor` |
| Up | Up | Down | Up | No event generated. |
| Up | Up | Down | Down | No event generated. |
| Up | Down | Up | Up | `"The interface has come up."`<br>`Severity = Clear` |
| Up | Down | Down | Up | `"The interface has come up, although it should be down."`<br>`Severity = Clear` |
| Up | Down | Down | Down | `"An administrator has confirmed that the interface should be down."`<br>`Severity = Clear` |
| Down | Up | Up | Up | No event generated. |

Table 46: Events Generated by the libPollerStitcherSnmpLinkStatus Stitcher (2 of 2)

| Previous poll | | Current poll | | Event generated |
|---|---|---|---|---|
| Administrative status | Operational status | Administrative status | Operational status | |
| Down | Up | Up | Down | `"The interface has gone down."`<br>`Severity = Minor` |
| Down | Up | Down | Down | No event generated. |
| Down | Down | Up | Up | No event generated. |
| Down | Down | Up | Down | `"An administrator has instructed the interface to come up, but it hasn't."`<br>`Severity = Minor` |
| Down | Down | Down | Up | No event generated. |

The `AlternativeSnmpLinkStatus.txt` stitcher is functionally similar to `libPollerStitcherSnmpLinkStatus.so`. However, instead of polling a main node to find out the status of that node's interfaces, this stitcher can poll individual interfaces. This is useful if, for example, you wanted to poll only those interfaces on a node which had a particular `ifType`.

**Note:** This stitcher uses `AgentKey+`. For Information on the use of the plus character after the `AgentKey` attribute, see *Attributes of the Poll Definitions* on page 75.

## Text-Based Stitchers

The following stitchers are referenced by the poll definitions. You can change their parameters through the poll definitions, and you can, if necessary, alter their logic by editing their text files.

### Monitoring Cisco Power Supplies

The `CiscoPowerSupply.stch` stitcher sends an event if there is a problem with a Cisco power supply unit (PSU), or sends a restore event if a previous problem with the PSU is no longer detected. The stitcher monitors any Cisco device which supports the MIB variables `chassisPs1Status` and `chassisPs1Status`. The stitcher is described in Table 47.

Table 47: Overview of the CiscoPowerSupply Stitcher (1 of 2)

| | |
|---|---|
| Name | `CiscoPowerSupply.stch` |
| Type | Text-based |

Table 47: Overview of the CiscoPowerSupply Stitcher (2 of 2)

| | |
|---|---|
| Poll definition attributes used | `Frequency`<br>`StitcherInfo`<br>• `EventName`<br>• `RuleSet`<br>• `MibVariable`<br>• `Severity` |
| Agent | `ncp_m_timedstitcher` |

## Performing Simple Threshold Polling Against a Specified MIB Variable

The `DefaultSnmpThreshold.stch` stitcher is a generic stitcher which performs an `SNMP GetBulk` request on a device and tests the returned variables against an SNMP evaluation condition. You should use this stitcher only if you require different functionality to that provided by the precompiled stitcher `AocDefinedThreshold`. The evaluation condition is defined in the `StitcherInfo` section of the poll definition. The stitcher is described in Table 48.

Table 48: Overview of the DefaultSnmpThreshold Stitcher

| | |
|---|---|
| Name | `DefaultSnmpThreshold.stch` |
| Type | Text-based |
| Poll definition attributes used | `Frequency`<br>`StitcherInfo`<br>• `Description`<br>• `RuleSet`<br>• `EventName`<br>• `Severity`<br>• `RestoreEvent`<br>• `PollFailEvent`<br>• `MibVariable`<br>• `Comparison`<br>• `Threshold` |
| Agent | `ncp_m_timedstitcher` |

## Checking for Syslog Messages

The `DefaultSyslog.stch` stitcher is used to monitor system messages. It sends events based on updates to system files. The stitcher is described in Table 49.

Table 49: Overview of the DefaultSyslog Stitcher

| | |
|---|---|
| Name | `DefaultSyslog.stch` |
| Type | Text-based |
| Poll definition attributes used | `AgentControl`<br>• `FileNames`<br>• `FieldNames`<br>• `Reread`<br>• `RegExps`<br>• `Delimiter`<br>• `Mappings`<br>`StitcherInfo`<br>• `Threshold`<br>• `Severity` |
| Agent | `ncp_m_syslogstitcher` |

## Analysis of IP Traffic Statistics

The `SnmpIPMonitoring.stch` stitcher performs statistical analysis of IP traffic on a device. It polls at intervals, and compares the values returned by the last poll to the values returned by the current poll. It calculates the amount of traffic in the interval between polls, as well as the number of various errors that have occurred, including fragmented packets.

Fragmenting packets can represent a major performance issue. Routers that are fragmenting packets (with or without errors) may be doing so as a result of misconfigured PDU (Packet Data Unit) sizes between the link and the network layer or mismatched PDU sizes between routers.

The stitcher sends an event if any of the following conditions are met:

- The percentage of inbound packets with errors is greater than five percent of the total number of inbound packets.
- The percentage of outbound packets with errors is greater than five percent of the total number of outbound packets.
- Any routing errors have occurred.

- Any errors have occurred due to the device being unable to fragment packets, or due to the device being unable to reassemble fragmented packets.
- The device has itself fragmented any packets.

The stitcher is described in Table 50.

Table 50: Overview of the SnmpIPMonitoring Stitcher

| | |
|---|---|
| Name | `SnmpIPMonitoring.stch` |
| Type | Text-based |
| Poll definition attributes used | `Frequency`<br>`StitcherInfo`<br>• `RuleSet` |
| Agent | `ncp_m_timedstitcher` |

## Monitoring Traffic on Each Interface

The `SnmpLinkMonitoringAllInterfaces.stch` stitcher monitors interface traffic. It polls devices at intervals, measuring the traffic across interfaces and the number of errors between polls.

The stitcher sends an event if any of the following conditions are met:

- Errors on inbound packets are greater than ten percent of the total inbound packets.
- Errors on outbound packets are greater than ten percent of the total outbound packets.
- Total errors constitute more than ten percent of the total number of packets.
- The device is not of a valid type (For example, the device has no interfaces).

The stitcher is described in Table 51.

Table 51: Overview of the SnmpLinkMonitoringAllInterfaces Stitcher

| | |
|---|---|
| Name | `SnmpLinkMonitoringAllInterfaces.stch` |
| Type | Text-based |
| Poll definition attributes used | `Frequency`<br>`StitcherInfo`<br>• `RuleSet` |
| Agent | `ncp_m_timedstitcher` |

## Monitoring TCP Traffic for a Device

The `SnmpTCPMonitoring.stch` stitcher monitors TCP (Transmission Control Protocol) traffic on a device. It polls a device at intervals and measures the amount of traffic and the number of errors between polls.

The stitcher generates an event if any of the following conditions are met:

- Inbound errors are greater than five percent of the total inbound traffic.
- Outbound errors are greater than five percent of the total outbound traffic.
- Total errors are greater than five percent of the total traffic (including inbound and outbound segments, and retransmitted segments).
- No SNMP results are returned.

The stitcher is described in Table 52.

Table 52: Overview of the SnmpTCPMonitoring Stitcher

| | |
|---|---|
| Name | `SnmpTCPMonitoring.stch` |
| Type | Text-based |
| Poll definition attributes used | `Frequency`<br>`StitcherInfo`<br>• `RuleSet` |
| Agent | `ncp_m_timedstitcher` |

## Checking for Possible Device Reboots

The `SysUpTime.stch` stitcher polls devices to find out how long they have been up. It stores the `sysUpTime` from the previous poll and compares it to the current value. If the current value is less than the previous one, the device must have restarted in the meantime, and an event is sent to this effect. The stitcher is described in Table 53.

Table 53: Overview of the SysUpTime Stitcher

| | |
|---|---|
| Name | `SysUpTime.stch` |
| Type | Text-based |
| Poll definition attributes used | `Frequency`<br>`StitcherInfo`<br>• `RuleSet` |
| Agent | `ncp_m_timedstitcher` |

# 4.3 Stitcher Rules

This section provides the information about the text-based stitcher rules. It also provides examples of the usage of each stitcher rule. This section must be read in conjunction with stitcher language appendix in the *Netcool/Precision IP Discovery Configuration Guide*. This section also assumes that you are familiar with the polling process as described in Chapter 2: *Network Polling* on page 21.

## Stitcher Rules for MONITOR and DISCO

The stitcher rules that are common to DISCO and the polling agents are described in Table 54. These stitcher rules are not tagged with a prefix.

Table 54: Stitcher Rules Common to DISCO and the Polling Agents

| Stitcher Rule Name | Description |
|---|---|
| `ExecuteStitcher` | Invokes the execution of a named stitcher. |
| `ExecuteOQL` | Declares stitcher rules that are written in OQL. Anything between the parentheses following the `ExecuteOQL` rule must be valid OQL syntax. |
| `RetrieveOQL` | Creates a list of the data type `RecordList` generated by OQL statements. |

## Stitcher Rules for Polling Agents

This section describes the stitcher rules that are unique to the polling agents. The stitcher rules are:

- `PollerDoesTableExist`
- `PollerDoPing`
- `PollerGetLocalIpAddr`
- `PollerGetPollDef`
- `PollerGetTriggerRecord`
- `PollerInsertRecords`
- `PollerIntDeltaRecordList`
- `PollerMibTextToOid`
- `PollerSnmpGetBulk`
- `PollerGetLocalEntityName`
- `SendEvent`

These stitcher rules are tagged with a prefix, for example *Poller*`DoPing`.

## PollerDoesTableExist

The `PollerDoesTableExist` stitcher rule determines whether a database and table exists. The stitcher rule is described in Table 55.

Table 55: PollerDoesTableExist Stitcher Rule Description

| Variable | Description |
|---|---|
| Rule | `PollerDoesTableExist(dbName, tblName)` |
| Input | The variables `dbName` and `tblName`. These must be predefined. See example usage. |
| Output | Integer variable which is used to determine whether the database exists. False (0), True (1). |

### Example

In the following example, the integer variable `exists` is specified in the call to the rule. This variable can then be tested to determine whether the database table exists.

```
text dbName = "mojo";
text tlbName = "events";
int exists = PollerDoesTableExist(dbName, tblName);
```

The values `dbName` and `tblName` are defined for `mojo.events`.

## PollerDoPing

The `PollerDoPing` stitcher rule pings a device. If the device is uncontactable, the error condition is extracted from the ICMP datagram and is available for use in the stitcher. The stitcher rule is described in Table 56.

Table 56: PollerDoPing Stitcher Rule Description (1 of 2)

| Variable | Description |
|---|---|
| Rule | `PollerDoPing(ipAddress, TimeOut, retries)` |

Table 56: PollerDoPing Stitcher Rule Description (2 of 2)

| Variable | Description |
|---|---|
| Input | `ipAddress`—specifies the IP address of the network entity to be polled. This is generally extracted from the associated entity using an eval statement.<br>`TimeOut`—specifies the time to wait for a response from a device. If a device does not respond within the timeout period the device is repolled based on the next input, retries.<br>`retries`—specifies the number of times the device should be repolled if the poll is unsuccessful. After this, the poll is considered to have failed. |
| Output | A single record, which consists of two columns, `pingResults` and `errorString`:<br>• `pingResults`—an enumerated type which specifies the result of the ping poll. It can have the following values:<br>– `$Success` = 0<br>– `$TimeOut` = 1<br>– `$MultipleRetries` = 2<br>– `$ErrorReply` = 3<br>– `$AwaitingResponse` = 4<br>• `errorString`—contains the ICMP error string returned in the datagram. |

## Example

The following example assigns the output of the `PollerDoPing` rule to the variable `pingResults`, which is of type `RecordList`. To use `pingResults`, you must assign integer and text variables, as shown below.

```
RecordList pingResults = PollerDoPing(ipAddress, TimeOut, retries);

foreach (pingResults)
          {
               int myResult = eval(int,'&pingResults');
               text myString = eval(int,'&errorString');
          }
```

## PollerGetLocalIpAddr

The `PollerGetLocalIpAddr` stitcher rule retrieves the IP address of the device on which the polling agent is running (the polling location). The stitcher rule is described in Table 57.

Table 57: PollerGetLocalIpAddr Stitcher Rule Description

| Variable | Description |
|---|---|
| Rule | `PollerGetLocalIpAddr` |
| Input | Not required, see example usage. |
| Output | Text variable specifying the IP address of the polling agent. |

In previous versions of Netcool/Precision IP, the root cause analysis process required the IP address of the machine on which the stitcher agent is running. In the current release of Netcool/Precision IP the root cause analysis process uses the entity name stored in the network topology database. The `EntityName` can be obtained using the `PollerGetLocalEntityName` stitcher rule, as described in *PollerGetLocalEntityName* on page 114.

### Example

In the following example, the IP address output by the rule is assigned to the text variable `agentAddr`. The IP address can then be extracted from this variable by using an eval statement of the form `eval(text,'agentAddr')`.

```
text agentAddr = PollerGetLocalIpAddr();
```

## PollerGetPollDef

The `PollerGetPollDef` stitcher rule retrieves the poll definition associated with a device and allows the poll definition to be available in the scope of the stitcher. The stitcher rule is described in Table 58.

Table 58: PollerGetPollDef Stitcher Rule Description

| Variable | Description |
|---|---|
| Rule | `PollerGetPollDef()` |
| Input | Not required, see example usage. |
| Output | A single record, which consists of columns based on the `polldefCache.polldefs` table in the MONITOR stitcher agent configuration files. |

### Example

The following example assigns the value of the `PollStatus` column.

```
RecordList pollDef = PollerGetPollDef();

foreach(pollDef)
          {
               int myStatus = eval(int,'&PollStatus');
          }
delete(pollDef);
```

This example, uses the `RecordList` rule. To extract the column values from the `RecordList`, you need to assign them to variables.

## PollerGetTriggerRecord

The `PollerGetTriggerRecord` stitcher rule retrieves the trigger record for a non-timed agent, for example, the trap trigger record, and allows it to be available in the scope of the stitcher. The stitcher rule is described in Table 59.

**Note:** Only non-timed agents, such as, `ncp_m_trapstitcher` and `ncp_m_syslogstitcher`, have trigger records.

Table 59: PollerGetTriggerRecord Stitcher Rule Description

| Variable | Description |
|---|---|
| Rule | `PollerGetTriggerRecord()` |
| Input | Not required, see example usage. |
| Output | A single record, which consists of columns based on the `triggers.despatch` table in the MONITOR stitcher agent configuration files. |

### Example

The following example uses a column from a Trap agent trigger record.

```
RecordList triggerRecord = PollerGetTriggerRecord();

foreach(triggerRecord)
          {
               int myTrapType = eval(int,'&TrapType');
          }
```

This example, uses the `RecordList` rule. To extract the column values from the `RecordList`, you need to assign them to variables.

## PollerInsertRecords

The `PollerInsertRecords` stitcher rule inserts the contents of a list into a database table. This stitcher should not be used for inserting information into an external database, for example, `class.activeclasses`. It should be used for inserting information into a database which has already been defined within the stitcher. For information on creating databases and tables, see the *Netcool/Precision IP Discovery Configuration Guide*. The stitcher rule is described in Table 60.

Table 60: PollerInsertRecords Stitcher Rule Description

| Variable | Description |
|---|---|
| Rule | `PollerInsertRecords(myList, dbName, tblName)` |
| Input | `myList`—the name of the `RecordList` you want to insert.<br><br>`dbName`—the name of the target database.<br><br>`tblName`—the name of the target table.<br><br>The `dbName` and `tblName` should already be defined in the stitcher file, as shown in the example below. |
| Output | As a result of running this rule, the specified `RecordList` is inserted into the specified database table. |

### Example

The following example defines the `dbName` and `tblName` in the first section of the stitcher file.

```
text dbName="DeviceA";
text tblName="tcpHistory";
```

The `RecordList` used is called `snmpResults`. This would previously have been retrieved using a rule such as `PollerSnmpGetBulk`.

The format of the rule for the `RecordList` `snmpResults` is.

```
PollerInsertRecords(snmpResults, dbName, tblName);
```

The output is a database with the following format.

```
DeviceA.tcpHistory.tcpSegments
DeviceA.tcpHistory.tcpErrors
DeviceA.tcpHistory.tcpInSegments
DeviceA.tcpHistory.tcpInErrors
DeviceA.tcpHistory.tcpOutSegments
DeviceA.tcpHistory.tcpOutErrors
```

In the above output the first component of each row is the database name, for example, DeviceA. The second component is the table name. for example, tcpHistory, and the third component is the name of an SNMP variable from the snmpResults list, for example, tcpSegments. Immediately after the contents of the list snmpResults have been inserted into the database, the list is deleted or discarded and is no longer available.

## PollerIntDeltaRecordList

The PollerIntDeltaRecordList stitcher rule determines the difference between two lists of SNMP variables. Typically these lists would have been retrieved from the network using the PollerSnmpGetBulk rule. The stitcher rule is described in Table 61.

Table 61: PollerIntDeltaRecordList Stitcher Rule Description

| Variable | Description |
|---|---|
| Rule | PollerIntDeltaRecordList(historyResults, snmpResults) |
| Inputs | The inputs to PollerIntDeltaRecordList are two lists of SNMP variables. For example, historyResults and snmpResults. These inputs can be written generically as PollerIntDeltaRecordList(a,b). The input lists must be consistent in terms of length and the variables they contain. The stitcher rule PollerIntDeltaRecordList only determines the difference between identical lists. If the lists are dissimilar in any way the result is undefined. |
| Output | The result or output of PollerIntDeltaRecordList follows the logic b - a and is a list of variables which contains the differences between b and a. In the code table below, the output is assigned to a list called deltaResults. Additionally, the list "a" gets deleted from the symbol table and is no longer available after the subtraction function has been undertaken. |

### Example

The following example performs a delta function to allow statistical analysis to be conducted.

```
deltaResults = PollerIntDeltaRecordList(historyResults,snmpResults);
```

A threshold condition (or watermark calculation) can then be evaluated and as a consequence an event generated and inserted into mojo.events. Additionally, it is possible to include some of the information from the statistical analysis in a list, which can be included in the ExtraInfo column of the generated event record.

## PollerMibTextToOid

The `PollerMibTextToOid` stitcher rule converts an SNMP text variable to its OID equivalent. The stitcher rule is described in Table 62.

Table 62: PollerMibTextToOid Stitcher Rule Description

| Variable | Description |
|---|---|
| Rule | `PollerMibTextToOid(MIBText)` |
| Input | `MIBText`—the name of an SNMP variable, for example, `SysDescr`. |
| Output | A text variable containing the OID of the SNMP variable. |

### Example

The following example assigns the OID of the SNMP variable `SysDescr` to the text variable `MySnmpVar`.

```
text MySnmpVar = PollerMibTextToOid(SysDescr);
```

## PollerSnmpGetBulk

The `PollerSnmpGetBulk` stitcher rule retrieves SNMP data from network devices, using an SNMP GetBulk request. This allows the total amount of SNMP traffic generated when downloading non-scalar (or large numbers of individual scalar) SNMP variables to be reduced.

The rule can be used to undertake a table poll, for instance, to retrieve values from a device and its interfaces. The column names in the returned record are defined by the list of SNMP input variables made in the stitcher rule call to `PollerSnmpGetBulk`. The values for each of these column names are the values retrieved from each device and its relevant interfaces. The final column name in the returned record is called `m_Indices` and is a list of indices associated with the interfaces (`ifIndex`) on the polled device.

**Warning:** It is important that the correlation between variables is correct. If the first *n* variables in the variable list are non-repeater variables, the value put into the `nonRepeaters` argument must be *n*. Non-repeater variables are scalar MIB variables such as `sysUpTime` and `ifNumber`.

The stitcher rule is described in Table 63.

Table 63: PollerSnmpGetBulk Stitcher Rule Description

| Variable | Description |
|---|---|
| Rule | `PollerSnmpGetBulk(ipAddress, communitySuffix, [list of SNMP variables], nonRepeaters, maxRepetitions, appendageString, TimeOut, retries)` |
| Inputs | `ipAddress`—specifies the IP address of the network entity that is being polled. This is generally extracted from the associated entity via an eval statement.<br><br>`communitySuffix`—provides an option to add a suffix to the SNMP community string. Default is NULL.<br><br>`[list of SNMP variables]`—Specifies the list of SNMP variables and their values to retrieve from the network.<br><br>`nonRepeaters`—specifies the number of standalone SNMP variables, i.e., those that are only retrieved once. For instance, if `nonRepeaters` was equal to 2 then a single value would be retrieved from the network for the first two SNMP variables in the list - in the example usage of `PollerSnmpGetBulk` which follows, these would be `sysDescr` and `sysContact`. The remaining variables in the list are treated as non-scalar (repeating) variables, for which either the total number of instances or `maxRepetitions` instances—whichever is smaller—are retrieved.<br><br>`maxRepetitions`—specifies the maximum number of instances of a non-scalar SNMP variable which is downloaded when performing the SNMP GetBulk request. For example, if '`ifOperStatus`' were being retrieved from a device with 100 entries in the `ifTable`, and `maxRepetitions` were set to 50, only the first 50 instances of '`ifOperStatus`' from the `ifTable` would be retrieved.<br><br>If you set `maxRepetitions` to `-1`, all instances of each specified repeater variable are retrieved. As a result, you do not need to know the number of instances each column variable has in advance of making the `PollerSnmpGetBulk` call.<br><br>`TimeOut`—specifies the time to wait for a response from a device. If a device does not respond during the time out period the device is repolled based on the next input, `retries`.<br><br>`retries`—specifies the number of times the device should be repolled. |
| Output | The output from `PollerSnmpGetBulk` is a list of records. The column names of such records are dependant on the list of SNMP variables specified in the stitcher rule call `PollerSnmpGetBulk`, for instance, `snmpResults`. However, the final column name in the returned record is always called `m_Indices`, which is a list of indices associated with the interfaces (`ifIndex`) on the polled device. |

## Example

A device with three interfaces is being polled for the following variables: `sysDescr`, `sysContact`, `ifInUcastPkts` and `ifInNUCastPkts`. The poll definition in the AOC which has led to this stitcher rule being triggered includes the section.

```
StitcherInfo = {
            TimeOut = 5000,
```

```
            Retries = 2,
}
```

One way the call to the rule `PollerSnmpGetBulk` might be executed in the stitcher is as follows.

```
// get the IP address of the entity being processed by this Stitcher
text ipAddress = eval(text,'&Address(2)')
int TimeOut = 0;
int retries = 0;
int maxRepetitions = 0;
int nonRepeaters = 2;
// set maximum repetitions to the actual number of interfaces
maxRepetitions = ExecuteStitcher('SnmpGetIfNumber');
RecordList pollDef = PollerGetPollDef(); // get poll definition
foreach(pollDef) {
                          TimeOut = eval(int, '&StitcherInfo->TimeOut');
                          retries = eval(int, '&StitcherInfo->Retries');
}
RecordList                                snmpResults=PollerSnmpGetBulk(
                          ipAddress,
                          NULL,   // appendage to the community string
                          [
                               'sysDescr',
                               'sysContact',
                               'ifInUcastPkts',
                               'ifInNUcastPkts'
                          ],
                          nonRepeaters,     // number of non repeating SNMP variables
                          maxRepetitions,   // maximum instances of non-scalar SNMP
                                            // variables
                          NULL,   // OID appendage to the SNMP variable list
                          TimeOut,   // defined in the AOC
                          retries,   // defined in the AOC
                          );
```

The output (`snmpResults`) from the above (provided the device is contactable) would resemble the following.

```
sysDescr.0             = Cisco7500 Router;
sysContact.0           = Alfred Sanders;
ifInUcastPkts.2        = 223256;
ifInNUCastPkts.2       = 124001;
ifInUcastPkts.6        = 234500;
ifInNUCastPkts.6       = 127780;
ifInUcastPkts.9        = 453012;
ifInNUCastPkts.9       = 321544;
m_Indices              = ['2', '6', '9'];
```

The `PollerSnmpGetBulk` rule treats quoted strings as literal values and unquoted strings as stitcher variables, which should contain an appropriate value for the parameter position they occupy. For example, `TimeOut` in the above call is resolved to the integer 5000, whereas ‘`sysDescr`’ is one of the SNMP variables which the `PollerSnmpGetBulk` rule attempts to retrieve.

## PollerGetLocalEntityName

The `PollerGetLocalEntityName` stitcher rule attempts to retrieve the `EntityName` field of the topology entry that represents the machine on which the stitcher agent is running.

The stitcher rule determines the IP address of the machine then executes the following query against MODEL.

```
select EntityName from master.entityByName where EntityType = 1 AND
        (
        Address(2) = ipAddr OR
        ipAddr IN (ExtraInfo->m_AssocAddress)
        )
;
```

Where *`ipAddr`* is the IP address of the machine on which the stitcher agent is running.

## SendEvent

The `SendEvent` stitcher rule is used to send events, in the form of OQL inserts, from the monitoring process to the virtual database `mojo.events` in the MONITOR probe.

To ensure the event is sent to the MONITOR probe the `service` command line option must be set to `Monitor2ObjServ`. If no service has been specified the event is sent to the database `mojo.events` in AMOS. Micromuse does not recommend the delivery of events directly to AMOS using this stitcher rule.

The stitcher rule is described in Table 64.

Table 64: SendEvent Stitcher Rule Description

| Variable | Description |
|---|---|
| Rule | `SendEvent(OQL insert)` |
| Input | A standard OQL database insert (for information on using OQL to make inserts into databases, see the *Netcool/Precision IP Discovery Configuration Guide*). In the stitchers, the inserts are to the virtual database `mojo.events` in the MONITOR probe. |
| Output | As a result of running this rule, an event is sent to the MONITOR probe, then to the ObjectServer, then to the gateway, and finally to AMOS for event correlation. |

## Example

The following example shows an insert ping fail event using the `SendEvent` rule.

```
SendEvent(
          "
          insert into mojo.events
          (
              EventId,
              EntityName,
              ClassName,
              Description,
              EventName,
              RuleSet,
              EventType,
              Severity,
              AssignedTo,
              Acknowledged,
              AgentAddress,
              EventGroupId
          )
          values
          (
               0,
               eval(text,'&EntityName'),
               eval(text, '&ClassName'),
               eval(text,'CAT(`Ping fail for `,&Address(2),` `,$errorString)'),
               eval(text, '$eventName'),
               eval(text, '$ruleSet'),
               0,
               3,
               '',
               0,
               eval(text,'$localAddr'),
               0
          );
          "
     );
```

# 4.4 Creating and Editing Stitchers

This section provides the information required to create or edit text-based stitchers. This section must be read in conjunction with stitcher language appendix in the *Netcool/Precision IP Discovery Configuration Guide*.

**Warning:** Micromuse recommends that creating and editing stitchers is only undertaken by advanced users of Netcool/Precision IP. You must be familiar with all aspects of network polling and Netcool/Precision IP.

Before creating or editing a stitcher you should:

- Ensure the polling functionality you require is not already available using the existing stitchers. It is easier to customize a poll definition than edit a stitcher.
- Back up your existing stitchers.
- Reduce the risk of errors by copying and modifying an existing stitcher.
- Ensure your new stitcher has a unique name and write a new poll definition referencing that name.
- Add the new poll definition to whichever classes you require to run the new poll.
- Store a copy of your stitcher in the `NCHOME/precision/monitor/stitchers` directory.

## Stitcher Scope

This section explains ampersand usage and scope within the context of MONITOR stitchers. General information on ampersand usage and scope is provided in the *Netcool/Precision IP Discovery Configuration Guide*.

In the context of scope, the following types of records are especially important:

- The *model instance*, also known as the *model record*. When an agent executable starts a polling stitcher, it passes the stitcher information about the device to be polled (or from which a trap has been received). This information is in the form of a record from MODEL's `master.entityByName` table, the schema of which can be found in the *Netcool/Precision IP Discovery Configuration Guide*. This information does not need to be retrieved, as it is automatically available.
- The *poll definition*. Every stitcher which is initially called by a polling agent is controlled by a poll definition. Some stitchers are called from within another stitcher, in which case they may, or may not, reference a poll definition. The poll definition must be brought into the scope of the stitcher using the stitcher rule `PollerGetPollDef`.
- The *trigger record*. This is relevant to non-timed stitchers only. The trigger record is contained in the `triggers.dispatch` database table of the relevant agent schema. The agent executable first populates the table with information relating to either the trap received or the syslog message found, and then starts the stitcher. For the `triggers.dispatch` schema, see Chapter 2: *Network Polling* on page 21. The trigger record must be brought into the scope of the stitcher using the `PollerGetTriggerRecord` stitcher rule.

The model instance can be referred to as being at the top level of scope available for reference in the stitcher (in the *global* scope). Any further levels of scope defined in the stitcher are inside the global scope. Variables in the global scope are usually referenced from within the stitcher using a single ampersand (unless they are referenced from within further levels of scope), for example:

```
    text    ipAddress = eval(text,‘&Address(2)’);
```

In the above code extract, the internal text variable `ipAddress` is defined (using an eval statement) as equal to the text variable `Address(2)`. `Address(2)` is from the record in the global scope, and since no further scopes have yet been defined, it is referenced with a single ampersand. In general, *one ampersand is used to reference values from the record currently in scope* (also known as the local scope). Internal variables, by contrast, do not require ampersands.

Further levels of scope within the global scope are defined by being enclosed in a pair of curly braces { }. Scope is most commonly defined in stitchers by using `foreach([variable]){}` loops.

There are usually several levels of scope defined within stitchers. To reference outside each level you require an extra ampersand. For example:

```
StitcherRules
{
...
    RecordList polldef = PollerGetPollDef();
    foreach(polldef)
    {
        AgentName = eval(text, ‘&AgentName’);
        text    ipAddress=eval(text,‘&&Address(2)’);
```

In this example code extract, the `AgentName` variable only requires one ampersand, because it references the record currently in scope (the poll definition).

The `ipAddress` variable (which when referenced in the first example required only one ampersand) now requires *two* ampersands, one to reference outside the current scope, and one to reference the global scope.

Any number of scopes can be defined in a stitcher, although nesting several scopes inside each other is not recommended for performance reasons. The number of ampersands required to reference a given record depends entirely on the relative positions of the record being referenced and where the record is being referenced from. The simplified example below shows some possible combinations.

```
// assume polldef and trigger record have been retrieved

&variable1    // This references the global scope
    foreach(polldef)
    {
        &variable2    //This references the polldef
        &&variable3    //This references the global scope
        foreach(triggerRecord)
        {
            &variable4    //This references the triggerRecord
            &&variable5    //This references the polldef
            &&&variable6    //This references the global scope
        }
    }
    foreach(snmpResults)
    {
        &variable7    //This references the snmpResults
        &&variable8    //This references the global scope
    }
```

In the above example, it is assumed that the poll definition and trigger record have been brought into the scope of the stitcher using the appropriate stitcher rules (`PollerGetPollDef` and `PollerGetTriggerRecord`).

It is important to note that neither the `polldef` nor the `triggerRecord` can be referenced from within the `foreach(snmpResults)` loop. This is because as far as the `foreach(snmpResults)` loop is concerned, the records in the `foreach(polldef)` and `foreach(triggerRecord)` loops either have not yet been created or have been destroyed.

## Stitcher Structure

All text-based stitchers have the following basic form:

```
UserDefinedStitcher
               {
```

```
          StitcherTrigger
               {
               }
          PollerStitcherExterns
               {
               }
          StitcherRules
               {
               }
     }
```

Each of the sections shown above is only used once in any one stitcher.

## UserDefinedStitcher

This section declares that the stitcher is a text-based stitcher. It must be included at the beginning of every stitcher in the form given above. The rest of the stitcher is enclosed within its curly braces.

## StitcherTrigger

Declares when the stitcher is to start. This section is used for the discovery stitchers. All monitoring stitchers are called on demand when the polling agent needs them. This section should therefore be left blank, as shown above.

## PollerStitcherExterns

This optional section defines external variables to be used in the stitcher.

Declared in the `PollerStitcherExterns` section, these variables are assigned their specified value the first time that the stitcher is run. Once they have been declared in the `PollerStitcherExterns` section, external variables can then have different values assigned to them in the next section, the `StitcherRules` section, in the same way as local variables.

External variables are not destroyed when the stitcher finishes, but are stored until the polling agent responsible for running the stitcher is stopped. For this reason, external variables can be created on one execution of a stitcher and used again on the next execution. This makes them useful for a variety of tasks including delta polling.

The code table below gives an example declaration of external variables:

```
PollerStitcherExterns
               {
                    extern int failedLastTime=0;
                    extern int multipleReplies=0;
               }
```

### Stitcher Rules

Local variables are declared only in the `StitcherRules` section of the stitcher. They are stored until the stitcher finishes or until they are deleted.

The stitcher rule contains the logic of the stitcher. This usually takes the following broad form:

```
StitcherRules
               {
               // declaration of local variables

               // retrieval of information using Stitcher rules

               // processing and evaluation of information

               // generation of event if appropriate
               }
```

**Tip:** To check your stitcher for parse errors, you can run `ncp_timedstitcher` in debug mode and examine the debug output. The following example shows one way to do this.

#### Example: Running ncp_timedstitcher in debug mode

If you have defined a poll using your stitcher with the key `MYSTITCHER`, you can use the following command to run `ncp_timedstitcher` in full debug mode and pipe the output to a file named `mydebug.out`:

```
ncp_m_timedtstitcher -domain TEST -latency 100000 -service Monitor2ObjServ -debug 4
-key MYSTITCHER >&mydebug.out&
```

Note that this command uses `csh` under UNIX. You should use the appropriate equivalent for your system.

## Poll Definitions and Stitchers

Monitoring stitchers can reference zero or more variables from the poll definitions. This is the normal way to control the polling process. The polling processes are described in *Default Polling Process Descriptions* on page 29.

Before the poll definitions can be used in a stitcher, they must be brought into scope. This is done using the `PollerGetPollDef` rule. The following example shows an extract of the poll definition which led to this stitcher being called.

```
// extract from poll definition

TimeOut = 12000,
AgentName = "ncp_m_timedstitcher",
```

```
StitcherInfo =
               {
                    EventName = "PingFail",
                    RuleSet='pingFailToCorrelatedRootCause'
               }
```

The example below shows how the above variables are referenced in the stitcher.

```
// extract from StitcherRules section of Stitcher

int timeOut=5000;  // declare local variables and set their
text ruleSet="";   // default values

RecordList pollDef = PollerGetPollDef();  // get the poll def

foreach(pollDef)  // referencing the poll def
               {
                  timeOut=eval(int,'&TimeOut');  // assign TimeOut from poll def to
                                                        // local variable timeOut

                  ruleSet=eval(text,'&StitcherInfo->RuleSet');  // assign RuleSet from
                                                        //poll def to local variable

               }

delete(pollDef);  // good practice to delete as no longer wanted
```

In this example you need to use the object identifier `->` to reference the `RuleSet` variable because it is contained within the object `StitcherInfo`. Objects within `StitcherInfo` do not have their data types defined in the poll definitions. Their data types must be defined in the stitcher. In this example `RuleSet` is defined as data type text.

# 4.5 Example Poll Definition and Stitcher

This section contains an example stitcher which illustrates the use of rules, scope and structure. It also includes the use of the poll definition interface.

## Poll description

This poll checks whether a device has recently rebooted. It retrieves the SNMP variable `sysUpTime`, which is defined as the number of milliseconds since a device was last initialized, and compares it to the value of `sysUpTime` which was retrieved on the last poll. If the second value is smaller than the first, this indicates that the device may have rebooted in the meantime, and an event is generated to warn the user of this.

## Poll definition

The poll definition which runs this poll appears in the AOC as follows.

```
{
PollName="SysUpReBoot",
AgentName="ncp_m_timedstitcher",
AgentKey="SNMP",
Frequency=300,
StitcherName="SysUpTime",
StitcherInfo={
      RuleSet='DefaultSNMPRuleSet',
   }
},
```

## Stitcher

The stitcher which is called to perform this poll is shown, broken down into sections. The first section shows the start of the stitcher file and the extern variable declarations.

```
UserDefinedStitcher
{

               StitcherTrigger  // called on demand
               {

               }

               PollerStitcherExterns
               {
                    extern int lastSysUpTime = -1;  // used to compare between polls
               }
```

The next section shows the local variables being declared, and information being retrieved from the poll definition.

```
StitcherRules
{
               text ipAddress = eval(text,'&Address(2)');   // declare local variables

               text entityName = eval(text,'&EntityName');  // & here refers to the
               text className = eval(text,'&ClassName');    // model instance

               text agentAddr = PollerGetLocalIPAddress();  // assign using rule

               text ruleset = 'NOT_SET';  // default value

               RecordList polldef = PollerGetPollDef();  // get poll def

               foreach(polldef)  // referencing poll def, get RuleSet from StitcherInfo
                   {
                      ruleset = eval(text, '&StitcherInfo->RuleSet');  // & is poll def
                   }
               delete(polldef);  // poll def no longer needed

int nonRepeaters = 1;  // needed as input to rule
```

The next section shows the SNMP poll being conducted.

```
RecordList snmpResults = PollerSnmpGetBulk  // poll for SysUpTime using rule
                                            // assign result of poll to snmpResults
               (                            // parentheses contain inputs to rule
                    ipAddress,       // ipAddress is an already-defined local variable
                    NULL,
                    [ 'sysUpTime' ],
                    nonRepeaters,
                    NULL,
                    NULL,
                    NULL,
                    NULL
               );

int sysUpTime=-1;  // sysUpTime can never be -1 from poll; note that this
                   // declares the local variable sysUpTime and does NOT
                   // overwrite the poll results because it is not within the
                   // foreach(snmpResults) loop
```

The next section shows the calculation which is used to decide whether or not to send an event.

```
foreach(snmpResults)  // referencing poll results
          {
               sysUpTime = eval(int,'&sysUpTime');  // & is snmpResults
               // eval statement means poll result is assigned to local variable so
               // can now be manipulated
```

```
        if(lastSysUpTime<>-1)  // remember lastSysUpTime is extern variable
                               // representing results of last poll; if this is
        {                      // the first poll it will = -1
             int deltaSysUpTime = 0;

        // calculate difference in sysUpTime

             deltaSysUpTime=sysUpTime-lastSysUpTime;

        // If deltaSysUpTime < 0 an event needs to be generated
```

The next section shows the event being generated, subject to the condition set up in the last section being met.

```
    if(deltaSysUpTime<0)
         {  // generate event

              SendEvent(
                  "insert into mojo.events  // mojo.events = AMOS Events Database
                  (  // list of columns
                       EventId,
                       EntityName,
                       ClassName,
                       Description,
                       EventName,
                       EventType,
                       Severity,
                       AssignedTo,
                       Acknowledged,
                       AgentAddress,
                       RuleSet,
                       EventGroupId
                  )
                  values
                  (
                       0,                 // $ must be used when referring to
                       eval(text,'$entityName'),   // Stitcher variables in eval
                       eval(text,'$className'),    // statements
                       eval(text,'CAT(`Possible Reboot condition: old sysUpTime
- `,$lastSysUpTime,` new sysUpTime - `,$sysUpTime)'),
                       'snmpSysUpTime',
                       0,
                       3,
                       '',
                       0,
                       eval(text,'$agentAddr'),
                       eval(text,'$ruleset'),
                       0
                  );
                  "
```

```
                    );
               }     // end event generation
          }  // end if(lastSysUpTime<>-1) loop
```

The final section ends the stitcher.

```
// store value of SysUpTime from this poll to be used in the next poll

               lastSysUpTime=sysUpTime;

}  // end foreach(snmpResults) loop

delete(snmpResults);

}  // end StitcherRules

}  // end Stitcher
```

# Chapter 5: The MONITOR Probe and Netcool/OMNIbus Probes

This chapter describes the functionality of the MONITOR probe, its role in the monitoring process, and how to start and configure it. It also describes how to configure other Netcool probes to populate the fields in the ObjectServer required for root cause analysis.

This chapter contains the following sections:

## 5.1 Overview of the MONITOR Probe

The MONITOR probe, `nco_p_ncpmonitor`, is designed to enable events generated by the Netcool/Precision IP polling agents to be sent to the ObjectServer. This ObjectServer sends these events, and other network events, through the Event Gateway, to AMOS for root cause analysis. The MONITOR probe is installed in the `NCHOME/probes` directory.

**Note:** `NCHOME` is the environment variable that contains the path to the Netcool Suite home directory. For information on how this environment variable varies with platform, see *Operating System Considerations* on page 9.

**Note:** The Netcool/Knowledge Library is a set of rules files written to a common standard. It enables Netcool/OMNIbus probes to work seamlessly with Netcool/Precision IP without any need for configuration. The Netcool/Knowledge Library is available with your Netcool/OMNIbus installation. It is also available as a download on the Micromuse Support Site.

# 5.2 Starting the MONITOR Probe

Micromuse recommends that the MONITOR probe is started using the domain process controller CTRL. The use of CTRL to automatically manage processes is described in the *Netcool/Precision IP Discovery Configuration Guide*. There are no dependencies for starting the MONITOR probe.

**Warning:** If you are using Netcool/Precision IP with failover, you must start the MONITOR probe using CTRL. The CTRL process checks the status of the MONITOR probe and uses this information to generate the Health Check events used by the failover process. For more information on failover, see the *Netcool/Precision IP Installation and Deployment Guide*.

## Manually Starting the MONITOR Probe

On Microsoft Windows, Netcool/Precision IP components can be run as processes or as Windows services. Components run as processes are started from a command prompt in the same way as on UNIX platforms. For more information on running components as Windows services, see the *Netcool/Precision IP Discovery Configuration Guide*.

To manually run the MONITOR probe enter the command `nco_p_ncpmonitor`.

The command line options for `nco_p_ncpmonitor` are:

```
nco_p_ncpmonitor -domain DOMAIN_NAME [ -buffer ] [ -buffersize ] [ -capturefile ] [
-debug DEBUG ] [ -help ] [ -latency LATENCY ] [ -manager ] [ -mapfile ]
[ -messagelevel ] [ -messagelog ] [ -name ] [ -nobuffer ] [ -noraw ] [ -propsfile ]
[ -raw ] [ -rulesfile ] [ -server ] [ -version ]
```

Table 65 describes the command line options for `nco_p_ncpmonitor`.

Table 65: nco_p_ncpmonitor Command Line Options (1 of 2)

| Option | Explanation |
|---|---|
| `-domain` *DOMAIN_NAME* | The name of the domain under which the Precision Server processes are running. |
| `-buffer` | Turn on alert buffering. |
| `-buffersize` | The size of the alert buffer to use. |
| `-capturefile` | Raw capture file to write to. |
| `-debug` *DEBUG* | The level of debugging output (1-4, where 4 represents the most detailed output). |
| `-help` | Prints out a synopsis of all command line options for the component.<br>If specified, the component is *not* started. |

Table 65: nco_p_ncpmonitor Command Line Options (2 of 2)

| Option | Explanation |
|---|---|
| `-latency` *`LATENCY`* | The maximum time in milliseconds (ms) that the component waits to connect to another Precision Server process via the messaging bus. This option is useful for large and busy networks where the default settings can cause the process to assume that there is a problem when in fact the communication delay is a result of network traffic. |
| `-manager` | Manager name. |
| `-mapfile` | Map file to read. |
| `-messagelevel` | Lowest level of message to be put into the message log. |
| `-messagelog` | Message log file to use for ObjectServer errors. |
| `-name` | Name of probe. |
| `-nobuffer` | Turn off alert buffering. |
| `-noraw` | Turn off raw capture mode. |
| `-propsfile` | Properties file to use. |
| `-raw` | Turn on raw capture mode. |
| `-rulesfile` | Rules file to use. |
| `-server` | ObjectServer to connect to. |
| `-version` | Prints the version number of the component.<br>If specified, the component is *not* started even if –version is used in conjunction with other arguments. |

## 5.3 The Probe and the Monitoring Subsystem

In order for the MONITOR stitcher agents to send events to the MONITOR probe, the stitcher agents must be started using the command line parameter `-service Monitor2ObjServ`.

In normal operation, the stitcher agents are started as required by MONITOR. The stitcher agents are started with the same `-service` parameter that was used to start MONITOR.

If you are using CTRL to start MONITOR as a managed process, you must ensure that the OQL inserts to the `ncp_ctrl` schema file `CtrlSchema.cfg` specify the `-service Monitor2ObjServ` parameter.

You can also start MONITOR manually using the following command line option:

```
ncp_monitor -service Monitor2ObjServ
```

For example, to manually start the `ncp_m_timedstitcher` stitcher agent, so that all events are sent to the probe, use a command line similar to the following:

```
ncp_m_timedstitcher -service Monitor2ObjServ
```

For more information about command line options for `ncp_monitor`, see *Starting MONITOR and Polling Agents* on page 22.

# 5.4 Configuring the MONITOR Probe

You can configure the operation of the probe using any of the following:

- Using the command line arguments described in *Manually Starting the MONITOR Probe* on page 129.
- Configuring the properties file.
- Configuring the map file.
- Configuring the rules file.

The properties (`nco_p_ncpmonitor.prop`), map (`nco_p_ncpmonitor.map`), and rules (`nco_p_ncpmonitor.rules`) files are installed in the `NCHOME/probes` directory.

## Properties File

A subset of the options available by using the command line arguments described above can be specified in the properties file. The properties file, in its unedited form, lists all the properties supported by the probe. To edit a property, you must remove the comment from the relevant line. An example line, which has been commented out, is shown below.

```
# MapFile   :   "NCHOME/probes/arch/nco_p_ncpmonitor.map"
```

To specify a different location for the map file, remove the comment and edit the line, as in the example below.

```
MapFile   :   "/home/johnsmith/nco_p_ncpmonitor.map"
```

In the above example, *`arch`* should be replaced by the name of the architecture on which the product is installed, for example, `solaris2`. For more information on configuring properties files, see the *Netcool/OMNIbus Administration Guide*.

## Map File

The probe converts events from the format in which they are generated by the polling agents to the format in which they are stored by the ObjectServer in a two-stage process. In the first stage, which is configured in the map file, the attributes of the event are converted to tokens accessible in the rules file. The example below shows the variable `ifIndex(1)`, which is contained in the object `ExtraInfo`, being mapped to the token `$IfIndex`.

```
$IfIndex = ExtraInfo->ifIndex(1)
```

You may need to edit the map file if your monitoring stitchers have been configured to use non-standard variables in generated events. Micromuse does not recommend storing complex data such as lists in the `ExtraInfo` object. For more information on the monitoring stitchers, see Chapter 4: *Stitchers Used for Polling* on page 89.

## Rules File

The second stage of the format conversion of events is governed by the rules file. In this stage, tokens are mapped to fields in the ObjectServer. This conversion is slightly more complex than the simple mapping of the first stage. The probe uses the rules file to perform some conditional processing on the tokens in order to convert them to fields in the ObjectServer. Configuring the rules file is only recommended for users with an advanced knowledge of Netcool/OMNIbus. For more information about the ObjectServer, see the *Netcool/OMNIbus Administration Guide*.

# Chapter 6: The Event Gateway

This chapter describes how to start and configure the Netcool/Precision IP Event Gateway. It also includes descriptions of the gateway databases.

This chapter contains the following sections:

## 6.1 Introduction to the Event Gateway

The Event Gateway provides a bidirectional interface between Netcool/Precision IP and Netcool/OMNIbus. The gateway enables you to send events from the Netcool/OMNIbus ObjectServer to the AMOS component of Netcool/Precision IP in order to perform root cause analysis.

The gateway also updates the ObjectServer with events from AMOS, and enriches events in the ObjectServer with network topology information from MODEL.

The gateway is used in conjunction with the MONITOR probe (described in Chapter 5: *The MONITOR Probe and Netcool/OMNIbus Probes* on page 127) to process events from the Netcool/Precision IP polling agents. The polling agents must be configured to send events to the probe, which then sends the events to the ObjectServer. From the ObjectServer they can be processed by the gateway.

# 6.2 Operation of the Gateway

On startup, the gateway downloads the topology from MODEL and stores it in an OQL database, described in *The Gateway Databases* on page 142.

**Note:** As the topology database may be large, you can use the command line options to configure the percentage of data that is cached on disk and the percentage that is stored in memory. Increasing the size of the disk cache reduces the memory used, however this may also reduce the throughput of the gateway.

Once the topology has been downloaded, the gateway listens for events from the ObjectServer. When events are received, the gateway uses the configuration information defined in the configuration file to filter the events. All matching events are enriched with topology data from MODEL. A number of these events are also mapped by the gateway to the AMOS database.

## Event Gateway Process

The flow of event data through the gateway is shown in Figure 15.

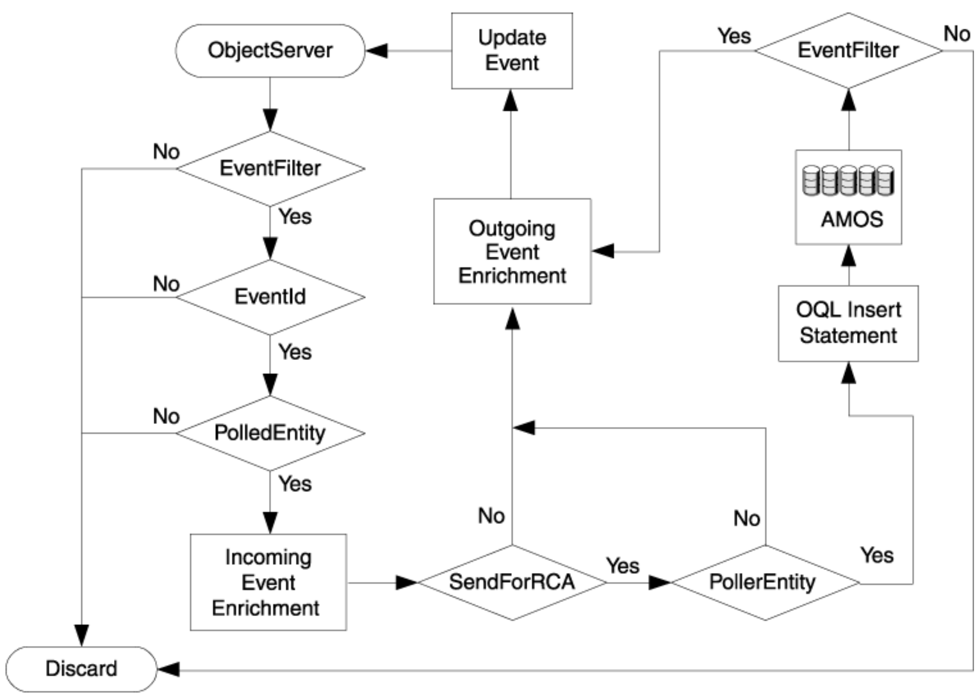


Figure 15: The Process Flow of Events Through the Gateway

The path an event follows through the gateway is:

1. When an event is received from the ObjectServer, the gateway checks that the event passes the EventFilter defined in the config.nco2ncp database table. If the event does not pass the filter, it is discarded.
2. The gateway finds the corresponding event mapping by searching the entries in the config.precedence table for an entry that matches the EventId.

**Note:** If no event mapping is found, the event is processed using the generic-event event map. This event map looks up the device relevant to the event in the topology and then updates the event in the ObjectServer. In this case, the event is not sent to AMOS for root cause analysis.

3. The gateway finds the location of the device that is referenced by the event in the topology model. For example, it may identify the IP address of the device. The gateway applies the PolledEntity topology filter from the event mapping to the entry in the topoCache.entityByName database table for the incoming event. If no topology entity is found, the event is discarded.
4. The event is now enriched with topology data from MODEL using the incoming EntityName.
5. The gateway determines whether the event should be sent to AMOS to perform RCA.

   If the SendForRCA field from the event map is set to 0, the event is not sent to AMOS for RCA. The enriched event is sent to the ObjectServer.

   If the SendForRCA field from the event map is set to 1, the event is sent to AMOS.
6. The gateway finds the location of the polling device in the topology model. For example, the gateway may find the IP address of the poller. The gateway applies the PollerEntity topology filter from the event map found in step 2 to the incoming event. If the polling device cannot be found, the event cannot be used for RCA. The enriched event is sent to the ObjectServer.
7. If the event is to be sent for RCA, an OQL insert statement is generated using the fields defined in the config.nco2ncp database table. This insert statement is sent to AMOS.
8. The gateway listens for updates from AMOS to obtain the result of the RCA calculation when it is completed. The AMOS event is tested against the EventFilter column of the config.ncp2nco table. If the event does not pass this filter it is discarded.
9. At this point the event can be further enriched with data, this time using the outgoing EntityName. This is important for trap and syslog events as the incoming EntityName for these events is often the main node (chassis) entity. AMOS, however, is able to search through the topology and set the outgoing EntityName to the interface entity within this main node.
10. If the AMOS event passes, the gateway generates an ObjectServer update based on the field mapping in the config.ncp2nco database table and sends the update back to the ObjectServer.

## Synchronizing the ObjectServer and the AMOS Database

The `mojo.events` database in AMOS should always contain a subset of the ObjectServer database. The AMOS database is always considered the slave database to the ObjectServer. To ensure consistency between the two databases, the gateway periodically checks the data is synchronized. You can change the frequency of this check by changing the `SyncCheckPeriod` value in the `config.defaults` gateway database. A value of `0` disables the synchronization checking.

When the gateway checks the synchronization of the databases, it checks that all the events in AMOS are also in the ObjectServer. If events are present in AMOS but not the ObjectServer, warning messages are generated in the Event Gateway log file.

The databases are also re-synchronized if no events are present in AMOS. This ensures that the databases automatically re-synchronize if AMOS is restarted.

## Forcing the Gateway to Synchronize

To force the gateway to synchronize at any time, enter the following SIGHUP command:

```
kill -HUP PID
```

Where *PID* with the process ID of the gateway.

In addition to synchronizing, the gateway checks the timestamp on its configuration file. If the configuration file has been modified, the gateway reads this file again in order to adapt to any configuration changes.

## Updating the Topology Cache

The gateway listens for updates to the Netcool/Precision IP MODEL database on the Rendezvous MODEL update subject. This ensures that the copy of the topology database in the gateway remains synchronized with the Netcool/Precision IP topology database.

# 6.3 Starting the Event Gateway

Micromuse recommends that the Event Gateway is started using the domain process controller CTRL. The use of CTRL to automatically manage processes is described in the *Netcool/Precision IP Discovery Configuration Guide*.

On Microsoft Windows, Netcool/Precision IP components can be run as processes or as Windows services. Components run as processes are started from a command prompt in the same way as on UNIX platforms. For more information on running components as Windows services, see the *Netcool/Precision IP Discovery Configuration Guide*.

**Warning:** If you are using Netcool/Precision IP with failover, you must start the Event Gateway using CTRL. The CTRL process checks the status of the Event Gateway component and uses this information to generate the Health Check events used by the failover process. For more information on failover, see the *Netcool/Precision IP Installation and Deployment Guide*

## Manually Starting the Event Gateway

To manually start the Event Gateway, run the `ncp_ncogate` command.

The command line options for `ncp_ncogate` are:

```
ncp_ncogate -domain DOMAIN_NAME [-debug DEBUG] [-latency LATENCY] [-cachesize
SIZE_IN_MB] [-cachepercent PERCENTAGE_OF_CACHE_IN_MEMORY] [-server OBJECTSERVER]
[-backup] [-help] [-version]
```

The command line options are described in Table 66.

Table 66: ncp_ncogate Command Line Options (1 of 2)

| Option | Description |
|---|---|
| `-domain` *DOMAIN_NAME* | The name of the domain under which to run `ncp_ncogate`. |
| `-debug` *DEBUG* | The level of debugging output (`1-4`), where `4` represents the most detailed output. |
| `-latency` *LATENCY* | The maximum time in milliseconds (ms) that `ncp_ncogate` waits to connect to another Precision Server process. This option is useful for large and busy networks where the default settings can cause processes to assume that there is a problem when in fact the communication delay is a result of network traffic.<br><br>The default value is `10000`. If you specify a lower value on the command line, it is increased to `10000`. |
| `-cachesize` *SIZE_IN_MB* | Specifies the size of the cache in megabytes (MB). |

Table 66: ncp_ncogate Command Line Options (2 of 2)

| Option | Description |
|---|---|
| `-cachepercent` *`PERCENTAGE_OF_CACHE_IN_MEMORY`* | Enables you to specify the ratio of the cache that is resident in memory to the cache that is resident on the hard disk.<br><br>The ratio that you specify depends on the amount of memory that exists on the host machine and the number of processes it is running. The default value is 100% memory cache.<br><br>Increasing the size of the disk cache reduces the memory consumption of the gateway, however, it can cause the gateway to run more slowly. |
| `-server` *`OBJECTSERVER`* | The name of the ObjectServer to connect to. This defaults to `NCOMS` if no server is specified. |
| `-backup` | Configures the Event Gateway to operate in backup mode. For information on failover, see the *Netcool/Precision IP Installation and Deployment Guide*. |
| `-help` | Prints out the command line options for `ncp_ncogate` then exits. |
| `-version` | Prints the version number of `ncp_ncogate` then exits. |

## 6.4 The Gateway Databases

The default configuration of the gateway is suitable for the majority of systems. If you are an advanced Netcool/Precision IP user, you can make adjustments to the configuration settings by modifying the values inserted into the gateway `config` database. This database contains the configuration settings that define the operation of the gateway. For example, you can modify the mappings that are used between Netcool/Precision IP and Netcool/OMNIbus and the filters that determine which events are processed.

The gateway databases are:

- `topoCache`
- `config`

The following sections describe the database configuration process and the gateway databases.

### Logging into the Gateway Databases Using the OQL Service Provider

To query the gateway databases, you must log into the databases using the OQL service provider and the service name `NcoGate`. The following example command logs in to the `NcoGate` service for the gateway running in the domain `Precision`, and using the username `admin`.

```
ncp_oql -domain Precision -service NcoGate -username admin
```

Enter the `admin` user password at the prompt.

### Applying Configuration Changes to the Gateway

Any configuration change made to the gateway can be applied while it is running by issuing a `SIGHUP` command to the gateway.

Enter the following command:

```
kill -HUP PID
```

Where *`PID`* with the process ID of the gateway.

The gateway checks the timestamp on its configuration file. If the configuration file has been modified, the gateway reads this file again in order to adapt to any configuration changes.

### The topoCache Database Schema

The `topoCache` database holds a copy of the MODEL topology database. This copy of the topology database is used to enrich event records with topology information.

The summary information for the `topoCache` database schema is shown in Table 67.

Table 67: topoCache Database Summary

| | |
|---|---|
| **Database name** | `topoCache` |
| **Defined in** | `NCHOME/etc/precision/NcoGateSchema.cfg` |
| **Fully qualified database table name** | `topoChache.entityByName` |

**Note:** `NCHOME` is the environment variable that contains the path to the Netcool Suite home directory. For information on how this environment variable varies with platform, see *Operating System Considerations* on page 9.

## The entityByName Table

The `topoCache.entityByName` table, described in Table 68, holds information about all the discovered network entities.

Table 68: topoCache.entityByName Table Description (1 of 3)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `ObjectId` | PRIMARY KEY<br>NOT NULL<br>UNIQUE | Long integer | The unique Object ID of the network entity, which is used to provide the value for the `NmosObjInst` field in the ObjectServer. |
| `EntityName` | PRIMARY KEY<br>NOT NULL<br>UNIQUE | Text | Unique descriptive name of a network entity. |
| `Address` | | List of text | List of OSI model layer 1 -7 addresses for the entity. |
| `Description` | | Text | Value of `sysDescr` MIB variable or other suitable description of the entity. |

Table 68: topoCache.entityByName Table Description (2 of 3)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| EntityType | Externally defined entityTypes data type | Integer | Element type of the entity. Possible values are:<br>• 0 - Unknown.<br>• 1 - Chassis.<br>• 2 - Interface.<br>• 3 - Logical interface.<br>• 4 - Vlan object.<br>• 5 - Card.<br>• 6 - PSU.<br>• 7 - Subnet.<br>• 8 - Module. |
| ClassName | | Text | Class name of network entity (if applicable). |
| EntityOID | | Text | Value of sysOID MIB variable of the entity. |
| Status | Externally defined status data type | Integer | Flag showing status of the network entity. |
| Security | | Text | Password to access network entity (if applicable). |
| RelatedTo | | List of text | List of entities that are connected to the network entity. |
| Contains | | List of text | List of elements or other containers contained in the current network entity. |
| UpwardConnections | | List of text | List of containers that contain this entity. |
| IsActive | Externally defined boolean data type | Integer | Flag indicating whether an Active Object Class is needed. |
| CreateTime | | Time | Creation time of network entity record in table. |
| ChangeTime | | Time | Time of last modification to the network entity record. |
| ActionType | Externally defined actions data type | Integer | The type of action associated with the record. |

Table 68: topoCache.entityByName Table Description (3 of 3)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `ExtraInfo` | Externally defined vblist data type | Object | A list of extra information. |
| `LingerTime` | NOT NULL<br>Default=3 | Integer | The linger time is used during rediscovery so that the new topology can be merged with the existing topology.<br>The value of `LingerTime` is decremented if the entity is not present in the new topology. The entity is only removed from the topology when the value of `LingerTime` reaches 0. |

## The config Database Schema

The `config` database is used to configure the way in which events are mapped between Netcool/OMNIbus and Netcool/Precision IP. The `config` database can also be used to define filters that restrict the events that are passed between Netcool/OMNIbus and Netcool/Precision IP.

The summary information for the `config` database schema is shown in Table 69.

Table 69: config Database Summary

| | |
|---|---|
| **Database name** | `config` |
| **Defined in** | `NCHOME/etc/precision/NcoGateSchema.cfg` |
| **Fully qualified database table names** | `config.defaults`<br>`config.eventMaps`<br>`config.nco2ncp`<br>`config.ncp2nco`<br>`config.failover`<br>`config.precedence` |

These tables are described in the following sections.

## The defaults Table

The `config.defaults` table, described in Table 70, contains general configuration data for the gateway.

Table 70: config.defaults Table Description

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `SyncCheckPeriod` | NOT NULL | Long integer | Specifies (in seconds) how frequently the gateway should check the synchronization of AMOS and the ObjectServer. |
| `IDUCFlushTime` | NOT NULL | Integer | The interval (in seconds) between IDUC (Insert Delete Update Control) flushes from the ObjectServer. The default is 1 second. |
| `NcoAuthUserName` | NOT NULL | Text | The username to use to access the ObjectServer when it is running in secure mode. |
| `NcoAuthPassword` | NOT NULL | Text | The password to use to access the ObjectServer when it is running in secure mode.<br>If necessary, you can encrypt this password using `nco_crypt` and enter the encrypted password in the configuration file. |
| `NcpServerEntity` | NOT NULL | Text | The IP address of the polling station. Use this field if the polling station is not in the topology or if it is necessary to pretend that the polling station is different from where it actually is. |

## The eventMaps Table

The `config.eventMaps` table, described in Table 71, holds information specific to each kind of Netcool/OMNIbus event that the gateway can process. The entries in this table indicate how an event from Netcool/OMNIbus with the specified Event ID should be handled by Netcool/Precision IP.

Table 71: config.eventMaps Table Description (1 of 2)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `EventMapName` | PRIMARY KEY<br>NOT NULL | Text | The name of the event map. This value is referenced by the config.precedence table, as described on page 152. |
| `ncpRuleName` | | Text | The AOC headrule to be applied to this event in AMOS when performing RCA. |

Table 71: config.eventMaps Table Description (2 of 2)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `ncpMainNode` | NOT NULL<br>Default=`0` | Integer | Indicates whether the events should be mapped to the polled entity (in the case of a ping fail) or the main node (which is usual in the case of traps). Possible values are:<br>• `0` - Map the event to the polled entity<br>• `1` - Map the event to the main node |
| `SendForRCA` | NOT NULL<br>Default=`1` | Integer | Indicates whether the event should be sent to AMOS to perform Root Cause Analysis (RCA). Possible values are:<br>• `0` - Do not send to AMOS. The event is enriched with topology data (if possible) and returned to the ObjectServer.<br>• `1` - Send to AMOS to perform RCA. |
| `PolledEntity` | NOT NULL | Text | A filter to be applied to the topology database to find the entity against which the event should be raised. |
| `PollerEntity` | NOT NULL | Text | A filter to be applied to determine the polling location for this event. This field may be empty if `SendForRCA` is set to `0`. |
| `ExtraInfoField` | | vblist | This field allows you to append data, that is specific to the event map, to the end of the `ExtraInfo` section of the `config.nco2ncp` entry.<br>This is an optional field. |

## The nco2ncp Table

The `config.nco2ncp` table, described in Table 72, is used to filter events being passed from Netcool/OMNIbus to Netcool/Precision IP.

Table 72: config.nco2ncpTable Description

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `EventFilter` | NOT NULL | Text | A filter that indicates which events should be processed by the gateway. Only events that match this filter are processed. |
| `EventFieldMap` | Externally defined vblist data type | Object | A mapping used to generate an OQL string using the fields in the incoming event and the topology record. |

The following example insert configures the filter and mappings for events passing from Netcool/OMNIbus to Netcool/Precision IP.

```
insert into config.nco2ncp
(
     EventFilter,
     EventFieldMap
)
values
(
     "LocalNodeAlias <> ''",
     {
          EntityName      = "eval(text, '&&EntityName')",
          ClassName       = "eval(text, '&&ClassName')",
          Description     = "eval(text, '&Summary')",
          EventName       = "eval(text, '&EventId')",
          RuleName        = "eval(text, '$RuleName')",
          RuleSet         = "TopologicalAlertCorrelation",
          EventType       = "eval(int,  '$EVENT')",
          Severity        = "eval(text, '&Severity')",
          AgentAddress    = "eval(text, '$AgentAddress')",
          CauseType       = "eval(int,  '$CAUSEUNKNOWN')",
          NcoSerial       = "eval(int,  '&Serial')",
          Occurred        = "eval(int,  '&Tally')",
          ExtraInfo       =
          {
               Precedence   = "eval(int, '$Precedence')",
               NmosObjInst  = "eval(long, '$MainNodeObjectId')",
               NmosSerial   = 0
          }
     }
);
```

In the example above, the ampersand (&) is used to navigate through the scope of the records being evaluated:

- A single ampersand accesses the Netcool/OMNIbus event record.
- A double ampersand accesses the Netcool/Precision IP topology record.

For more information about the eval statement, and the use of ampersands and scope in Netcool/Precision IP, see the *Netcool/Precision IP Discovery Configuration Guide*.

The example insert configures the gateway to:

- Only pass Netcool/OMNIbus events to Netcool/Precision IP where the `LocalNodeAlias` column of the ObjectServer record has been populated. This field provides a basis for looking up the event in the discovered topology.
- Perform the following mapping between the ObjectServer record, the topology database and the AMOS `mojo.events` database table:
  - Evaluate the value of the `EntityName` column in topology database and insert it into, or update, the `EntityName` column of the `mojo.events` database table.
  - Evaluate the value of the `ClassName` column in the topology database and insert it into, or update, the `ClassName` column of the `mojo.events` database table.
  - Evaluate the value of the `Summary` column in the ObjectServer record and insert it into, or update, the `Description` column of the `mojo.events` database table.
  - Insert the appropriate `EventName` using the ObjectServer's `EventId` into the `EventName` column of the `mojo.events` database table, or update the existing value.
  - Insert the appropriate `RuleName` (determined by the entries in the `config.eventMaps` table) into the `RuleName` column of the `mojo.events` database table, or update the existing value.
  - Set the `RuleSet` to `TopologicalAlertCorrelation`.
  - Insert the type of event into the `EventType` column of the `mojo.events` database table, or update the existing value.
  - Evaluate the value of the `Severity` column in the ObjectServer record and insert it into, or update, the `Severity` column of the `mojo.events` database table.
  - Insert the polling agent address into the `AgentAddress` column of the `mojo.events` database table, or update the existing value.
  - Set the `CauseType` of the event to `UNKNOWN`.
  - Evaluate the value of the `Serial` column in the ObjectServer record and insert it into, or update, the `NcoSerial` column of the `mojo.events` database table.
  - Evaluate the value of the `Tally` column in the ObjectServer record and insert it into, or update, the `Occurred` column of the `mojo.events` database table.

The gateway determines whether to insert a new record or update an existing one according to whether the ObjectServer sends the event as an insert using IDUC or as an update.

## The ncp2nco Table

The `config.ncp2nco` table, described in Table 73, is used to filter and map events being passed from Netcool/Precision IP to Netcool/OMNIbus.

Table 73: config.ncp2nco Table Description

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `EventFilter` | NOT NULL | Text | A filter that indicates which events should be processed by the gateway. Only events that match this filter are processed. |
| `EventFieldMap` | Externally defined vblist data type | Object | An object containing the mapping used to generate an OQL update string using the fields in the incoming event and the topology record. |

The following example insert configures the filter and mappings for events passing from Netcool/Precision IP to Netcool/OMNIbus.

```
insert into config.ncp2nco
(
     EventFilter,
     EventFieldMap
)
values
(
     "ActionType <> 2",
     {
          Severity       = "eval(int,  '&Severity')",
          NmosObjInst    = "eval(int,  '&ExtraInfo->NmosObjInst')",
          NmosSerial     = "eval(text, '&ExtraInfo->NmosSerial')",
          NmosCauseType = "eval(int,  '&CauseType')"
          //IfDescr      = "eval(text,  '&&ExtraInfo->m_IfDescr')"
     }
);
```

The foregoing example insert configures the gateway to:

- Only pass Netcool/Precision IP events to Netcool/OMNIbus where the `ActionType` column of the database record in the AMOS `mojo.events` table is not set to 2. This ensures only new events and updates are sent. Deletions are not sent.
- Perform the following mapping between the AMOS database and the ObjectServer record:
  - Evaluate the value of the `mojo.events.Severity` column and use it to update the ObjectServer `Severity` column.
  - Evaluate the value of the `NmosObjInst` field in `mojo.events.ExtraInfo` column and use it to update the ObjectServer `NmosObjInst` column.

  - Evaluate the value of the `NmosSerial` field in `mojo.events.ExtraInfo` column and use it to update the ObjectServer `NmosSerial` column.
  - Evaluate the value of the `mojo.events.CauseType` column and use it to update the ObjectServer `NmosCauseType` column.
- The line which has been commented in this sample fragment of code shows an example of outgoing topology enrichment and assumes that there is an ObjectServer field in `alerts.status` named `IfDescr`.

The `mojo.events.NcoSerial` column from the Netcool/Precision IP event is used to determine which ObjectServer event is updated.

The ObjectServer update generated by the configuration described here is in the following format (the items in angle brackets are replaced with the value evaluated from the specified column in the Netcool/Precision IP event).

```
update alerts.status
     set       Severity = <Severity>,
               NmosObjInst = <ExtraInfo->NmosObjInst>,
               NmosSerial = <ExtraInfo->NmosSerial>,
               NmosCauseType = <CauseType>,
               Where Serial = <NcoSerial>;
```

## The failover Table

The `config.failover` table contains the failover configuration and current failover state of the Event Gateway component. The columns are described in Table 74.

Table 74: config.failover Table Description

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `BackupGateway` | NOT NULL | Boolean | This value is true if the Event Gateway is started using the `-backup` command line option. Possible values are:<br>• `0` - Not configured as the backup system<br>• `1` - Configured as the backup system |
| `Failedover` | NOT NULL | Boolean | The failover state. Possible values are:<br>• `0` - Not in a failover state<br>• `1` - In a failover state |

## The precedence Table

The `config.precedence` table contains the information necessary to determine which event has precedence when multiple events occur on the same interface. The columns are described in Table 75.

Table 75: config.precedence Table Description

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `Precedence` | NOT NULL | Integer | A number used by AMOS when there are multiple events on the same entity within the network topology. The number is used to determine which of the events has precedence and therefore suppresses the other event on that interface. For example, a link down event has a higher `Precedence` value than a ping fail event and therefore the link down will suppress the ping fail event on that interface.<br><br>The following `Precedence` values have special meanings:<br>• `0` - An event with this `Precedence` value cannot become a root cause event. If an event's `Precedence` value is set to `0`, then this event can only become a symptom event or can be marked as cause unknown.<br>• `10000` and greater - An event with a `Precedence` value greater than or equal to `10000` cannot become a symptom event. It can only become a root cause event or be marked as cause unknown. |
| `EventMapName` | NOT NULL | Text | The name of the event map from the `config.eventMaps` table that is used to process the event with a matching `EventId`. For more information on the `config.eventMaps` table, see *The eventMaps Table* on page 146. |
| `NcoEventId` | PRIMARY KEY<br><br>NOT NULL | Text | Provides the mapping from the `EventId` in the `alerts.status` table of the Netcool/OMNIbus ObjectServer to the values of `Precedence` and `EventMapName` defined in this table. |

# 6.5 Sending Events to AMOS

The gateway sends events to AMOS by inserting events into the `mojo.events` database in AMOS. The following section provides an example of an insert to `mojo.events`.

## Example Insert

The following example shows an insert to `mojo.events`.

```
insert into mojo.events
(
               EntityName,
               ClassName,
               Description,
               EventName,
               RuleSet,
               RuleName,
               EventType,
               Severity,
               AgentAddress,
               CauseType,
               NcoSerial,
               Occurred,
               ExtraInfo
     )
values
(
               <&&EntityName>,
               <&&ClassName>,
               <&Summary>,
               <$EventId>,
               <$RuleName>,
               "TopologicalAlertCorrelation",
               <$EVENT>,
               <&Severity>,
               <$AgentAddress>,
               <$CAUSEUNKNOWN>,
               <&Serial>,
               <&Tally>,
               {
                    Precedence = <$Precedence>,
                    NmosObjInst = <$MainNodeObjectId>,
                    NmosSend = 0
               }
);
```

This example above uses the following rules:

- The items in angle brackets preceded by `&` are replaced with the value calculated based on the specified column in the ObjectServer event.
- The items in angle brackets preceded by `&&` are replaced with the value calculated based on the specified column in the topology database.
- The items in angle brackets preceded by `$` are determined from the configuration of the `config.eventMaps` table, with the exception of `$EVENT` and `$CAUSEUNKNOWN`, which represent enumerated constants.

# Chapter 7: Root Cause Analysis

This chapter describes AMOS, the root cause analysis component of Netcool/Precision IP. It also describes the AMOS databases and the event correlation rules in the AOC extensions.

This chapter contains the following sections:

## 7.1 Introduction to Root Cause Analysis

Root cause analysis is the process of determining the root cause of one or more alerts. A failure situation on the network usually generates multiple alerts. This is because a failure condition on one device may render other devices inaccessible. Polling agents are unable to access the device which has the failure condition. In addition, polling agents are also unable to access other devices rendered inaccessible by the error on the original device. Events are generated indicating that all of these devices are inaccessible. These events are sent by the MONITOR probe to the Netcool/OMNIbus ObjectServer.

In addition to the events sent by the MONITOR probe to the Netcool/OMNIbus ObjectServer, Netcool/Precision IP can perform root cause analysis on any event received from the ObjectServer. This includes events which arrive at the ObjectServer from Netcool/OMNIbus probes.

Netcool/Precision IP performs root cause analysis by correlating event information with topology information. This enables Netcool/Precision IP to determine which devices are temporarily inaccessible due to other network failures. Alerts on devices which are temporarily inaccessible are suppressed, that is, shown as symptoms of the original, root cause alert. When the root cause alert is resolved, these events remain in the ObjectServer and depending on whether they are real problems they may be cleared at a later stage, or they may, in turn, become root causes of other events.

The Netcool/Precision IP component which performs root cause analysis is called AMOS.

## Architecture of Root Cause Analysis

Figure 16 shows how the AMOS component applies topology-based RCA to events held in the ObjectServer.

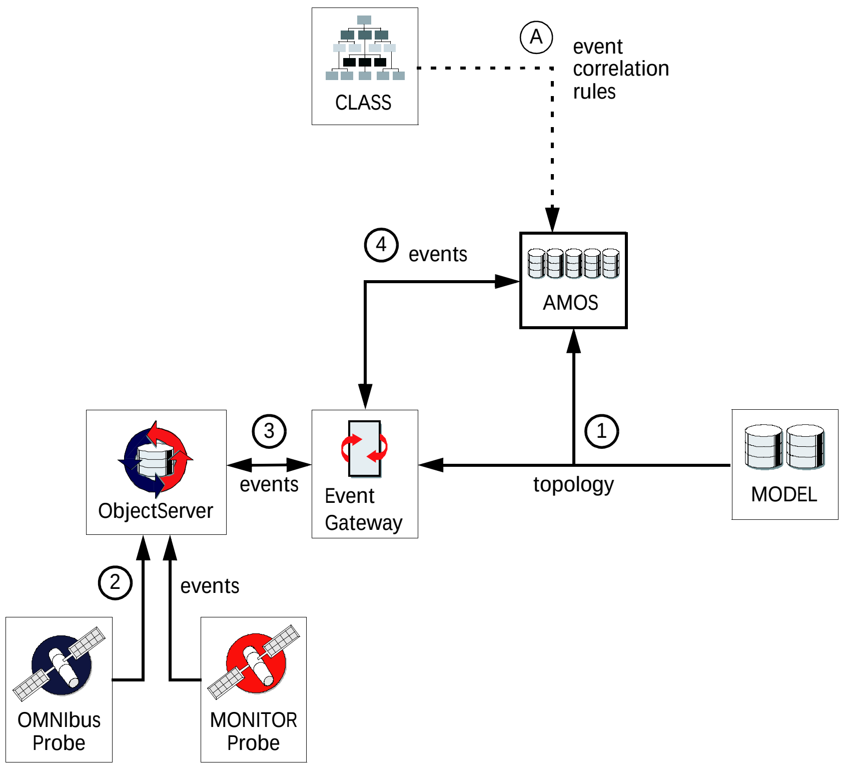


Figure 16: Root Cause Analysis Architecture

On startup, AMOS downloads event correlation rules from CLASS, the AOC management system. This is shown as item A in Figure 16. AMOS performs root cause analysis by correlating event information with topology information, using event correlation rules specified in the Active Object Classes (AOCs).

The other steps enumerated in Figure 16 are described below:

1. MODEL sends topology information to AMOS. This occurs on startup of AMOS but also occurs if the topology in MODEL is updated due to a new discovery on the network. AMOS stores this data in its `topoCache.entityByName` database table. This database table is described in *topoCache.entityByName Entity Database Table* on page 170.

   MODEL also sends topology information to the Event Gateway so that the Event Gateway can enrich events from the Object Server with topology information.

2. The ObjectServer receives events from Netcool/OMNIbus probes and from the MONITOR probe.
3. The ObjectServer performs event correlation and deduplication on all the events it stores. The ObjectServer sends a subset of its events to the Event Gateway. These events can be enriched with topology information and sent back to the ObjectServer.
4. The Event Gateway sends events to AMOS. AMOS performs RCA on these events and sends them back to the Event Gateway, which in turn sends them back to the ObjectServer. The ObjectServer is the master event repository.

   Prior to performing RCA on these events, AMOS stores the events in its `mojo.events` database table. This database table is described in *mojo.events Events Database Table* on page 168.

## Mechanism of Root Cause Analysis

AMOS performs root cause analysis by correlating event information with topology information.

- The event information is held in the AMOS `mojo.events` database table.
- The topology information is held in the AMOS `topoCache.entityByName` database table.

AMOS uses event correlation rules to perform root cause analysis. Using these rules, AMOS is able to analyze an event on one device and calculate the impact on each connected device in the network topology.

AMOS performs root cause analysis based on a number of considerations. These include the following:

- Containment associated with the network devices in the topology.
- Connectivity between network devices in the topology.
- Network infrastructure faults, such as cable breaks.

The section which follows provides practical examples of how AMOS performs root cause analysis. Detailed information on the structure of the event correlation rules can be found in *The Event Correlation Rules* on page 173.

## Examples of Root Cause Analysis

The examples in this section show how AMOS performs RCA based on these considerations for different types of network device and interface.

The terms *downstream* and *upstream* are used in many of the examples in this section. These terms consider the perspective of the polling station.

- The term *downstream* refers to a location on the network topologically more distant from the polling station but on the same physical path as a second location. For example, in Figure 17, device B is downstream of device A.
- The term *upstream* refers to a location on the network topologically closer to the polling station but on the same physical path as a second location. For example, in Figure 17, device A is upstream of device B.

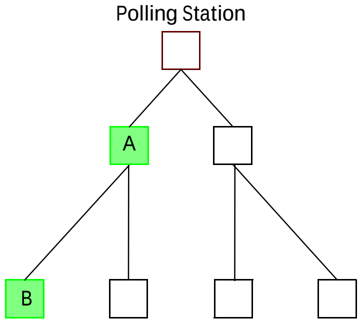


Figure 17: Downstream and Upstream Devices

In complex networks the distance of devices from the polling station changes as devices are deactivated. This in turn has an impact on which devices are upstream or downstream.

**Note:** The examples of RCA shown in this chapter are for illustrative purposes only. RCA in large networks is extremely complex and the examples shown here are only meant to show the principles which RCA uses.

## Chassis Devices and Loopback Interfaces

Failures on chassis devices (main node devices) are given priority. AMOS assumes that if a chassis has failed, then, in many cases, the root cause for other failures originates in the chassis. Chassis failures suppress failures on contained interfaces, connected interfaces and on downstream chassis devices.

The loopback interface has a special function within a chassis device, whether router or switch. A loopback interface always has an IP address, which corresponds to the IP address of the device. This means that the loopback interface represents the whole chassis and can be polled individually. It also means that failures on the loopback interface suppress failures on connected and contained entities in exactly the same way as failures on chassis devices.

The examples below provide examples of each of these connections and show which failures are flagged as root cause and which are suppressed:

- A failure on a chassis device suppresses failures on interfaces contained within that chassis, as shown in *Example Contained Interfaces* on page 160.
- A failure on a chassis device suppresses failures on directly connected interfaces, whether upstream or downstream of the chassis device, as shown in *Example Connected Interfaces* on page 161.
- A chassis device contains one or more entities. A failure on the chassis device suppresses failures on entities directly connected to any of the entities contained within that chassis device, as shown in *Example Entities Connected To A Contained Entity* on page 161.
- A failure on a chassis device suppresses failures on all downstream chassis devices, as shown in *Example Downstream Chassis Devices* on page 163.

## Example Contained Interfaces

A chassis failure suppresses all failures on interfaces contained within that chassis. In Figure 18, failure on chassis device A suppresses failures on interfaces b, c and d which are all contained within chassis device A.

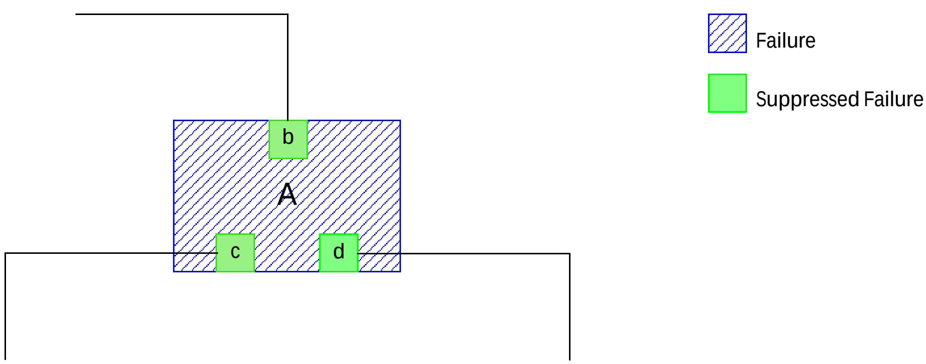


Figure 18: Chassis Failure Suppresses Failures On Contained Interfaces

## Example Connected Interfaces

A chassis failure suppresses all failures on interfaces connected to that chassis device. Failures are suppressed on both upstream and downstream interfaces. In Figure 19, device A suppresses failures on upstream interface b and on downstream interfaces c and d.

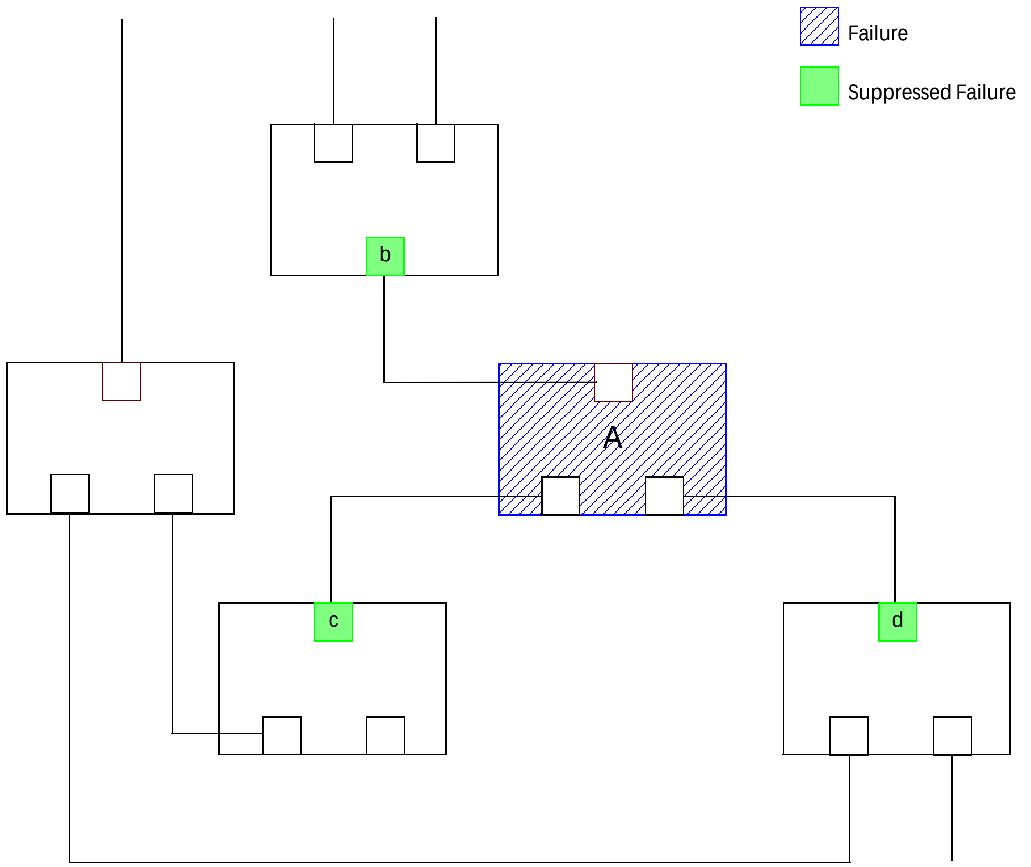


Figure 19: Chassis Failure Suppresses Failures On Connected Interfaces

## Example Entities Connected To A Contained Entity

A chassis device may contain one or more entities. Examples of entities which can be contained within a chassis device are VLANs, cards and virtual routers. A contained entity may have one or more interfaces. For more information on containment within the Netcool/Precision IP network model, see the *Netcool/Precision IP Discovery Configuration Guide*.

A failure on the chassis device suppresses failures on entities directly connected to any of the entities contained within that chassis device. In Figure 20, entity B is contained within chassis device A. A failure on chassis device A suppresses a failure on interface d on device D and interface e on device E. Both interfaces d and e are directly connected to entity B.

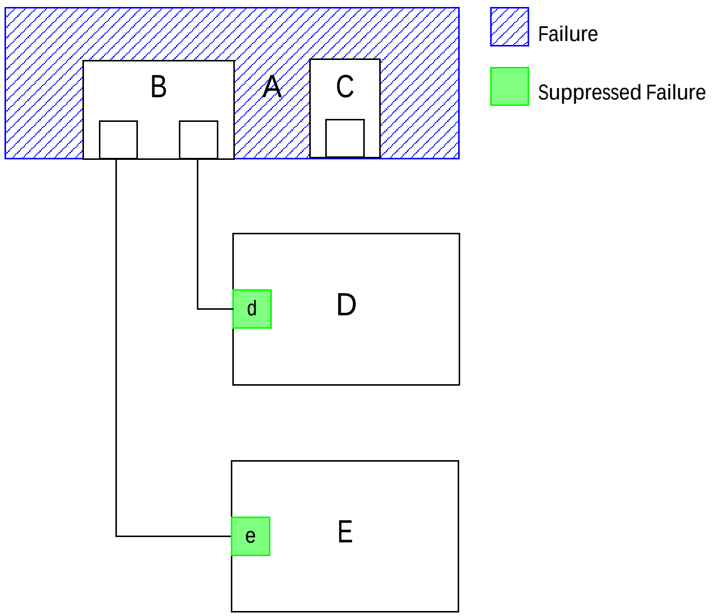


Figure 20: Chassis Failure Suppresses Failures Devices Connected to Contained Entities

## Example Downstream Chassis Devices

A failure on a chassis device suppresses failures on all chassis devices downstream of the chassis where the failure occurred. In Figure 21, failure on chassis device A suppresses failures on chassis devices B, C and D which are all downstream of chassis device A.

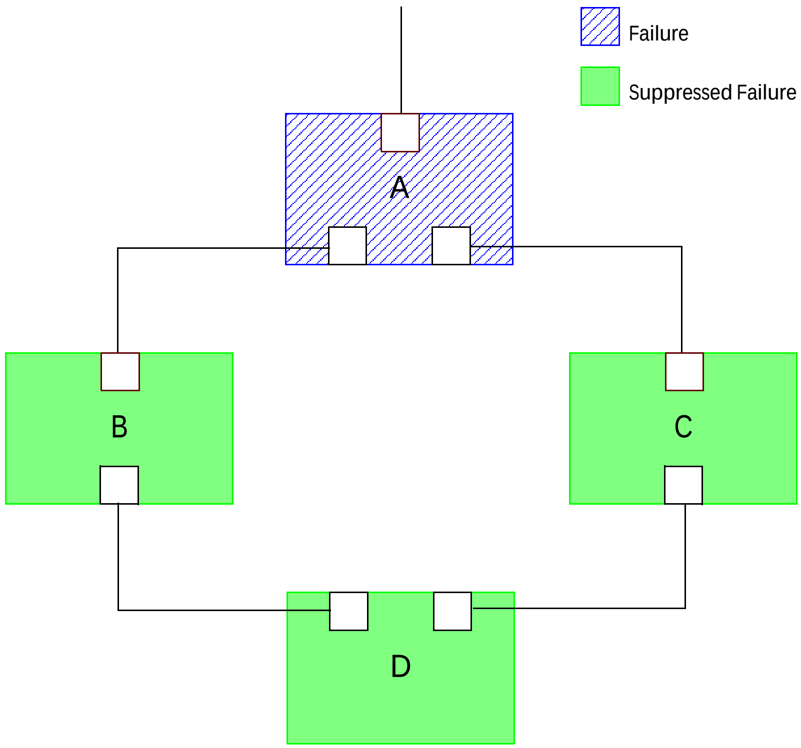


Figure 21: Chassis Failure Suppresses Failures On Downstream Entities

## Interfaces

Standard physical interface failures are not capable of suppressing interface failures on downstream entities.

A standard interface failure can only suppress a second physical interface failure if the two interfaces are directly connected. The interface whose suppression rule fires first, suppresses the other interface. Suppression of one interface failure by a second interface failure can only occur if these interface failures are not already being suppressed by a chassis failure or a loopback interface failure.

A physical interface can contain multiple logical interfaces. A failure on a physical interface can suppress failures on its related logical interfaces. The physical interface can suppress its related logical interface even if there is connectivity between that logical interface and an external neighbor. A suppressed physical interface can pass on the details of its suppressor entity to the events on its associated logical interfaces.

- *Example Directly Connected Interface* on page 164 illustrates how an interface failure can suppress a more recent failure on a directly connected interface.
- *Example Related Logical Interface* on page 164 illustrates how a physical interface failure can suppress a failure on related logical interfaces.

## Example Directly Connected Interface

A standard physical interface failure suppresses a second physical interface failure if the two interfaces are directly connected. The suppressing interface failure must be older than the failure to be suppressed. Suppression of one interface failure by a second interface failure can only occur if these interface failures are not already being suppressed by a chassis failure or a loopback interface failure. In Figure 22, failure on interface a suppresses the more recent failure on directly connected interface b.

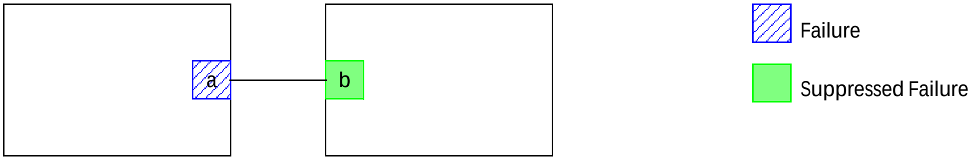


Figure 22: Interface Failure Suppresses More Recent Failure On Directly Connected Neighbor Interface

## Example Related Logical Interface

A failure on a physical interface suppresses failures on its related logical interfaces. In Figure 23, failure on physical interface a suppresses failures on contained logical interfaces b and c.


Figure 23: Physical Interface Failure Suppresses Failures on Contained Logical Interfaces

## Example Downstream Suppression For Interfaces at The Edge of a Network

A failure on a logical or physical interface that is the sole connection between other entities and the network will suppress failures in the downstream entities. In Figure 24, failure on interface d in device A suppresses failures on devices B, C and D and their interfaces.

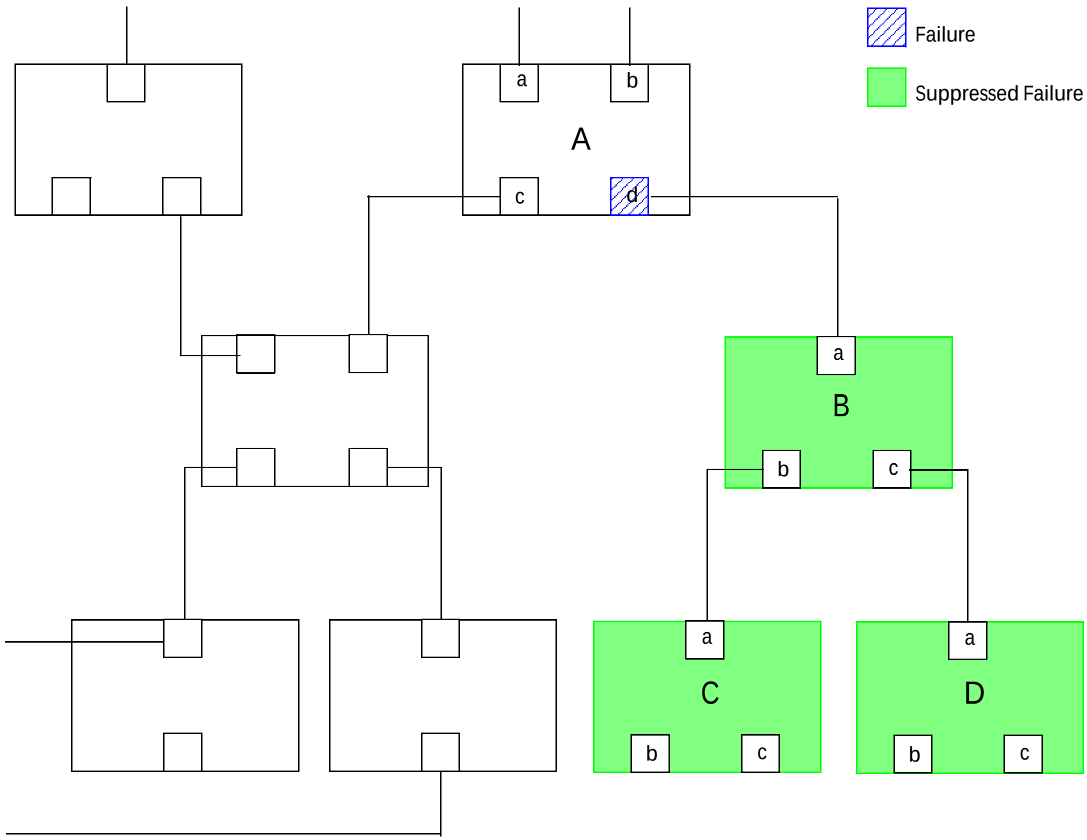


Figure 24: Interface Failure Suppresses More Recent Failure On Directly Connected Neighbor Interface

# 7.2 Starting AMOS

Micromuse recommends that AMOS is started using the domain process controller CTRL. The use of CTRL to automatically manage processes is described in the *Netcool/Precision IP Discovery Configuration Guide*.

On Microsoft Windows, Netcool/Precision IP components can be run as processes or as Windows services. Components run as processes are started from a command prompt in the same way as on UNIX platforms. For more information on running components as Windows services, see the *Netcool/Precision IP Discovery Configuration Guide*.

**Warning:** If you are using Netcool/Precision IP with failover, you must start AMOS using CTRL. The CTRL process checks the status of the AMOS component and uses this information to generate the Health Check events used by the failover process. For more information on failover, see the *Netcool/Precision IP Installation and Deployment Guide*.

## Prerequisites for Starting AMOS

The following Precision Server processes need to be running before starting AMOS:

- CTRL, the domain process controller. CTRL may be configured to start other Precision Server processes. CTRL must be running in order to start the subprocesses of DISCO and MONITOR.
- DISCO, the discovery process controller. DISCO must have successfully completed a network discovery, and the network topology and containment model must have been passed to MODEL.
- MODEL, the network topology distributor. AMOS loads topology information from MODEL into the AMOS entity database.
- CLASS, the AOC management system. AMOS downloads the fault rules from CLASS.
- MONITOR, the polling process controller. MONITOR launches and manages the polling agents.

**Note:** You can also use Netcool/OMNIbus probes. Netcool/OMNIbus probes work seamlessly with Netcool/Precision IP without any need for configuration if you have the Netcool/Knowledge Library installed. The Netcool/Knowledge Library is available with your Netcool/OMNIbus installation. It is also available as a download on the Micromuse Support Site.

- STORE, the persistent storage engine. When AMOS starts, it loads information from STORE into the AMOS events database.

## Manually Starting AMOS

To manually start AMOS, run the `ncp_f_amos` command.

The command line options for `ncp_f_amos` are:

```
ncp_f_amos -domain DOMAIN_NAME [-latency LATENCY] [-debug DEBUG] [-help] [-version]
```

Table 76 describes the command line options for `ncp_f_amos`.

Table 76: ncp_f_amos Command Line Options

| Option | Explanation |
|---|---|
| `-domain` *DOMAIN_NAME* | The name of the domain under which to run AMOS. |
| `-latency` *LATENCY* | The maximum time in milliseconds that AMOS waits for the to connect to another process using the messaging bus. This option is useful for large and busy networks where the default settings can cause the process to assume that there is a problem when in fact the communication delay is a result of network traffic. |
| `-debug` *DEBUG* | The level of debugging output (`1-4`, where `4` represents the most detailed output). |
| `-help` | Prints out a synopsis of all command line options for `ncp_f_amos` then exits. |
| `-version` | Prints the version number of `ncp_f_amos` then exits. |

## Process Flow in AMOS

The following steps describe the processes that occur when AMOS is launched:

1. The following information is transferred:
   - Event information is loaded from STORE into the AMOS events database.
   - The network topology is downloaded from MODEL into the AMOS entity database.
   - The fault rules, used for event correlation, are downloaded from CLASS.
2. The Netcool/Precision IP gateway process synchronizes the events in the AMOS database with those in the ObjectServer. AMOS continues to monitor the events passing through the Event Gateway.
3. AMOS listens for updates from MODEL and CLASS allowing it to maintain an up to date topology model and set of fault rules.

# 7.3 AMOS Databases

AMOS uses two main database tables to store information:

- `mojo.events` - stores event information.
- `topoCache.entityByName` - stores entity information.

In addition to these database tables, the `translations` databases are defined in `AmosSchema.cfg`. These are used internally and should not be modified by the user. AMOS also stores class-based information in an internal database.

## mojo.events Events Database Table

The `mojo` database is defined in `NCHOME/etc/precision/AmosSchema.cfg`. It contains the table `mojo.events`.

**Note:** `NCHOME` is the environment variable that contains the path to the Netcool Suite home directory. For information on how this environment variable varies with platform, see *Operating System Considerations* on page 9.

The `mojo.events` table, described in Table 77, stores all of the event records sent for root cause analysis by the Event Gateway. When AMOS is launched, event information is also read from STORE into this database. The column names of the records are used in many of the conditional filters when constructing event correlation methods.

Table 77: mojo.events Table Description (1 of 3)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `EventId` | PRIMARY KEY<br>NOT NULL<br>UNIQUE | Long Integer | The event ID. |
| `EntityName` | PRIMARY KEY<br>NOT NULL | Text | The name of the associated entity. |
| `ClassName` | NOT NULL | Text | The name of the associated Active Object Class. |
| `NcoSerial` | | Int | The serial number of the event as assigned by the ObjectServer. |
| `Description` | | Text | A textual description of the event. |
| `EventName` | | Varchar (256) | The name of the event which should be made the `EventID` field in the ObjectServer. |

Table 77: mojo.events Table Description (2 of 3)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| RuleSet | | Text | Set of rules that are used together to handle a particular occurrence of an event. |
| RuleName | | Text | The name of the head rule for this event. |
| EventType | Externally defined eventType data type | Integer | Type of event. Possible value are:<br>• 0 - Event<br>• 1 - Data<br>• 2 - Alert |
| Severity | PRIMARY KEY<br>NOT NULL Externally defined severity data type | Integer | The OSI severity code. Possible values are:<br>• 0 - Clear<br>• 1 - Unknown<br>• 2 - Warning<br>• 3 - Minor<br>• 4 - Major<br>• 5 - Critical<br>• 6 - No severity |
| Contact | | Text | The contact group responsible for the device that generated the event. |
| AssignedTo | | Text | The person the event has been assigned to. |
| Acknowledged | | Integer of boolean type | Denotes whether the event has been unacknowledged or acknowledged. Possible values are:<br>• 0 - Unacknowledged<br>• 1 - Acknowledged |
| Location | | Text | Location of the device that generated the event. |
| CorrelatedId | | List of Long Integers | List of associated event IDs. |
| EventGroupId | | Long Integer | List of associated events. |
| ActionGlyph | | Text | An alert display glyph (icon). |
| CreateTime | TIMESTAMP | Long Integer | Time of the first occurrence of the event. |
| ChangeTime | TIMESTAMP | Long Integer | Time of the last occurrence of the event. |
| Occurred | COUNTER | Integer | Number of identical events that have occurred. |

Table 77: mojo.events Table Description (3 of 3)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| CauseType | Externally defined causeType data type | Integer | The type of alert. Possible values are:<br>• 0 - Unknown<br>• 1 - Root<br>• 2 - Symptom |
| ActionType | Externally defined actionType data type | Integer | The type of action the event represents—a new event, updated event or deleted event. Possible values are:<br>• 0 - New<br>• 1 - Update<br>• 2 - Delete |
| InternalAction | Externally defined internalAction data type | Integer | The internal action associated with the event. Possible values are:<br>• 0 - None<br>• 1 - Acknowledged<br>• 2 - Unacknowledged<br>• 3 - Assign<br>• 4 - Deassign<br>• 5 - Tool<br>• 6 - Clear<br>• 7 - Clear chain<br>• 8 - Clear with |
| AgentAddress | | Text | polling agent location (IP address). |
| Dist | | Int | Indicates whether the event should be broadcast using ncp_dist. |
| AlertType | | Text | Description of the type of alert or event, for example, topology alert or temperature alert. |
| ExtraInfo | Externally defined varbind list | Object | List of additional information. |

## topoCache.entityByName Entity Database Table

The topoCache database is defined in NCHOME/etc/precision/AmosSchema.cfg. It contains the table topoCache.entityByName.

The topoCache.entityByName table, described in Table 78, contains all information relating to network entities, their containment and connections. When AMOS is launched, it reads entity information from MODEL into this database table. This information allows AMOS to differentiate objects of the same class using conditional tests and filters in the poll definitions or event correlation methods.

Table 78: topoCache.entityByName Table Description (1 of 2)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| ObjectId | PRIMARY KEY<br>NOT NULL<br>UNIQUE | Long Integer | Unique Object ID of the network entity. |
| EntityName | PRIMARY KEY<br>NOT NULL<br>UNIQUE | Text | Unique descriptive name of a network entity. |
| Address | | List of Text data type | List of OSI model Layer 1-7 addresses for the entity. |
| Description | | Text | Value of sysDescr MIB variable or other suitable description of the entity. |
| EntityType | Externally defined entityType data type | Integer | Element type of the object. Possible values are:<br>• 0 - Unknown<br>• 1 - Chassis<br>• 2 - Interface<br>• 3 - Logical interface<br>• 4 - VLAN object<br>• 5 - Card<br>• 6 - PSU<br>• 7 - Subnet<br>• 8 - Module |
| ClassName | | Text | The associated Active Object Class (if applicable). |
| EntityOID | | Text | Value of sysOID MIB variable of the object. |
| ExtraInfo | Externally defined varbind list | Object | List of additional information. |
| Status | Externally defined status data type | Integer | Flag showing status of the network object. |
| Security | | Text | Password to access network entity (if applicable). |

Table 78: topoCache.entityByName Table Description (2 of 2)

| Column Name | Constraints | Data Type | Description |
|---|---|---|---|
| `RelatedTo` | | List of Text data type | List of connections to the network object. |
| `Contains` | | List of Text data type | List of elements or other containers contained by this object. |
| `UpwardConnections` | | List of Text data type | The name of the physical container to which the network object belongs. |
| `IsActive` | | Integer of boolean type | Flag indicating whether an Active Object Class is needed. |
| `CreateTime` | | Time | Creation time of network entity record in table. |
| `ChangeTime` | | Time | Time of last modification to the network entity record. |
| `ActionType` | Externally defined actionType data type | Integer | The type of action the event represents—a new event, updated event or deleted event. Possible values are:<br>• `0` - New<br>• `1` - Update<br>• `2` - Delete |
| `LingerTime` | NOT NULL | Integer | The number of discoveries which have to be run before an entity which has an entry in this table but which has not since been discovered is removed from the table. An entity that can not longer be discovered indicates the device may have been removed from the network. |

# 7.4 The Event Correlation Rules

When AMOS is launched, it downloads the event rules for each AOC stored in CLASS. The event rules are on standby until triggered. An event rule can be triggered by a timer in the rule itself, by an incoming event, or by another event rule. If an event rule is triggered by another event rule, this is called rule chaining.

The AOC files are stored in the directory `NCHOME/precision/aoc` and the event rules file are stored below the directory `NCHOME/precision/aoc/rca_rules`. The rules are listed in the AOC file and can be located by searching for the section:

```
extension for Fault = {
  rules = [
```

The format of each rule entry is:

```
"directory_name/rule_name.rule",
```

Where *`directory_name`* is the directory containing the rule file and *`rule_name`* is the name of the rule.

To edit an event correlation rule, open the rule file and make changes using a text editor.

## Inherited Rules

When an AOC is downloaded it inherits event correlation rules from its parent class. You can create an event correlation rule in a class that overrides the inherited rule. This does *not* change that rule in the parent class.

## Rule Chaining

Rule chaining is a way of using event rules by joining them together in a chain. Rules are chained using one of the following attributes:

```
evaluate_after = "rulename",
evaluate_after_fired = "rulename",
evaluate_after_not_fired = "rulename",
evaluate_when = "rulename"
```

These attributes are described in *Event Rule Attributes* on page 174. The example in Figure 25 shows the rule chain when the attribute `evaluate_when_fired="NonTimedAlertTransition"` is added to the rule `EventEntityToAlert`.

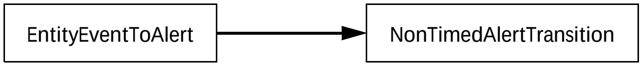


Figure 25: Example of Rule Chaining

### Multiple Parenting

Multiple parenting refers to chaining two different event rules from the same parent in the same way. The rules do this by creating one rule which triggers two other rules using the same `evaluate_` attribute.

## Event Rule Attributes

The event correlation rule is created using attributes. The top level attributes are:

- *rulename*
- *ruleset*
- *firing_condition*
- *execute_location*
- *execute_rule*

Many of these attributes have multiple levels of sub-attributes, as shown in Figure 26.

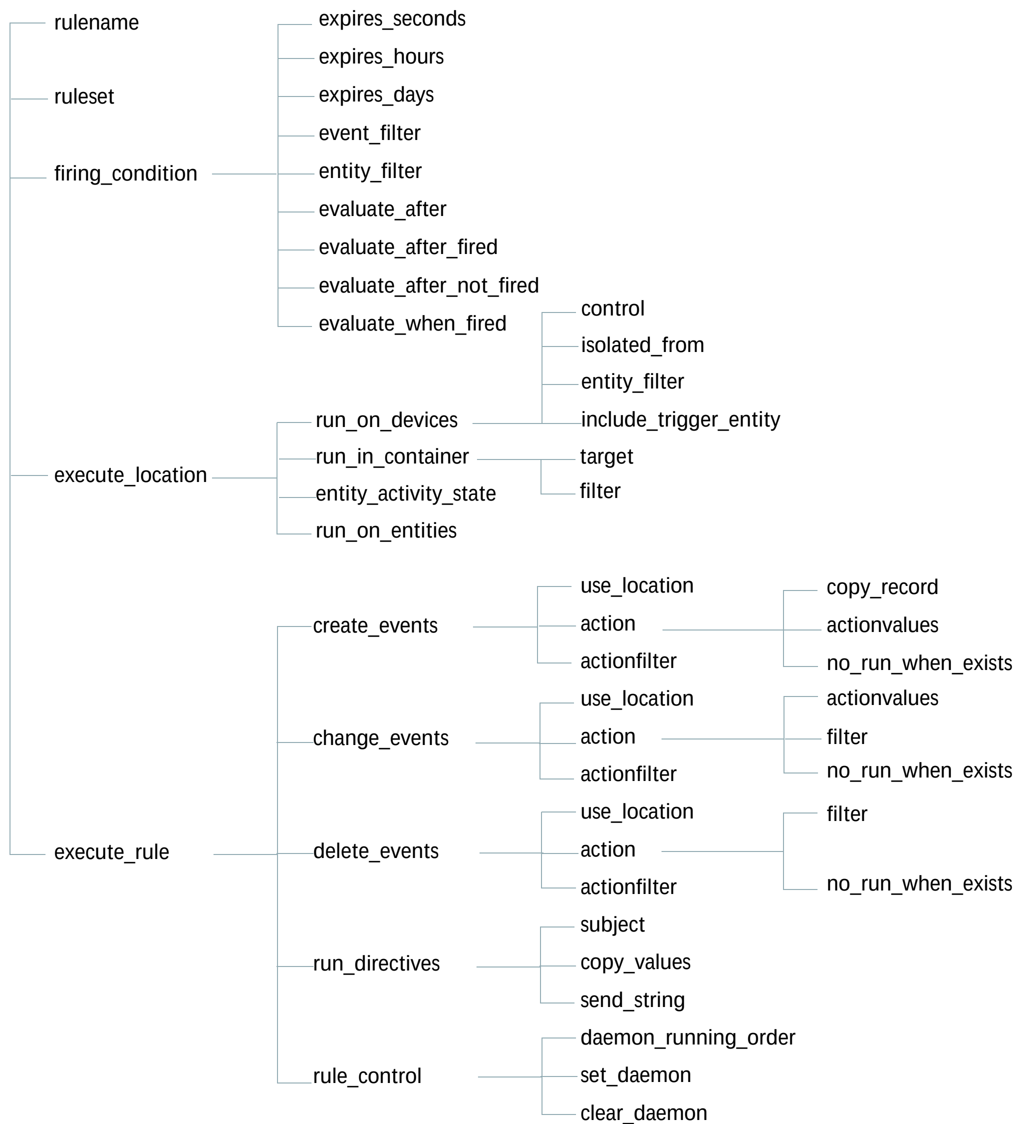


Figure 26: Event Correlation Rule Structure

The event correlation rule attributes are described in the following sections.

# rulename

The `rulename` attribute declares the name of the current event rule. Any string of text is accepted as a valid entry and must be unique within the specified AOC. An example `rulename` is given below:

```
rulename = 'pingFailEventToAlert',
```

The `rulename` attribute has no sub-attributes and must be followed by the `ruleset` attribute.

# ruleset

The `ruleset` attribute is a conditional test. The rule only fires if the ruleset of the incoming event matches the named ruleset. In the following example, the rule only fires if the event has the ruleset `TopologicalAlertCorrelation`.

```
ruleset = 'TopologicalAlertCorrelation',
```

The name of the ruleset is case-sensitive and must be enclosed in single quotes. The `ruleset` attribute has no sub-attributes and must be followed by the `firing_condition` attribute.

# firing_condition

The `firing_condition` attribute controls when the rule runs, and also controls rule chaining.

The sub-attributes are described in Table 79.

Table 79: firing_condition Sub-Attribute Descriptions (1 of 2)

| Sub-Attribute | Description |
|---|---|
| `expires_seconds` | Defines the interval at which the rule runs, in seconds. |
| `expires_hours` | Defines the interval at which the rule runs, in hours. |
| `expires_days` | Defines the interval at which the rule runs, in days. |
| `event_filter` | Applies a filter to the event which triggered the event rule. The filter can use fields from the AMOS Events database, as well as logical operators and the eval statement. |
| `entity_filter` | Gathers information about the device which triggered the event rule from AMOS entity database. Then applies a filter to this information. |
| `evaluate_after` | Runs the current event rule (the one in which this attribute appears) on the AMOS Events database when the specified event rule (the one named in this attribute) has been evaluated, regardless of whether or not it has fired. The example below runs the present event rule after the event rule `PingFailToAlert`: |

Table 79: firing_condition Sub-Attribute Descriptions (2 of 2)

| Sub-Attribute | Description |
|---|---|
| evaluate_after_fired | Runs the current rule on the AMOS Events database when the specified rule has fired. |
| evaluate_after_not_fired | Runs the current rule on the AMOS Events database when the specified rule has been evaluated and not fired. |
| evaluate_when_fired | Takes the output of the specified rule after it has fired, and uses it as the input of the current rule. |
| firing_policy | These sub-attributes are for internal use only. |

The `expires_` attributes define an event rule as timed. If they are all set to zero, the event rule is untimed and the `_filter` attributes and the `evaluate_` attributes define the trigger condition. If the rule is not timed and there are no filters, the rule is applied to all incoming events.

**Note:** A rule is said to have fired if all the filters in the firing condition have been passed and processing has moved on to the `execute_rule` section. The `execute_rule` section is the part of the rule which takes action on the event. If one or more of the filters have failed, the rule is said to have been evaluated but not fired.

The `evaluate_` attributes provide the ability to chain rules together. Rule chaining allows a series of simple event rules to be combined to form more complex event rules. For example, when one event rule finishes its processing, its output can be processed by a second event rule. This improves efficiency in determining root cause and reduces the number of event rules required in the AOCs.

## firing_condition Examples

The following `firing_condition` attribute is taken from one of the default event rules files.

```
firing_condition = {
   expires_seconds = 0,
   expires_hours   = 0,
   expires_days    = 0,
   //-----------------------------------------------
   event_filter=    "
        InternalAction=eval(int,'$NONE') AND
        EventType=eval(int,'$EVENT') AND
        ActionType=eval(int,'$NEW') AND
        Severity>eval(int,'$CLEAR')
        ",
   entity_filter= "",
   evaluate_after = "",
   evaluate_after_fired = "",
   evaluate_after_not_fired = "",
```

```
   evaluate_when_fired = "",
},
```

You can make the following changes to your `firing_condition` attribute to fire the rule every 30 seconds.

```
firing_condition = {
   expires_seconds = 30,
   expires_hours = 0,
   expires_days = 0,
```

You can make the following changes to your `firing_condition` attribute to create a filter that requires the following to be true:

- The event is a new event from the polling process, (`EventType = EVENT` and `ActionType = NEW`).
- The severity of the event is greater than `UNKNOWN`.

```
event_filter = "
   EventType=eval(int,'$EVENT') AND
   Actiontype=eval(int,'$NEW') AND
   Severity>eval(int,'$UNKNOWN')
   ",
```

You can make the following change to your `firing_condition` attribute to filter the alert to ensure that the rule is only run on instances of the class `Cisco4000`.

```
entity_filter = "ClassName='Cisco4000'",
```

You can make the following change to your `firing_condition` attribute to run your current event rule after the event rule `PingFailToAlert`.

```
evaluate_after = "PingFailToAlert",
```

# execute_location

The `execute_location` attribute contains a set of sub-attributes which defines which events the rule takes action upon.

The sub-attributes are:

- *run_on_devices*
- *run_on_entities*
- *run_in_container*
- *entity_activity_state*

These sub-attributes are described in the following sections.

## run_on_devices

The `run_on_devices` attribute is a sub-attribute of `execute_location` and contains its own sub-attributes, as described in Table 80.

Table 80: run_on_devices Sub-Attribute Descriptions (1 of 2)

| Sub-Attribute | Description |
|---|---|
| `control` | Possible topologically-related values are:<br>• `0` - Instance - Selects only the object which generated the event that triggered the event rule.<br>• `1` - Isolated - Selects all objects which can only be reached by going though the instance.<br>• `2` - Connected - Selects all objects that are directly connected to the instance.<br>The scope specified by each of the above options is shown in Figure 27 on page 180. The event correlation rule is applied to all alerts from devices in scope, plus the original poll fail alert. In this example network, the device selection is also subject to the application of the containment model.<br>If `2` is selected then the following entities are used to run the rule:<br>• Any entity contained by the trigger entity<br>• Any entity directly connected to the trigger entity<br>• Any entity directly connected to an entity contained by the trigger entity<br>A further possible value for this attribute is `3` - Arbitrary Query. This value applies an OQL query to the `topoCache.entityByName` table. The query can make use of field values from the trigger event, if required. The `topoCache.entityByName` table, described in *topoCache.entityByName Entity Database Table* on page 170, contains all information relating to network entities, their containment and connections: |
| `isolated_from` | Defines the location of the polling agent. Used by AMOS to identify the devices that are isolated by the failure of the original device.<br>You can specify the location of the polling Agent using its entity name. This can be extracted from the rule triggering event using an eval statement similar to:<br>`isolated_from = "EntityName=eval(text,‘&AgentAddress’)",`<br>Note: This attribute is only required if `control` is set to `1`. |

Table 80: run_on_devices Sub-Attribute Descriptions (2 of 2)

| Sub-Attribute | Description |
|---|---|
| `entity_filter` | Gathers information about the device which triggered the event rule from AMOS entity database. Then applies an OQL statement in order to identify a related IP address. This filter is equivalent to the `where` clause of an OQL query that AMOS uses to select the required entities from the `topoCache.entityByName` table.<br><br>**Note:** This attribute is only required if `control` is set to `3`.<br><br>Here is an example:<br><br>`entity_filter = "Address(2) = eval(text,'&ExtraInfo->m_NbrIpAddress') OR eval(text,'&ExtraInfo->m_NbrIpAddress') IN (ExtraInfo->m_BgpPeers)"` |
| `include_trigger_entity` | Defines whether the trigger entity is included in the list of entities selected by the `execute_location` attribute. Possible values are:<br><br>• `true` - trigger entity is included<br>• `false` - trigger entity is not included<br><br>**Note:** This attribute is only required if `control` is set to `1`, `2`, or `3`. |

Figure 27 shows how the `control` attribute selects network devices.

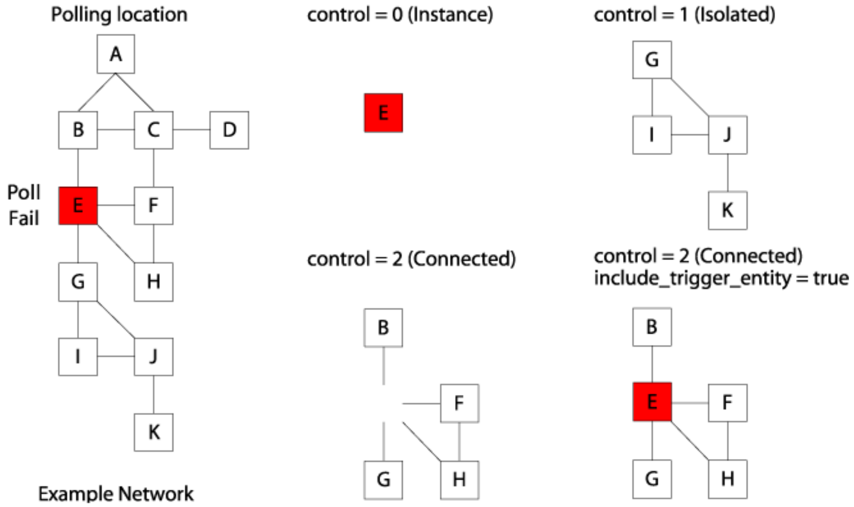


Figure 27: Application of the control Attribute

## run_on_entities

The `run_on_devices` attribute, described on page 179, returns all events on connected devices, using the object that generated the event as the reference. The `run_in_container` attribute, described on page 181, returns all events contained within the object that generated the event.

In contrast, the `run_on_entities` attribute provides a more generic way for you to search within the `mojo.events` database table for events that you are interested in. Formulate this search by defining a filter within the `run_on_entities` attribute. This filter is equivalent to a `where` statement within a `select` clause.

Formulate your event filter in *`event-filter`*. Use the IP address on which the event occurred or some other appropriate attribute of the event in order to formulate your event filter.

## run_in_container

The `run_in_container` attribute is a sub-attribute of `execute_location`. It defines the containment model the event correlation rule uses. For a full description of containment, see the *Netcool/Precision IP Discovery Configuration Guide*.

For example, a switch consists of a chassis that contains a series of cards, which may in turn contain a series of ports. Additionally, these ports may be associated with a series of VLANs. Using the `run_in_container` attribute, you could consider only those events coming from the switch, or only the cards or ports, or some combination of these.

The `run_in_container` attribute contains the sub-attributes `target` and `filter`. These are described in Table 81.

Table 81: run_in_container Sub-Attribute Descriptions

| Sub-Attribute | Description |
|---|---|
| `target` | Defines the target for the event correlation rule using an eval statement. It must be entered in the format:<br>`run_in_container = {`<br>`target = "eval(list type text,‘this(`*`traverse_up`*`,`<br>*`traverse_dwn`*`,`*`controlFlag`*`)->EntityName’)",`<br>The value of *`traverseUp`* is an integer which specifies the number of levels to traverse *up* through the containment model. For example, if the event which triggered the event rule came from a card on a switch, a value of 1 for *`traverseUp`* would bring events on the switch into scope. It can also take the predefined value `$physical`, which recurses up to the top level of containment.<br>The value of *`traverse_dwn`* is an integer which specifies the number of levels to recurse *down* through the containment model. For example, if the event which triggered the event rule came from a card on a switch, a value of 1 for *`traverse_dwn`* would bring the ports in the card into scope. It can also take the predefined value `$physical`, which recurses down to the bottom level of containment.<br>The possible values of *`controlFlag`* are:<br>• `$includerecursed` - includes events on all devices along the path followed by traversing up or down the containment model.<br>• `$excluderecursed` - includes only events on the end device. |
| `filter` | Used to exclude some of the events you have just included by using the `target` attribute. An event record is only classed as in scope, if this filter evaluates to `true`. |

**Note:** To disable recursing through the containment model the `run_in_container` attribute is left empty, that is, `run_in_container = {}`

## run_in_container Examples

Figure 28 shows an example container structure.

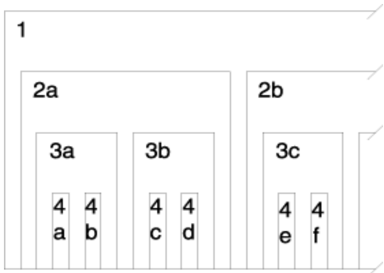


Figure 28: Example Device Showing Container Structure

Using the following `run_in_container` syntax, when an alert is received from a device in container 4a, the scope transverses up by 2 levels and includes the events from the devices in the containers at each level. The included devices are in containers 2a, 3a and 4a.

```
run_in_container = {
               target = "eval(list type
text,'this(2,0,$includerecursed)->EntityName')",
```

Using the following `run_in_container` syntax, when an alert is received from a device in container 2a, the scope transverses down by 2 levels and includes the events from the devices in the containers at each level. The included devices are in containers 2a, 3a, 3b, 4a, 4b, 4c and 4d.

```
run_in_container = {
               target = "eval(list type
text,'this(0,2,$includerecursed)->EntityName')",
```

Using the following `run_in_container` syntax, when an alert is received from a device in container 4a, the scope transverses up by 2 levels and includes only the events from the devices in the end container. The included devices are in container 2a.

```
run_in_container = {
               target = "eval(list type
text,'this(2,0,$excluderecursed)->EntityName')",
```

Using the following `run_in_container` syntax, when an alert is received from a device in container 2a, the scope transverses down by 2 levels and includes only the events from the devices in the end container. The included devices are in containers 4a, 4b, 4c and 4d.

```
run_in_container = {
               target = "eval(list type
text,'this(0,2,$excluderecursed)->EntityName')",
```

In the following example, the `run_in_container` section uses the `filter` attribute to exclude all event records not from interfaces.

```
run_in_container = {
               target = "eval(list type text,‘this(0,2,$excluderecursed)
                               ->EntityName’)",
               filter = "EntityType = 2 AND
                               ExtraInfo->m_ifIndex=eval(int,‘&ExtraInfo->ifIndex’)"
               }
```

### entity_activity_state

The `entity_activity_state` attribute is a sub-attribute of `execute_location`. The attribute has three possible values, as described in Table 82.

Table 82: entity_activity_state Attribute Description

| Value | Description |
|---|---|
| 0 | DEACTIVATE – Select this value when the event that triggered this rule indicates that connectivity has been lost to the device. For example, a LinkDown trap or a PingFail event. |
| 1 | ACTIVATE – Select this value when the event that triggered this rule indicates that connectivity has been restored to the device. For example, a LinkUp trap or a PingRestore event. |
| 2 | HOLDSTATE – Select this value when the event that triggered this rule does not necessarily indicate a restoration or a loss of connectivity to the device. For example, a vendor-specific trap indicating that the temperature of a router had exceeded a certain limit. |

## execute_rule

The `execute_rule` attribute contains a set of sub-attributes which manipulates the records in the AMOS Events database.

The sub-attributes are:

- *create_events*
- *change_events*
- *delete_events*
- *run_directives*
- *rule_control*

AMOS always runs the components of the event rule in this order. If you want to run the components in a different order, you must define two separate event correlation rules and use rule chaining.

**Note:** In the `execute_rule` section a double ampersand (&&) references the trigger record (the record that originally triggered this event rule). A single ampersand (&) references the current record (the record that the rule is presently considering).

## create_events

The `create_events` attribute is a sub-attribute of `execute_rule`. It allows you to create new events and send them to the AMOS events database.

The `create_events` attributes contains sub-attributes `use_location`, `action` and `actionfilter`. An instance of these sub-attributes in combination is referred to as an *action*. There can be more than one action in the `create_events` section. The following text shows a simplified example of the syntax used for multiple actions.

```
create_events = [

               {  // first action is everything within these curly braces

                    use_location = 1,
                    action = {
                         copy_record = 0,
                         actionvalues = [                         ],
                         no_run_when_exists= ""
                         },
                    actionfilter = ""

               },  // end of first action

               {  // second action is everything within these curly braces

                    use_location = 1,
                    action = {
                         copy_record = 0,
                         actionvalues = [                         ],
                         no_run_when_exists=""
                         },
                    actionfilter = ""

               }  // end of second action

],
```

The sub-attributes of `create_events` are described in Table 83.

Table 83: create_events Sub-Attribute Descriptions

| Sub-Attribute | Description |
|---|---|
| `use_location` | A boolean integer which defines whether the scope specified in `actionfilter` is used or not. Possible values are:<br>• `0` - false - the `actionfilter` is bypassed and any record specified by the `execute_location` section runs the `action`, subject to the `no_run_when_exists` attribute.<br>• `1` - true - the `actionfilter` functions as normal. |
| `action` | This attribute has three further sub-attributes:<br>• `copy_record`<br>• `actionvalues`<br>• `no_run_when_exists`<br>These are described in Table 84 on page 187. |
| `actionfilter` | Determines whether the action goes ahead. The filter must evaluate *true* for the action to be carried out.<br>The following example allows the `action` to be carried out only on the trigger event.<br>`actionfilter = "EventId=eval(long,'&&EventId')"    //&& is the trigger event`<br>`}`<br>`]` |

Table 84 describes the sub-attributes of the `action` attribute. The `action` attribute is itself a sub-attribute of `create_events`.

Table 84: action Sub-Attribute Descriptions

| Sub-Attribute | Description |
| --- | --- |
| `copy_record` | An integer value that specifies which record is to be used as the template for the record that is being created. The record being created uses the column names of the specified record, and keeps the values of that record unless they are overridden by the `actionvalues` attribute. Possible values are:<br>• 0 - Copies the trigger record.<br>• 1 - Copies the record currently being considered.<br>• 2 - Specifies that no record is to be copied. If this option is used, you must ensure that the `actionvalues` creates an entire, valid record. Specifically, all columns which are specified in the schema as being KEY or NOT NULL must be populated.<br>The following example copies the trigger record:<br>`create_events = [`<br>`{`<br>`action = {`<br>`copy_record = 0,` |
| `actionvalues` | Specifies column names of the record (defined in `copy_record`) to be overwritten, and the values to be entered into them. The following example shows the escalation of an event to an alert.<br>`actionvalues = [`<br>`"EventType = eval(int,‘$ALERT’)",`<br>`"CauseType = eval(int,‘$CAUSEUNKNOWN’)",`<br>`"Occurred = 1",`<br>`"Severity = eval(int,‘$MINOR’)"`<br>`],` |
| `no_run_when_exists` | Used to check whether the event or alert which has just been defined by the `copy_record` and `values` attributes already exists. It consists of a logical test on the column values of the event which has just been created. If the test evaluates *true*, this particular `action` (remember there can be more than one action in any one particular `create_events` section) is *not* run. An example of this filter is given below.<br>`no_run_when_exists = "`<br>`EntityName = eval(text,‘&&EntityName’) AND   // && is the trigger record`<br>`EventName = eval(text,‘&&EventName’) AND`<br>`RuleSet = eval(text,‘&&RuleSet’) AND`<br>`EventType = eval(int,‘$ALERT’)"`<br>`},` |

## change_events

The `change_events` attribute is a sub-attribute of `execute_rule`. It allows you to change AMOS events that match the `filter` attribute, when events in scope match the `actionfilter` attribute.

The `change_events` attributes contains sub-attributes `use_location`, `action` and `actionfilter`. An instance of these sub-attributes in combination is referred to as an action. There can be more than one action in the `change_events` section. The following text shows a simplified example of the syntax used for multiple actions.

```
change_events = [

               {  // first action is everything within these curly braces

                    use_location = 1,
                    action = {
                      no_run_when_exists= ""
                      },
                    actionfilter=""

               },  // end of first action

               {  // second action is everything within these curly braces

                    use_location = 1,
                    action = {},
                    actionfilter=""

               }  // end of second action

     ],
```

The sub-attributes of change_events are described in Table 85.

Table 85: change_events Sub-Attribute Descriptions

| Sub-Attribute | Description |
|---|---|
| use_location | A boolean integer which defines whether the scope specified in actionfilter is used or not. Possible values are:<br>• 0 - false - the actionfilter is bypassed and any record specified by the execute_location section runs the action, subject to the no_run_when_exists attribute.<br>• 1 - true - the actionfilter functions as normal. |
| action | This attribute has two further sub-attributes:<br>• actionvalues<br>• filter<br>• no_run_when_exists<br>These are described in Table 86. |
| actionfilter | This attribute ensures the following is true in order for the action to proceed:<br>• At least one record has been selected to be in scope by execute_location.<br>• At least one of the records in scope has been passed by actionfilter, unless actionfilter has been disabled by use_location.<br>• The record to be changed (not necessarily one of the records in scope) has been passed by filter.<br>There may be several actions in any one create_events section. The actionfilter of the last action marks the end of the create_events section. The next section, delete_events, is the simplest of the sections which modify the AMOS Events database.<br>The behavior of the filter and actionfilter is shown in Figure 29 on page 191 |

Table 86 describes the sub-attributes of the `action` attribute. The `action` attribute is itself a sub-attribute of `change_events`.

Table 86: action Sub-Attribute Descriptions

<table>
<tr><th>Sub-Attribute</th><th>Description</th></tr>
<tr><td><code>actionvalues</code></td><td>Allows you to change values of the records in scope. The following example increments the <code>Occurred</code> field of the records.
<pre>change_events = [
        {
              action = {
                    values = [
                          "Occurred = eval(int,‘(&Occurred) + 1’)"
                    ],</pre>
The <code>Occurred</code> field is displayed in the List Views of the Precision Desktop. It shows how many times the same event on the same device has occurred.</td></tr>
<tr><td><code>filter</code></td><td>The following must evaluate as true, for the action to proceed on the record specified in the <code>filter</code> (subject to its passing the <code>actionfilter</code>):
<ul><li>The records in scope have been chosen by previous sections and passed onto this section.</li>
<li>The records in scope must pass the <code>actionfilter</code> for this <code>action</code>. However, the filter is not limited to the event records in scope.</li></ul>
The relationship between the <code>filter</code> and <code>actionfilter</code> attributes is discussed in filter and actionfilter Attribute Behavior on page 191.<br><br>
Note: Each <code>filter</code> attribute only applies to the current action in a multiple action section. The <code>filter</code> also usually refers to the record currently in scope. This uses a single ampersand.<br><br>
The following example specifies that the <code>action</code> runs on the record currently in scope.
<pre>      filter = "EventId = eval(long,‘&EventId’)"
},</pre></td></tr>
<tr><td><code>no_run_when_exists</code></td><td>Used to check whether the event or alert which has just been defined by the <code>values</code> and <code>filter</code> attributes already exists. It consists of a logical test on the column values of the event which has just been changed. If the test evaluates true, this particular <code>action</code> (remember there can be more than one action in any one particular <code>change_events</code> section) is not run. An example of this filter is given below.
<pre>no_run_when_exists = "
      EntityName = eval(text,‘&&EntityName’) AND   // && is the
trigger record
      EventName = eval(text,‘&&EventName’) AND
      RuleSet = eval(text,‘&&RuleSet’) AND
      EventType = eval(int,‘$ALERT’)"
},</pre></td></tr>
</table>

### filter and actionfilter Attribute Behavior

Figure 29 shows the `filter` and `actionfilter` in use.

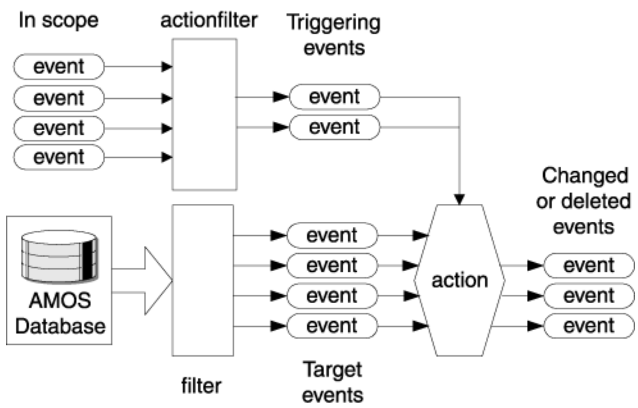


Figure 29: filter and actionfilter Attribute Operation

Only an event which has passed all the tests and filters up to this point can trigger the `action`, but the `action` can be run on any event. The `actionfilter` attribute specifies which events run the `action`, and the `filter` attribute specifies which events the `action` is run on.

## delete_events

The `delete_events` attribute is a sub-attribute of `execute_rule`. It allows you to delete AMOS events that match the `filter` attribute, when events in scope match the `actionfilter` attribute.

This `delete_events` attribute has a similar structure to the `change_events` attribute. As there is only one possible action, to delete the event, the sub-attribute `actionvalues` is not required. For descriptions of all other sub-attributes, see *change_events* on page 188.

Like the sections for `create_events` and `change_events`, the `delete_events` section can contain more than one action. The table below gives a simplified example of the syntax of multiple actions. The following text shows a simplified example of the syntax used for multiple actions.

```
delete_events = [

               {  // first action is everything within these curly braces

                    use_location=1
                    action={
                         filter=""
                         no_run_when_exists=""
                    },
                    actionfilter=""
```

```
        },  // end of first action

        {  // second action is everything within these curly braces

             use_location=1
             action={
                  filter=""
                  no_run_when_exists=""
             },
             actionfilter=""

        }  // end of second action

    ],
```

**Note:** Micromuse recommends that you delete events from the Netcool/OMNIbus ObjectServer.

## run_directives

The `run_directives` attribute is a sub-attribute of `execute_rule`. It allows you to run external actions by sending commands to Netcool/Precision IP Precision Server components.

**Note:** Micromuse recommends this command functionality is achieved using the automations available with the Netcool/OMNIbus ObjectServer.

The sub-attributes of run_directives are described in Table 87.

Table 87: run_directives Sub-Attribute Descriptions

| Sub-Attribute | Description |
|---|---|
| subject | Identifies the service or Netcool/Precision IP component to which you want to send a command. In the example below, a command is being sent to EXEC.<br>`run_directives = [`<br>`{`<br>`subject = "eval(text,'$Exec')",`<br>For a complete list of the service names of all Precision Server components, see the *Netcool/Precision IP Discovery Configuration Guide*. |
| copy_values | Specifies which record is to be used as the template for the record that is being created. The copy_values attribute is similar to the copy_record attribute used in change_events. Possible values are:<br>• true - the values of the trigger record overrides those in the database referenced in the send_string attribute.<br>• false - only the values specified in the send_string attribute are inserted into the target database. |
| send_string | Contains an OQL string that is sent to the service specified in subject. For descriptions of the OQL syntax and the eval statement, see the *Netcool/Precision IP Discovery Configuration Guide*.<br>For the columns and data types of the target database (which constrain the contents of the send_string attribute), refer to the schema of the relevant database, described in the *Netcool/Precision IP Discovery Configuration Guide*. |

There may be several directives within the square brackets. Directives should be enclosed by curly braces and separated by commas. If you wish to reference values from event records in send_string, you should note that, in contrast to the other sub-attributes of execute_rule, run_directives is only run once, and only against the trigger record. Therefore, a double ampersand reference (&&) has no meaning within run_directives, and a single ampersand, which usually references the record currently in scope, in this case always references the trigger record.

## run_directives Example

The following example of the run_directives attribute launches a pager script.

```
run_directives = [
            {
             subject = "eval(text,'$Exec')",  // service name of EXEC
             copy_values = False,
             send_string = "insert into actions.inTray (ActionId, Path, ArgList,
RunCount, ActionState) values
                   (42, '/export/paul/pager.sh', ['Come to the office a.s.a.p'], 1, 0);"
             }
]
```

## rule_control

The `rule_control` attribute is a sub-attribute of `execute_rule`. It allows you to set up virtual daemons on certain devices, to assist in downstream suppression.

**Note:** Suppression of downstream alerts begins in the `change_events` section of the event correlation rule. A correctly configured event correlation rule can suppress any alerts which already exist in the AMOS events database. However, you may want to continue suppressing downstream alerts until the original alert has been cleared. To do this you must set up a virtual daemon to watch each downstream device.

A virtual daemon, when run in a particular event rule, runs on the triggering device, and watches for events coming from any of the devices specified in the `execute_location` section of the rule that set up the daemon. When an event from one of these devices comes into the AMOS events database, the daemon runs the `execute_rule` section of the rule which set up the daemon.

Virtual daemons are usually used in downstream suppression, therefore, they are usually run from a rule whose `execute_rule` section suppresses downstream events, and whose `execute_location` section specifies devices downstream.

Virtual daemons run until they are cleared. They can be cleared by running a specific rule on the device on which the daemon is running.

The sub-attributes of `rule_control` are described in Table 88.

Table 88: rule_control Sub-Attribute Descriptions

<table>
<tr><th>Sub-Attribute</th><th>Description</th></tr>
<tr><td><code>daemon_running_order</code></td><td>Whenever an event from a downstream device comes in to the AMOS Events database, it has certain event rules run on it, depending on its properties. This is known as standard event processing. The incoming event may also be suppressed by a virtual daemon, by having the <code>execute_rule</code> section of the event rule that set up the daemon run on it. The <code>daemon_running_order</code> attribute specifies whether this suppression occurs before or after standard event processing. Possible values are:<br>• <code>0</code> - Runs the <code>execute_rule</code> section of the event rule before standard event processing.<br>• <code>1</code> - Runs the <code>execute_rule</code> section of the event rule after standard event processing.<br>• <code>2</code> - Never runs the <code>execute_rule</code> section of the event rule.<br>In the following example, the <code>execute_rule</code> section of the event rule runs after standard event processing. This is the usual setting.<br><pre>rule_control = {
     daemon_running_order = 1,</pre></td></tr>
<tr><td><code>set_daemon</code></td><td>Activates the daemon. Possible values are:<br>• <code>true</code> - The daemon is activated<br>• <code>false</code> - the daemon is prevented from running</td></tr>
<tr><td><code>clear_daemon</code></td><td>Specifies the rule or rules which, when run on the device the daemon is running on, clears the daemon.<br>The following example shows the <code>rule_control</code> section with the <code>clear_daemon</code> attribute specifying two rules (<code>pingFailClearEvent</code> and <code>linkDownRootCauseSuppressDownstream</code>).<br><pre>rule_control = {
               daemon_running_order = 1,
               clear_daemon = [ "pingFailClearEvent",
                    "linkDownRootCauseSuppressDownstream" ]
               }</pre></td></tr>
</table>

## 7.5 TopologicalAlertCorrelation Ruleset

The event correlation policy, or *ruleset* is a group of linked event correlation rules. The event correlation rules which constitute a ruleset can be grouped in the following ways:

- The rules in a ruleset can be chained together using the `evaluate_` attribute, as described in *firing_condition* on page 176.
- Rules can refer to the ruleset to which they belong in their `ruleset` attribute, as follows:

  ```
  ruleset = 'TopologicalAlertCorrelation',
  ```

  By setting this `ruleset` attribute in your rule, only events which, amongst other conditions, have been assigned to the ruleset `TopologicalAlertCorrelation` can run the rule.

An example use for a policy is to correlate ping fails. When a ping fail or restore event occurs you might want to:

- Escalate the event to root cause if the failure has occurred enough times.
- Suppress any events topologically downstream of the ping fail from the perspective of the polling station.
- Clear the corresponding ping fail event and reawaken any events that had previously been suppressed by it.

A single ruleset, the `TopologicalAlertCorrelation` ruleset, is supplied with Netcool/Precision IP. This ruleset links event correlation rules to perform the following actions:

- It suppresses alerts on downstream and connected devices based on events received from polling. The rules that make up the suppression part of the ruleset are described in *Suppression* on page 197.
- Once the root cause alert has been cleared, AMOS unsuppresses events which had been suppressed due to this root cause alert. This act of resetting is known as *wakeup*. The rules that make up the wakeup part of the ruleset are described in *Wakeup* on page 199.

---

**Note:** The `TopologicalAlertCorrelation` ruleset is designed to enable effective event correlation without configuration. It is also extendable should you wish to add your own rules to this ruleset. Every rule should belong to a ruleset. AMOS gives a warning if you create a rule that does not belong to a ruleset, although the rule still functions.

---

When AMOS escalates an alert to root cause, the alert appears in the Netcool/OMNIbus event list as having severity `Critical` (red). AMOS sets all the symptom alerts to severity `Unknown` (purple). Alerts which AMOS is unable to classify as either root cause or symptoms are set to severity `Warning` (blue).

When a `Clear` event is received for a root cause alert, AMOS sets this alert to severity `Clear` (green). It unsuppresses all the symptom alerts and sets them to severity `Warning` (blue). These events remain in the ObjectServer and depending on whether they are real problems they may be cleared at a later stage, or they may, in turn, become root causes of other events.

## Suppression

The `TopologicalAlertCorrelation` ruleset performs suppression by applying three different types of rule in sequence – head rules, transition rules and suppression rules:

- *Head Rules*: receive events from the Netcool/OMNIbus ObjectServer via the Event Gateway and determine the device or interface which generated the event.
- *Transition Rules*: determine whether or not it is necessary to perform suppression on this event.
- *Suppression Rules*: perform suppression of events on entities downstream and connected to the device on which the triggering event occurred.

The rules that are involved in suppression within the `TopologicalAlertCorrelation` ruleset are listed in Table 89.

Table 89: Rules Involved in Suppression Within the TopologicalAlertCorrelation Ruleset (1 of 3)

| Rule | Rule Type | Description |
|---|---|---|
| `EntityEventToAlert` | Head Rule | This rule is triggered by an incoming event, known as the *triggering* event. The rule determines whether this event already exists in the `mojo.events` table, and then takes one of the following actions:<br>• If the event does not exist, the rule creates a new event based on this event in the `mojo.events` table.<br>• If the event already exists, the rule updates the `mojo.events` table to increase the value of the `Occurred` field, which indicates how many times this event has occurred. |
| `InterfaceOrModule EventToAlert` | Head Rule | This rule carries out the same tasks as the `EntityEventToAlert` rule, with the following difference: in the `EntityEventToAlert` rule, the incoming event comes directly from the affected entity, while in this rule the incoming event comes from the main chassis device rather than directly from the affected entity. The `InterfaceOrModuleEventToAlert` rule therefore has to first determine which is the affected entity by performing a containment search using the `IfIndex` field or `IfDescr` field of the event as a key to search on. |

Table 89: Rules Involved in Suppression Within the TopologicalAlertCorrelation Ruleset (2 of 3)

| Rule | Rule Type | Description |
|---|---|---|
| NonTimedAlert Transition | Transition Rule | This rule is fired by one of the head rules, EntityEventToAlert or InterfaceOrModuleEventToAlert. The rule determines if the triggering event processed by the relevant head rule is the most important event on this interface. The most important event suppresses all other events on the interface.<br><br>The rule determines which is the most important event by inspecting the Precedence value within the config.precedence table for the triggering event and comparing this Precedence value with the Precedence values for all other alerts on this interface.<br><br>• If the triggering event is the only event on this interface, or if the Precedence value for the triggering event is higher than the Precedence value for all other events on this interface, then the triggering event becomes the root cause on this interface and suppresses any other events on this interface. The rule then also fires the suppression rules, SuppressConnectedAlert and SuppressDownstreamAlerts.<br>• If the triggering event has a lower Precedence value than one or more other events on this interface, then this rule terminates without firing any suppression rules. One of the other events on this interface will be discovered to be root cause and will initiate suppression.<br><br>For information on how to set the Precedence value within the config.precedence table, see *The precedence Table* on page 152. |
| TimedAlert Transition | Transition Rule | This rule carries out the same task as the NonTimedAlertTransition rule, with the following differences:<br><br>• The TimedAlertTransition rule is not fired by a head rule. Rather, it fires periodically every 30 seconds.<br>• It runs against all events in the mojo.events table. It will therefore process any new event which arrives from the Event Gateway. This is in contrast to the NonTimedAlertTransition rule which processes only the triggering event processed by the relevant head rule. |

Table 89: Rules Involved in Suppression Within the TopologicalAlertCorrelation Ruleset (3 of 3)

| Rule | Rule Type | Description |
|---|---|---|
| `Suppress ConnectedAlerts` | Suppression Rule | Suppresses all events which are on devices topologically one hop away from the root cause event identified by one of the transition rules. Examples of events which would be suppressed by this rule are shown in the following:<br>• Figure 19 *Chassis Failure Suppresses Failures On Connected Interfaces* on page 161<br>• Figure 20 *Chassis Failure Suppresses Failures Devices Connected to Contained Entities* on page 162<br>• Figure 22 *Interface Failure Suppresses More Recent Failure On Directly Connected Neighbor Interface* on page 164<br>This rule also sets up virtual daemons to sweep for further alerts on connected devices. These daemons are all cleared when a `Clear` event is received for the root cause event and subsequently one of the wakeup rules is run on the root cause event. |
| `Suppress DownstreamAlerts` | Suppression Rule | Suppresses all events which are on devices downstream of the root cause event identified by one of the transition rules. An example of events which would be suppressed by this rule are shown in Figure 21 *Chassis Failure Suppresses Failures On Downstream Entities* on page 163.<br>This rule also sets up virtual daemons to sweep for further alerts on downstream devices. These daemons are all cleared when a `Clear` event is received for the root cause event and subsequently one of the wakeup rules is run on the root cause event. |

# Wakeup

The `TopologicalAlertCorrelation` ruleset performs wakeup by applying two different types of rule in sequence – head rules and wakeup rules:

- *Head Rules*: filter out `Clear` events received from the Netcool/OMNIbus ObjectServer via the Event Gateway and determine the device or interface which generated the event.
- *Wakeup Rules*: unsuppress events on entities downstream and connected to the device on which the triggering event occurred.

The rules that are involved in wakeup within the `TopologicalAlertCorrelation` ruleset are listed in Table 90.

Table 90: Rules Involved in Wakeup Within the TopologicalAlertCorrelation Ruleset

| Rule | Rule Type | Description |
|---|---|---|
| `EntityClearEvent` | Head Rule | This rule corresponds to the `EntityEventToAlert` rule. That rule creates a new alert (or updates an existing alert) based on an event indicating a problem. This rule clears that alert (if it exists).<br><br>This rule is triggered by an incoming `Clear` event. This type of event indicates that a problem on an entity has cleared. The rule determines whether a corresponding problem-indicating alert exists for this entity in the `mojo.events` table and then takes one of the following actions:<br>• If there is no such problem-indicating alert in the `mojo.events` table, the rule creates a new `Clear` alert in the `mojo.events` table based on the newly arrived event.<br>• If a problem-indicating alert already exists, the rule updates the `mojo.events` table to change the severity of the event to `Clear`. This clears the root cause alert identified in one of the transition rules. |
| `InterfaceOrModule Clear` | Head Rule | This rule carries out the same tasks as the `EntityClearEvent` rule, with the following difference: in the `EntityClearEvent` rule, the incoming event comes directly from the affected entity, while in this rule the incoming event comes from the main chassis device rather than directly from the affected entity. The `InterfaceOrModule Clear` rule therefore has to first determine which is the affected interface by performing a containment search using the `IfIndex` field or `IfDescr` field of the event as a key to search on. |
| `ClearEventAwaken Connected` | Suppression Rule | This rule corresponds to the `SuppressConnectedAlerts` rule. That rule suppresses all events which are on devices topologically one hop away from the root cause event. This rule unsuppresses all these symptom events. |
| `ClearEventAwaken Downstream` | Suppression Rule | This rule corresponds to the `SuppressDownstreamAlerts` rule. That rule suppresses all events which are on devices downstream of the root cause event. This rule unsuppresses all these symptom events. |

# Index

## G



## I



## M



## N

# Contact Information

Corporate

| Region | Address | Telephone | Fax | World Wide Web |
|---|---|---|---|---|
| USA | Micromuse Inc. (HQ)<br>650 Townsend Street<br>San Francisco<br>CA 94103<br>USA | 1-800-Netcool (638 2665)<br>+1 415 568 9800 | +1 415 568 9801 | http://www.micromuse.com |
| Europe | Micromuse Ltd.<br>Disraeli House<br>90 Putney Bridge Road<br>London SW18 1DA<br>United Kingdom | +44 (0) 20 8875 9500 | +44 (0) 20 8875 9995 | http://www.micromuse.co.uk |
| Asia-Pacific | Micromuse Ltd.<br>Level 25<br>77 St Georges Terrace<br>Perth WA 6000<br>Australia | +61 (0) 8 9213 3400 | +61 (0) 8 9325 5030 | http://www.micromuse.com.au |

Technical Support

| Region | Telephone | Fax |
|---|---|---|
| USA | 1-800-Netcool (800 638 2665)<br>+1 415 568 9800 (San Francisco) | +1 415 568 9801 |
| Europe | +44 (0) 20 8877 0073 (London, UK) | +44 (0) 20 8875 0991 |
| Asia-Pacific | +61 (0) 8 9213 3470 (Perth, Australia)<br>+10 800 852 1012 (North China)<br>+10 800 152 1012 (South China) | +61 (0) 8 9486 1116 |

Online

| Team | E-Mail | World Wide Web |
|---|---|---|
| Licensing | Temporary Licenses: temp.licensing@micromuse.com<br>Permanent Licenses: perm.licensing@micromuse.com | support.micromuse.com/helpdesk/licenses |
| Support | support@micromuse.com | support.micromuse.com |