Oracle® Communications IP Service Activator

SDK Developer Overview Guide

Release 7.5
F59548-01
September 2022

Oracle Communications IP Service Activator SDK Developer Overview Guide, Release 7.5

F59548-01

Contents

Preface

Audience vii
Documentation Accessibility vii
Diversity and Inclusion vii

1 Overview of the Software Development Kit

About the SDK 1-1
What You Can Produce Using the SDK 1-1
- Additional SDK Terms 1-1
- SDK Structured Development Process 1-2
- SDK Samples 1-2
- Cartridges in Use 1-2
- Next Steps in Learning About the SDK 1-3
  - SDK Developer Guides 1-3
  - Other Documentation Sources 1-4

2 Overview of the Network Processor

Introduction 2-1
Network Processor Concepts 2-1
Network Processor and Cartridge Components 2-2
- Process 2-3
- Registering Cartridges with the Network Processor 2-4

Data Flow from a Configuration Policy through the Network Processor and Cartridge 2-4
Service Model 2-4
- Sample Service Model 2-5
- Definitions and Associations 2-11
- Sample Service Model Associations 2-11

Device Model 2-13
- Sample Device Model 2-13
- Device Model and the Service Model to Device Model Transform 2-14
- Sample Relating the Service Model to the Device Model 2-15

netFlow 2-15
staticRoute 2-15
Device Model Validation 2-16
Sample Annotated Device Model 2-16
Sample Relating the Device Model to the Annotated Device Model 2-17
Sample CLI Commands and the Device Model to CLI Transform 2-18
Sample CLI Document 2-18
CLI Elements 2-22
Sample Relating the Annotated Device Model to the CLI Document 2-22
CLI Merging 2-22
Merge Section Descriptions 2-22
Sample Relating the CLI Document to Configuration Commands 2-23
Command Executor 2-23
Network Processor End to End Flow-Through Illustration 2-24
Device Model Extension 2-24
Changeables and Identifiables 2-24

3 Cartridge Overview

Introduction to Cartridges 3-1
Base Cartridges 3-2
Service Cartridges 3-2
Configuration Policies 3-2
Cartridge Registration 3-2
Base Cartridge Registry.xml 3-3
Service Cartridge Extension.xml 3-5
About Subscriptions 3-6
Definition Type 3-6
Configuration Policy Identification 3-7
Configuration Policy ConfigPolicyRegistry.xml 3-7

4 Cartridge Operations

Audits 4-1
Audit Template 4-1
Audit Template Command Attributes 4-6
Audit Synonyms 4-10
Options 4-10
Capabilities 4-11
Message Definition 4-14
Overriding Message Definitions 4-15

Pre- and Post-Checks 4-16
Types of Pre- and Post-Checks 4-16
Type I Checks 4-16
Type II Checks 4-16
Cartridge Version 4-17
Device Model Upgrades 4-17
Network Processor NpUpgrade 4-18
Configuration Version 4-18
Configuration Management Support 4-18
For More Information on Cartridge Operations 4-18

5 Testing, Monitoring, and Error Handling

Test Environments 5-1
Unit Test 5-1
DM Validation 5-1
Logging 5-1
Audit Trail Logging 5-1
Handling Faults and Errors 5-2
Rollback 5-2
Device Model IDs 5-2
Rollback Failures 5-2
Quarantine 5-3

6 Best Practices

Choosing Whether to Extend a Cartridge or to Plan a New Cartridge 6-1
Choosing the Cartridge Implementation 6-1
Developer Knowledge 6-1
Transformation Complexity 6-1
Model Size 6-2
Time to Complete 6-2
XQuery Advantages and Disadvantages 6-2
Java Advantages and Disadvantages 6-2
XQuery Transform Best Practices 6-3
XQuery Performance Optimization 6-3
XQuery Searches 6-3
Best Practices for Coding XQuery 6-4
Syntax For Entering Control Characters in XQuery 6-4
Java Transform Best Practices 6-4
Java Searches 6-5

Service Model to Device Model Java Transform 6-5
Annotated Device Model to CLI Document Java Transform 6-6
Best Practices for Extending the Device Model 6-6

Preface

This guide gives an overview of the tools provided by the Oracle Communications IP Service Activator Software Development Kit (SDK), and the underlying SDK, Network Processor, cartridge, and configuration policy concepts.

Audience

This guide is intended for system developers using the SDK toolset to develop service cartridges, base cartridges, and configuration policies.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Diversity and Inclusion

Oracle is fully committed to diversity and inclusion. Oracle respects and values having a diverse workforce that increases thought leadership and innovation. As part of our initiative to build a more inclusive culture that positively impacts our employees, customers, and partners, we are working to remove insensitive terms from our products and documentation. We are also mindful of the necessity to maintain compatibility with our customers' existing technologies and the need to ensure continuity of service as Oracle's offerings and industry standards evolve. Because of these technical constraints, our effort to remove insensitive terms is ongoing and will take time and external cooperation.

1

Overview of the Software Development Kit

This chapter provides an overview of the Oracle Communications IP Service Activator Software Development Kit (SDK) and an introduction to its tools and techniques.

About the SDK

The SDK is a toolset for creating cartridges and configuration policies to be used by the Network Processor within a IP Service Activator installation to provide a flexible and scalable device driver solution.

The structured development process inherent to the SDK combined with its robust suite of tools will enable you to achieve accurate and consistent results.

What You Can Produce Using the SDK

The SDK is used to create base cartridges, service cartridges, and configuration policies for deployment into an IP Service Activator installation.

- Base cartridges: provide basic communications between IP Service Activator's Network Processor and its devices. A base cartridge provides the platform upon which to build service cartridges for a family of devices.
- Service cartridges: administer specific services on a specific family of devices, the basic functionality of which is provided by a base or core cartridge. A service cartridge must be deployed as an extension to a base cartridge or a core cartridge in an IP Service Activator Network Processor installation.
- Configuration policies: provide a GUI form and schema to collect data for services from IP Service Activator client users. In an IP Service Activator installation, configuration policies are implemented through service cartridges.
- Configuration Management support: using the SDK, you can create and configure cartridges that support the use of services from the Configuration Management product. For details on the Configuration Management product, refer to *Configuration Management Planning Guide*.

Additional SDK Terms

Core cartridges are existing IP Service Activator cartridges integrated within IP Service Activator. A core cartridge combines the basic communication functionality of a base cartridge with service administration all within one package.

Vendor cartridge is a conceptual term. A vendor cartridge consists of the union of a base or core cartridge with a number of service cartridges. Thus, a vendor cartridge has the functionality to connect to a specific device type, and administer the services handled by its service cartridges.

SDK Structured Development Process

The basic process for developing cartridges and configuration policies with the SDK is as follows:

1. Create and edit a properties file.
2. Generate the cartridge source files using the provided generator tool. This generator tool is an Ant script that uses the properties in the properties file to configure a set of skeleton source files for the cartridge. This script and others are run using a command line interface.
3. Edit the generated source files to add the specific functionality for your cartridge. This is where most of your developments efforts will be spent.
4. Compile and package the cartridge using the provided Ant script.
5. Perform standalone testing using the provided scripts.
6. Deploy the cartridge into a test IP Service Activator installation and perform end-to-end testing.

Specific details to perform these steps are described in the SDK developer guides.

SDK Samples

With the SDK, you receive a number of simple, working samples. The samples are valuable to serve as examples to learn from, and to use as starting points to develop your own cartridges and configuration policies.

SDK samples include:

- Base cartridge:
  - cisco: simple working base cartridge for Cisco IOS devices
- Service cartridges:
  - ciscoBanner: implements a banner on Cisco IOS devices using XQuery transforms
  - ciscoBannerJava: implements a banner on Cisco IOS devices using Java transforms
  - ciscoStaticRoute: implements a static route on Cisco IOS devices
  - ciscoMartini: implements a Martini Layer 2 VPN site on Cisco IOS devices
- Configuration policies:
  - bannerSample: implements a banner on Cisco IOS devices
  - staticrouteSample: implements a static route on Cisco IOS devices

Creating, deploying and testing the samples are described in the SDK developer guides.

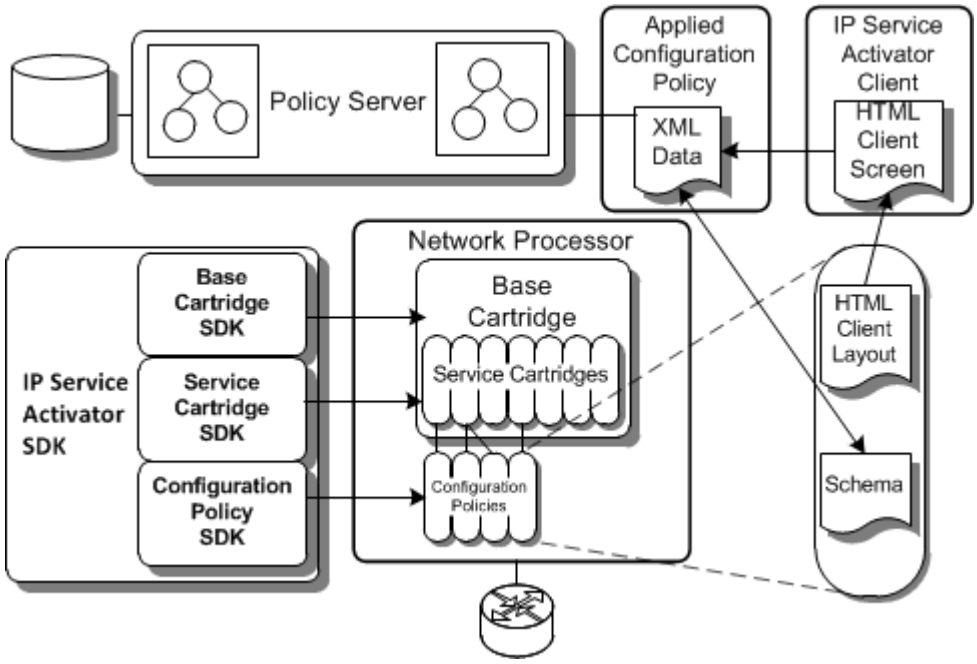
Cartridges in Use

Figure 1-1 shows cartridges and configuration policies developed using the SDK deployed in an IP Service Activator installation. The policy server (at the top center of the diagram) is the central component of IP Service Activator. It manages the flow of

data to and from its database (top left), provides GUIs for users to monitor and administer the network (top right) and through its Network Processors (lower center), it performs device discovery, runs audits, and sends configuration to devices.

As you can see, configuration policies (which provide a GUI form and schema to GUI users to collect and validate XML data for services) are implemented through service cartridges. A service cartridge is deployed as an extension to a base or core cartridge.

Figure 1-1 SDK Cartridges Deployed in an IP Service Activator Installation



Next Steps in Learning About the SDK

Continue reading this guide. It gives an effective description of the Network Processor, cartridges, and cartridge operation support components.

SDK Developer Guides

The developer guides in the IP Service Activator SDK documentation suite provide detailed technical explanations of how to create and configure cartridges and configuration policies. Procedures to build, package and test them are also included. Each developer guide contains detailed reference material explaining parameters, options, and the function of the various source files. Reading the following guides is recommended:

- *IP Service Activator SDK Base Cartridge Developer Guide*: learn how to create base cartridges with the SDK.
- *IP Service Activator SDK Service Cartridge Developer Guide*: many of the concepts learned in creating base cartridges are applicable to creating service cartridges. It is recommended you read IP Service Activator Base Cartridge Developer Guide first.
- *IP Service Activator SDK Configuration Policy Extension Developer Guide*: continues to build your knowledge about the SDK by detailing how to create configuration policies, which work in conjunction with service cartridges.

- *IP Service Activator SDK Configuration Management Developer Guide*: learn how to use cartridges to administer Configuration Management module services.

Other Documentation Sources

For a more technical understanding of the Network Processor, and how to configure it, refer to the discussion of Network Processor administration and maintenance in *IP Service Activator System Administrator's Guide*.

2

Overview of the Network Processor

This chapter provides an overview of the Oracle Communications IP Service Activator Network Processor, and follows a detailed configuration scenario from start to finish using sample service and device models, command-line interface (CLI) document, and final configuration commands.

In order to understand how to use the SDK to support new devices and services in IP Service Activator, you will need an understanding of the Network Processor framework which provides the link between the cartridges and configuration policies, IP Service Activator, and the devices themselves.

Detailed information on configuring and administering the operation of the Network Processor is given in *IP Service Activator System Administrator's Guide*. Additional information on related topics can be found in the IP Service Activator product documentation or online help.

Introduction

The Network Processor manages communication to and from devices through cartridges. Within the context of IP Service Activator, it performs the conversion of user changes to the configuration of IP Service Activator objects into a set of CLI commands for delivery to devices.

The Network Processor/cartridge architecture is extensible and scalable. Support for new services and devices can be added by creating and deploying new cartridges and cartridge components.

Within the context of Configuration Management, the Network Processor manages the flow of information from a configuration policy (or configlet) to a device, again by performing the conversion of configuration commands into a set of CLI commands for delivery to the device.

Network Processor Concepts

As mentioned above, the Network Processor converts IP Service Activator user changes into CLI commands that can be pushed to the devices. This task is accomplished in a series of steps that make use of key internal structures. These structures include:

- **Service model**: an XML document containing a representation of a device's topology and its service objects and their relationships. Even though there is one Service Model for each device in an IP Service Activator installation which is managed by a Network Processor, the representation of the service objects is device agnostic. The service model is device independent in the sense that the XML schema allows for the description of services in a device-agnostic manner.
- **Device model**: device-specific cartridge XML document derived from the service model. It contains a set of data elements that defines the complete device state. It extends the Network Processor's device model by adding validation rules so that the new service(s) which the cartridge enables can be fully described.
- **Annotated device model**: an XML document that contains the device model, with annotations to indicate the changes in device configuration between the device's state at

the last successful configuration push (as persisted in the database), and its expected state (target) after the current configuration is pushed.

> **Note:**
>
> At the time the Network Processor merges the base command document with the service cartridge command documents, the assocIdArray of each command and each nested command in the following sections of the resulting document will be populated with all non-zero assocIds (collected from all the documents to be merged): initialization, preconfig, postconfig, and finalize.

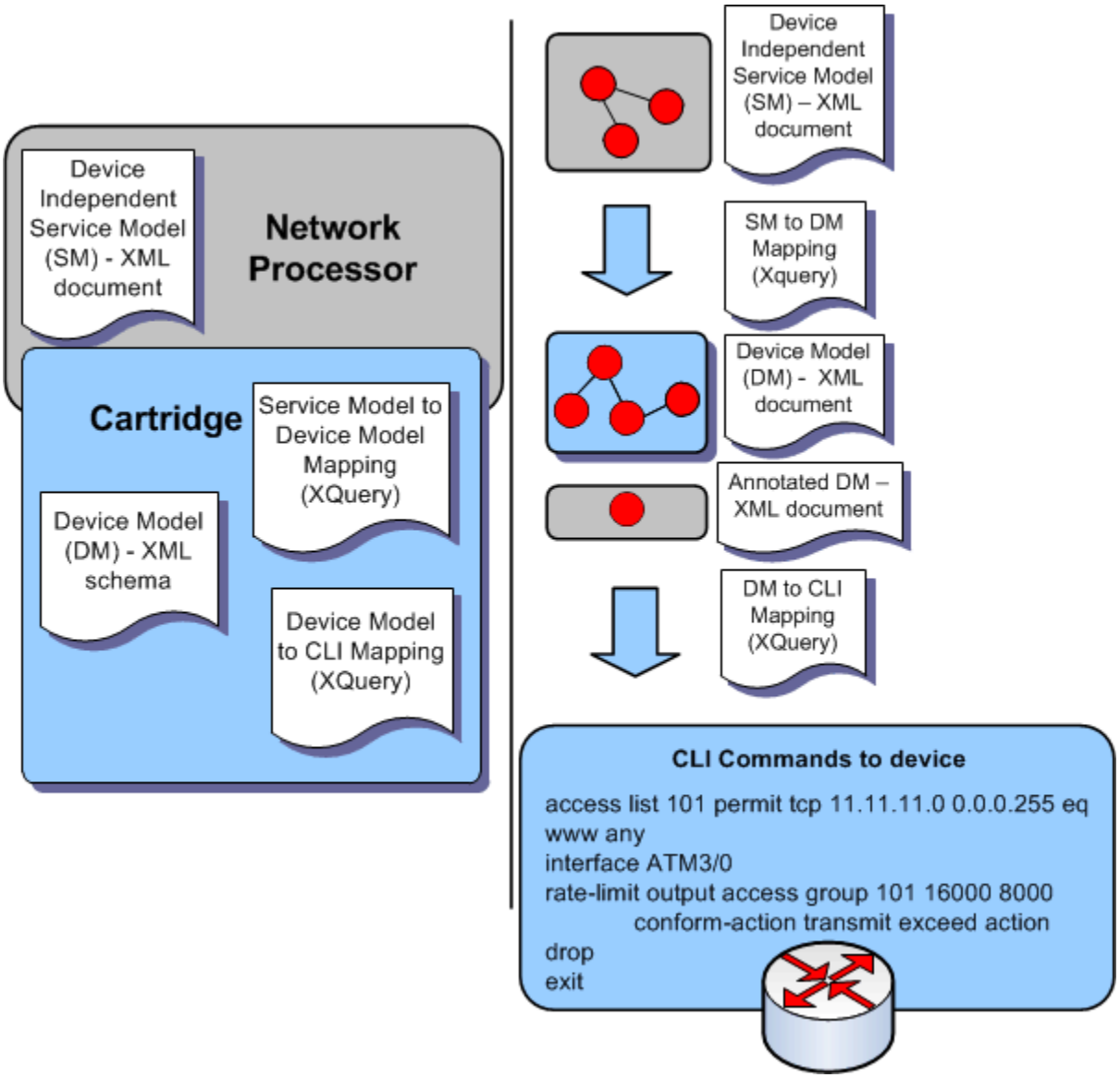
The task of converting user changes to the IP Service Activator object model into CLI commands uses key resources in the Network Processor framework and in the cartridge. These include:

- **Service model to device model transform**: this Java or XQuery transform takes a device-independent XML based service model and from it creates a device-specific XML based device model.
- **Annotated device model to CLI transform**: this Java or XQuery transform takes a device-specific XML based annotated device model and converts it to a CLI document.
- **DM validation**: this script is optional. If present in the cartridge, it is used by the Network Processor to validate the data in the generated device model.

Network Processor and Cartridge Components

Figure 2-1 shows an overview of the Network Processor tasks.

Figure 2-1 Network Processor Task Overview



Process

When a user makes changes to logical policies:

1. The policy server forwards these changes to the Network Processor that manages the affected device(s).
2. The Network Processor creates a target service model for each affected device.
3. The Network Processor determines which vendor cartridge to use for each affected device. It applies the service model to device model transform using the vendor cartridge to create a device model for each affected device.
4. The Network Processor compares the newly created target device model to the device model that was persisted to its database after the last successful push to that device. An

annotated device model contains annotations that indicate whether configuration is being added, deleted, or modified.

5. The Network Processor runs the annotated device model to CLI transform in the cartridge to create a CLI document. This CLI document contains data elements that represent the commands needed to change the device configuration from its present state to the desired state. That is, the document reflects the delta between the current and desired state.
6. The Network Processor executes the CLI document and sends the commands to the device via telnet or SSH.
7. Device responses are interpreted, and feedback is provided on the success or failure of changing the configuration to the desired state.

Once the commands have been successfully sent to the device, the service model and device model are persisted to the database so they can be referred to as the last service and device model in the next transaction committed on the device.

In a scenario in which multiple independent logical policy changes are made affecting the same device, it is possible for some to succeed and others to fail. In such a scenario, the persisted service model and device model will represent the successfully applied configuration changes.

Registering Cartridges with the Network Processor

When a cartridge is installed into a IP Service Activator system, the Network Processor creates a mapping from the information in the XML registry file supplied with the cartridge. When a service is applied to a device, the Network Processor can then determine which cartridge contains the necessary additional components which configure that service. Cartridge registration will be described in detail later on.

Data Flow from a Configuration Policy through the Network Processor and Cartridge

As shown in Figure 1-1, a configuration policy includes two key components:

- **HTML GUI screen**: this is an HTML document defining a data input form for the configuration policy
- **Schema**: this schema defines the data for the particular service that the configuration policy is used to create

When a configuration policy is applied to a device, the IP Service Activator user enters appropriate data for the service into the GUI through the HTML GUI form.

This data is stored in an XML file and is validated by the configuration policy's schema. A valid XML data file is then passed to the policy server which determines which Network Processor manages the targeted device.

The XML data is then managed by the Network Processor in the same manner as described above.

Service Model

This section discusses the details of a service model.

When the Network Processor receives a transaction commit for a device from the IP Service Activator policy server, it constructs a service model. A service model is a normalized representation of the device topology and the logical policies that are applied to it. The service model document contains two major sections: topology information, and definitions of logical policies that are applied to the device.

The sample service model has bold sections. These sections contain the data for the sample configuration that is being pushed. Other bold sections help visually define the structure of the document and emphasize certain data.

Sample Service Model

```
<ser:device xmlns:ser="http://www.metasolv.com/serviceactivator/servicemodel"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <ser:component_id>7020</ser:component_id>
  <ser:component_name>rotgsr-1.kanata.ca.oracle.com</ser:component_name>
  <ser:driver_type>cisco</ser:driver_type>
  <ser:ip_address>178013278</ser:ip_address>
  <ser:authentication_information>

    <ser:item>
      <ser:name>ORCHESTREAM.GENERIC.AUTHENTICATION.SNMP_READ_COMMUNITY</ser:name>
      <ser:value xsi:type="xs:string" xmlns:xs="http://www.w3.org/2001/
XMLSchema">U2w6VwySz9Q=</ser:value>
    </ser:item>
    <ser:item>
      <ser:name>ORCHESTREAM.CISCO.AUTHENTICATION.TACACS</ser:name>
      <ser:value xsi:type="xs:boolean" xmlns:xs="http://www.w3.org/2001/
XMLSchema">true</ser:value>
    </ser:item>
    <ser:item>
      <ser:name>ORCHESTREAM.CISCO.AUTHENTICATION.TACACS.USER_NAME</ser:name>
      <ser:value xsi:type="xs:string" xmlns:xs="http://www.w3.org/2001/XMLSchema"/>
    </ser:item>
    <ser:item>
      <ser:name>ORCHESTREAM.CISCO.AUTHENTICATION.TACACS.USER_RESPONSES</ser:name>
      <ser:value xsi:type="xs:string" xmlns:xs="http://www.w3.org/2001/XMLSchema"/>
    </ser:item>
    <ser:item>
      <ser:name>ORCHESTREAM.CISCO.AUTHENTICATION.ALL.ENABLE_PASSWORD</ser:name>
      <ser:value xsi:type="xs:string" xmlns:xs="http://www.w3.org/2001/XMLSchema"/>
    </ser:item>
  </
  <ser:configuration_information>
    <ser:item>
      <ser:name>ORCHESTREAM.GENERIC.CONFIGURATION.SNMP.RETRIES</ser:name>
      <ser:value xsi:type="xs:long" xmlns:xs="http://www.w3.org/2001/XMLSchema">2</
ser:value>
    </ser:item>
    <ser:item>
      <ser:name>ORCHESTREAM.GENERIC.CONFIGURATION.SNMP.TIMEOUT</ser:name>
      <ser:value xsi:type="xs:long" xmlns:xs="http://www.w3.org/2001/XMLSchema">3</
ser:value>
    </ser:item>
    <ser:item>
      <ser:name>ORCHESTREAM.GENERIC.CONFIGURATION.BGPLOCALAS</ser:name>
      <ser:value xsi:type="xs:long" xmlns:xs="http://www.w3.org/2001/XMLSchema">1</
ser:value>
    </ser:item>
    <ser:item>
```

```
      <ser:name>ORCHESTREAM.GENERIC.CONFIGURATION.MANUAL_CONFIG_MODE</ser:name>
      <ser:value xsi:type="xs:long" xmlns:xs="http://www.w3.org/2001/
XMLSchema">0</ser:value>
    </ser:item>
    <ser:item>
      <ser:name>ORCHESTREAM.GENERIC.CONFIGURATION.MAX_TRANSACTION_SIZE</ser:name>
      <ser:value xsi:type="xs:long" xmlns:xs="http://www.w3.org/2001/
XMLSchema">-1</ser:value>
    </ser:item>
    <ser:item>
      <ser:name>ORCHESTREAM.GENERIC.CONFIGURATION.PATTERN_TRANSACTION_SIZE</
ser:name>
      <ser:value xsi:type="xs:string" xmlns:xs="http://www.w3.org/2001/
XMLSchema">^no\s(?!alias)</ser:value>
    </ser:item>
    <ser:item>
      <ser:name>ORCHESTREAM.GENERIC.CONFIGURATION.DEVICE_TYPE</ser:name>
      <ser:value xsi:type="xs:string" xmlns:xs="http://www.w3.org/2001/
XMLSchema">Cisco 12008</ser:value>
    </ser:item>
    <ser:item>
      <ser:name>ORCHESTREAM.GENERIC.CONFIGURATION.DEVICE_DESCRIPTION</ser:name>
      <ser:value xsi:type="xs:string" xmlns:xs="http://www.w3.org/2001/
XMLSchema">Cisco Internetwork Operating System Software
IOS (tm) GS Software (GSR-P-M), Version 12.0(32)S5, RELEASE SOFTWARE (fc2)
Tech</ser:value>
    </ser:item>
    <ser:item>
      <ser:name>ORCHESTREAM.GENERIC.CONFIGURATION.COMMAND_DELIVERY_MODE</
ser:name>
      <ser:value xsi:type="xs:long" xmlns:xs="http://www.w3.org/2001/
XMLSchema">1</ser:value>
    </ser:item>
    <ser:item>
      <ser:name>ORCHESTREAM.GENERIC.CONFIGURATION.DEVICE_STATE</ser:name>
      <ser:value xsi:type="xs:long" xmlns:xs="http://www.w3.org/2001/
XMLSchema">5</ser:value>
    </ser:item>
    </ser:configuration_information>
  <ser:interface>
    <ser:component_id>7252</ser:component_id>
    <ser:component_name/>
    <ser:interface_definition>
      <ser:discriminator
xsi:type="ser:DeviceDriver.InterfaceQuery.InterfaceType">Type_INTERFACE</
ser:discriminator>
      <ser:interfaceParameters>
        <ser:interfaceID>7252</ser:interfaceID>
        <ser:if_direction>Dir_OUT</ser:if_direction>
        <ser:if_bandwidth>44210</ser:if_bandwidth>
        <ser:ip_address>0</ser:ip_address>
        <ser:snmp_if_index>3626</ser:snmp_if_index>
        <ser:name>Serial3/0</ser:name>
        <ser:internal_name/>
        <ser:valid_fields>6</ser:valid_fields>
        <ser:snmp_if_type>22</ser:snmp_if_type>
        <ser:if_level>0</ser:if_level>
        <ser:parent_component_id>0</ser:parent_component_id>
        <ser:parent_name/>
        <ser:owned>false</ser:owned>
      </ser:interfaceParameters>
```

```
    </ser:interface_definition>
  <ser:interface>
    <ser:component_id>7252</ser:component_id>
    <ser:component_name/>
    <ser:interface_definition>
      <ser:discriminator
xsi:type="ser:DeviceDriver.InterfaceQuery.InterfaceType">Type_INTERFACE</
ser:discriminator>
      <ser:interfaceParameters>
        <ser:interfaceID>7252</ser:interfaceID>
        <ser:if_direction>Dir_OUT</ser:if_direction>
        <ser:if_bandwidth>44210</ser:if_bandwidth>
        <ser:ip_address>0</ser:ip_address>
        <ser:snmp_if_index>3626</ser:snmp_if_index>
        <ser:name>Serial3/0</ser:name>
        <ser:internal_name/>
        <ser:valid_fields>6</ser:valid_fields>
        <ser:snmp_if_type>22</ser:snmp_if_type>
        <ser:if_level>0</ser:if_level>
        <ser:parent_component_id>0</ser:parent_component_id>
        <ser:parent_name/>
        <ser:owned>false</ser:owned>
      </ser:interfaceParameters>
    </ser:interface_definition>
    <ser:associations>
      <ser:item xsi:type="ser:NetworkProcessor.Association">
        <ser:id>
          <ser:id>1578</ser:id>
          <ser:ver>0</ser:ver>
        </ser:id>
        <ser:assoc_id>7432</ser:assoc_id>
        <ser:outbound>true</ser:outbound>
      </ser:item>
      <ser:item xsi:type="ser:NetworkProcessor.Association">
        <ser:id>
          <ser:id>7460</ser:id>
          <ser:ver>0</ser:ver>
        </ser:id>
        <ser:assoc_id>7462</ser:assoc_id>
        <ser:outbound>true</ser:outbound>
      </ser:item>
    </ser:associations>
  </ser:interface>
<ser:associations>
    <ser:item>
      <ser:id>
        <ser:id>7020</ser:id>
        <ser:ver>1</ser:ver>
      </ser:id>
      <ser:assoc_id>0</ser:assoc_id>
    </ser:item>
    <ser:item>
      <ser:id>
        <ser:id>1082</ser:id>
        <ser:ver>1</ser:ver>
      </ser:id>
      <ser:assoc_id>0</ser:assoc_id>
    </ser:item>
  </ser:associations>
<ser:definitions>
    <ser:item>
```

```
      <ser:id>
        <ser:id>7020</ser:id>
        <ser:ver>1</ser:ver>
      </ser:id>
      <ser:name>OCH_DeviceParameters</ser:name>
      <ser:value>
        <ser:discriminator
xsi:type="ser:DefinitionSite.DefinitionType">ParameterSetDefinitionType</
ser:discriminator>
        <ser:parameter_set>
          <ser:item>
            <ser:name>OCH_AuthSnmpReadComm</ser:name>
            <ser:value xsi:type="xs:string" xmlns:xs="http://www.w3.org/2001/
XMLSchema">U2w6VwySz9Q=</ser:value>
          </ser:item>
          <ser:item>
            <ser:name>OCH_AuthTacacsEnabled</ser:name>
            <ser:value xsi:type="xs:boolean" xmlns:xs="http://www.w3.org/2001/
XMLSchema">true</ser:value>
          </ser:item>
          <ser:item>
            <ser:name>OCH_AuthTacacsUsername</ser:name>
            <ser:value xsi:type="xs:string" xmlns:xs="http://www.w3.org/2001/
XMLSchema"/>
          </ser:item>
          <ser:item>
            <ser:name>OCH_AuthTacacsResponses</ser:name>
            <ser:value xsi:type="xs:string" xmlns:xs="http://www.w3.org/2001/
XMLSchema"/>
          </ser:item>
          <ser:item>
            <ser:name>OCH_AuthEnablePw</ser:name>
            <ser:value xsi:type="xs:string" xmlns:xs="http://www.w3.org/2001/
XMLSchema"/>
          </ser:item>
          <ser:item>
            <ser:name>OCH_BgpLocalAs</ser:name>
            <ser:value xsi:type="xs:long" xmlns:xs="http://www.w3.org/2001/
XMLSchema">1</ser:value>
          </ser:item>
          <ser:item>
            <ser:name>OCH_ManualConfigMode</ser:name>
            <ser:value xsi:type="xs:long" xmlns:xs="http://www.w3.org/2001/
XMLSchema">0</ser:value>
          </ser:item>
          <ser:item>
            <ser:name>OCH_MaxTransactionSize</ser:name>
            <ser:value xsi:type="xs:long" xmlns:xs="http://www.w3.org/2001/
XMLSchema">-1</ser:value>
          </ser:item>
          <ser:item>
            <ser:name>OCH_PatternTransactionSize</ser:name>
            <ser:value xsi:type="xs:string" xmlns:xs="http://www.w3.org/2001/
XMLSchema">^no\s(?!alias)</ser:value>
          </ser:item>
          <ser:item>
            <ser:name>OCH_CommandDeliveryMode</ser:name>
            <ser:value xsi:type="xs:long" xmlns:xs="http://www.w3.org/2001/
XMLSchema">1</ser:value>
          </ser:item>
          <ser:item>
```

```
            <ser:name>OCH_DeviceState</ser:name>
            <ser:value xsi:type="xs:long" xmlns:xs="http://www.w3.org/2001/
XMLSchema">5</ser:value>
          </ser:item>
          <ser:item>
            <ser:name>OCH_NetflowSourceInterface</ser:name>
            <ser:value xsi:type="xs:string" xmlns:xs="http://www.w3.org/2001/
XMLSchema">loopback0</ser:value>
          </ser:item>
          <ser:item>
            <ser:name>OCH_SAARtrResponder</ser:name>
            <ser:value xsi:type="xs:boolean" xmlns:xs="http://www.w3.org/2001/
XMLSchema">false</ser:value>
          </ser:item>
        </ser:parameter_set>
      </ser:value>
      <ser:options/>
    </ser:item>
    <ser:item>
      <ser:id>
        <ser:id>1578</ser:id>
        <ser:ver>1</ser:ver>
      </ser:id>
      <ser:name>netflowCollector</ser:name>
      <ser:value>
        <ser:discriminator
xsi:type="ser:DefinitionSite.DefinitionType">GenericRuleDefinitionType</
ser:discriminator>
        <ser:generic_rule_def>
          <ser:contentType>collectorParameters</ser:contentType>
          <ser:contentValue>
            <collectorParameters xmlns="http://www.metasolv.com/serviceactivator/
collectorParameters">
              <type>Cisco Netflow FlowCollector</type>
              <externalSystem>
                <primaryIP>
                  <ipAddress>3.3.3.3</ipAddress>
                  <port>32</port>
                </primaryIP>
              </externalSystem>
            </collectorParameters>
          </ser:contentValue>
          <ser:rule_directives/>
          <ser:order>4026531840</ser:order>
        </ser:generic_rule_def>
      </ser:value>
    <ser:options/>
    </ser:item>

    <ser:item>
      <ser:id>
        <ser:id>1082</ser:id>
        <ser:ver>1</ser:ver>
      </ser:id>
      <ser:name>myDomain</ser:name>
      <ser:value>
        <ser:discriminator
xsi:type="ser:DefinitionSite.DefinitionType">IBgpBaseDefinitionType</ser:discriminator>
        <ser:ibgp_base_definition>
          <ser:configure>false</ser:configure>
          <ser:internal_asn>1</ser:internal_asn>
```

```
          <ser:max_paths>1</ser:max_paths>
          <ser:update_source>0</ser:update_source>
          <ser:std_communities>true</ser:std_communities>
          <ser:md5_authentication/>
        </ser:ibgp_base_definition>
      </ser:value>
      <ser:options/>
    </ser:item>
    <ser:item>
      <ser:id>
        <ser:id>7460</ser:id>
        <ser:ver>1</ser:ver>
      </ser:id>
      <ser:name>static route</ser:name>
      <ser:value>
        <ser:discriminator
xsi:type="ser:DefinitionSite.DefinitionType">GenericRuleDefinitionType</
ser:discriminator>
        <ser:generic_rule_def>
          <ser:contentType>staticRoutes</ser:contentType>
          <ser:contentValue>
            <staticRoutes xmlns="http://www.metasolv.com/serviceactivator/
staticroute">
              <staticRoute>
                <ip>20.20.20.20</ip>
                <mask>32</mask>
                <next_hop_ip>21.21.21.21</next_hop_ip>
                <distance_metric>3</distance_metric>
              </staticRoute>
            </staticRoutes>
          </ser:contentValue>
          <ser:rule_directives/>
          <ser:order>805306368</ser:order>
        </ser:generic_rule_def>
      </ser:value>
      <ser:options/>
    </ser:item>
  </ser:definitions>
</ser:device>
```

Key sections in the sample service model are:

- **component_id:** unique ID of the device
- **component_name:** name of device
- **driver_type**: family of devices that this device belongs to.

> **Note:**
>
> In order for the Network Processor to determine which cartridge instance to invoke to process the service model, it compares the driver type against the registry of cartridges, searching for a cartridge instance that administers this driver type. When found it will invoke the entries indicated by the cartridge instance.

- **ip_address**: IP address of the device
- **authentication_information**: information about login credentials of the device

- **configuration_information**: information on network processor behavior to implement for this device
- **device topology**: this is the target of a definition
  - **device (implied)**: physical hardware representation of the device
  - **device associations**: links to the logical policies that are applied to this device. The link is made using the logical policy's definition ID.
- **interface topology**: physical hardware or software representation of an interface
  - **interface associations**: links to logical policies that are applied to this interface. The link is made using the logical policy's definition ID.
- **definitions**: listing of logical policies that are applied to the device and/or interfaces
  - **id**: value which uniquely identifies the logical policy in the service model.
  - **name**: name of logical policy
  - **value**: details of the logical policy

Definitions and Associations

As mentioned above, the service model contains topology information and definitions of logical policies that are applied to the device. It also contains information which describes the relationships between the logical policies and the objects to which they apply.

A **definition**, in the context of a service model, defines a logical policy.

A **definition ID** is a value which uniquely identifies a particular logical policy in the service model.

An **association** in the service model links a logical policy definition with an object (e.g. device or interface).

An **association ID** is a value which uniquely identifies the association. The association ID is also used in the Network Processor to link the object with the concrete representing the logical policy in the IP Service Activator core.

> **Note:**
>
> Both the service model to device model and annotated device model to CLI transforms must correctly carry association IDs from the source to the target models. If this is not done correctly, quarantine, failure notifications, and faults will not operate correctly. These will be described in more detail later in this document.

Sample Service Model Associations

This section illustrates how logical policies are associated with objects in the service model using associations.

The following service model snippet from the service model example contains the definition of a logical policy called netflowCollector.

```
<ser:item>
  <ser:id>
```

```
        <ser:id>1578</ser:id>
        <ser:ver>1</ser:ver>
      </ser:id>
      <ser:name>netflowCollector</ser:name>
      <ser:value>
      <ser:discriminator
xsi:type="ser:DefinitionSite.DefinitionType">GenericRuleDefinitionType</
ser:discriminator>
        <ser:generic_rule_def>
          <ser:contentType>collectorParameters</ser:contentType>
          <ser:contentValue>
            <collectorParameters xmlns="http://www.metasolv.com/serviceactivator/
collectorParameters">
              <type>Cisco Netflow FlowCollector</type>
              <externalSystem>
                <primaryIP>
                  <ipAddress>3.3.3.3</ipAddress>
                  <port>32</port>
                </primaryIP>
              </externalSystem>
            </collectorParameters>
          </ser:contentValue>
          <ser:rule_directives/>
          <ser:order>4026531840</ser:order>
        </ser:generic_rule_def>
      </ser:value>
    <ser:options/>
    </ser:item>
```

Note that the **id** section of this logical policy contains the **definition ID** of **1578**. As well, the name is specified just below as **netflowCollector**.

This snippet from the same service model shows a static route logical policy definition.

```
    <ser:item>
      <ser:id>
        <ser:id>7460</ser:id>
        <ser:ver>1</ser:ver>
      </ser:id>
      <ser:name>static route</ser:name>
      <ser:value>
        <ser:discriminator
xsi:type="ser:DefinitionSite.DefinitionType">GenericRuleDefinitionType</
ser:discriminator>
        <ser:generic_rule_def>
          <ser:contentType>staticRoutes</ser:contentType>
          <ser:contentValue>
            <staticRoutes xmlns="http://www.metasolv.com/serviceactivator/
staticroute">
              <staticRoute>
                <ip>20.20.20.20</ip>
                <mask>32</mask>
                <next_hop_ip>21.21.21.21</next_hop_ip>
                <distance_metric>3</distance_metric>
              </staticRoute>
            </staticRoutes>
          </ser:contentValue>
          <ser:rule_directives/>
          <ser:order>805306368</ser:order>
        </ser:generic_rule_def>
      </ser:value>
```

```
      <ser:options/>
    </ser:item>
```

Note that the **id** section of this logical policy contains the **definition ID** of **7460**. As well, the name is specified just below as **static route**.

The snippet of the example service model below contains the associations of these logical policies with an interface.

```
  <ser:interface>
. . .
    <ser:associations>
      <ser:item xsi:type="ser:NetworkProcessor.Association">
        <ser:id>
          <ser:id>1578</ser:id>
          <ser:ver>0</ser:ver>
        </ser:id>
        <ser:assoc_id>7432</ser:assoc_id>
        <ser:outbound>true</ser:outbound>
      </ser:item>
      <ser:item xsi:type="ser:NetworkProcessor.Association">
        <ser:id>
          <ser:id>7460</ser:id>
          <ser:ver>0</ser:ver>
        </ser:id>
        <ser:assoc_id>7462</ser:assoc_id>
        <ser:outbound>true</ser:outbound>
      </ser:item>
    </ser:associations>
  </ser:interface>
```

For this interface, two associations are present. The first association has the **association ID** of **7432**. This association has the **definition id** value of **1578**. This means that the logical policy with the **definition ID** of **1578** is applied to this interface. Looking at the definition snippet above, we can confirm that **1578** identifies the netflow collector policy.

Similarly, the second association has the **association ID** of **7462**. This association has the **definition id** value of **7460**. This means that the logical policy with the **definition ID** of **7460** is applied to this interface. Again, looking at the second definition snippet above, we can confirm that **7460** identifies the static route policy.

Device Model

This section discusses the makeup of a device model.

A device model is a device-specific XML document derived from the service model. It contains a set of data elements that defines the complete device state.

A device model is an extension of the base device model provided by the Network Processor. This extension to the base device model schema is contained in the cartridge and is typically called **deviceModel.xsd**. A device model instance is validated using the **deviceModel.xsd** and **base_devicemodel.xsd** at run time.

The sample device model has highlighted sections similar to the sample service model.

Sample Device Model

```
<lib:device xsi:type="dm:CiscoDevice" xmlns:lib="http://www.metasolv.com/
serviceactivator/devicemodel" xmlns:dm="http://www.metasolv.com/serviceactivator/
```

```
cisco" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <lib:appInfo>
    <lib:version>2.3.0</lib:version>
  </lib:appInfo>
  <lib:componentId>7020</lib:componentId>
  <lib:componentName>rotgsr-1.kanata.ca.oracle.com</lib:componentName>
  <lib:driverType>cisco</lib:driverType>
  <lib:configurationInfo>
    <lib:deviceType>Cisco 12008</lib:deviceType>
    <lib:deviceDescription>Cisco Internetwork Operating System Software
IOS (tm) GS Software (GSR-P-M), Version 12.0(32)S5, RELEASE SOFTWARE (fc2)
Tech</lib:deviceDescription>
    <lib:cartridgeObjid>49</lib:cartridgeObjid>
  </lib:configurationInfo>
  <lib:ipAddress>10.156.68.94</lib:ipAddress>
  <lib:generationId>5</lib:generationId>
  <lib:configlets/>
  <lib:interfaces>
    <lib:interface xsi:type="dm:CiscoInterface">
      <lib:id>7252</lib:id>
      <lib:name>Serial3/0</lib:name>
      <dm:bandwidth>44210</dm:bandwidth>
    </lib:interface>
  </lib:interfaces>
  <lib:configVersion>
    <lib:assoc_id>0</lib:assoc_id>
    <lib:timestamp>2007-11-07T15:15:23.177Z</lib:timestamp>
  </lib:configVersion>
  <lib:schedules/>
  <dm:prefixLists/>
  <dm:staticRoutes>
    <dm:staticRoute smId="7460">
      <lib:assoc_id>7462</lib:assoc_id>
      <dm:networkAddress>20.20.20.20</dm:networkAddress>
      <dm:networkMask>32</dm:networkMask>
      <dm:nextHopIp>21.21.21.21</dm:nextHopIp>
      <dm:distanceMetric>3</dm:distanceMetric>
    </dm:staticRoute>
  </dm:staticRoutes>
  <dm:rtrs/>
<dm:netFlow>
    <dm:export-destinations>
      <dm:export-destination smId="1578">
        <lib:assoc_id>7432</lib:assoc_id>
        <dm:ipAddress>3.3.3.3</dm:ipAddress>
        <dm:port>32</dm:port>
      </dm:export-destination>
    </dm:export-destinations>
  </dm:netFlow>
  <dm:fwModuleGroups/>  <dm:aceModuleGroups/>
</lib:device>
```

Device Model and the Service Model to Device Model Transform

The first transform step is for the service model to be transformed to the device model. To do this, the Network Processor invokes the **<smToDm>** entry that is pointed to in the cartridge instance.

Using the cartridge-supplied transform, the network processor transforms the service model to a device, which is a device/vendor-specific model derived from the service model.

The **sm2Dm** is an Java or XQuery file that navigates the service model and reorganizes the information into the device model document. In addition to the basic conversion to device model, additional processing such as logic to invoke options, may be included at the developers discretion. If the transform is a java transform, then use <**smToDm transformType="java"**> as the entry.

Sample Relating the Service Model to the Device Model

This section shows how key portions of the service model in "Sample Service Model" are transformed to the device model in "Sample Device Model". Logical policies and the objects they are applied to are shown as discrete configuration sections.

netFlow

In this example, you can see the netflow element that combines the netflow logical policy with the configuration details. For the sample services (netflow and static route) although the service is applied conceptually at the interface level, the configuration command is given at the device level.

The example contains references to the unique service model **association id (7432)** and the logical policy definition **smId (1578)** which together generated the following configuration. It also contains relevant configuration data for IP address and port from the logical policy.

```
<dm:netFlow>
  <dm:export-destinations>
    <dm:export-destination smId="1578">
      <lib:assoc_id>7432</lib:assoc_id>
      <dm:ipAddress>3.3.3.3</dm:ipAddress>
      <dm:port>32</dm:port>
    </dm:export-destination>
  </dm:export-destinations>
</dm:netFlow>
```

staticRoute

In this example, you can see the static route element that combines the static route logical policy with the details for the particular interface to which it applies. It contains references to the unique service model **association id (7462)** and the logical policy definition **smId (7460)** which together generated the following configuration. It also contains relevant configuration data for network address, network mask, nextHopIp, and distanceMetric from the logical policy.

```
<dm:staticRoutes>
  <dm:staticRoute smId="7460">
    <lib:assoc_id>7462</lib:assoc_id>
    <dm:networkAddress>20.20.20.20</dm:networkAddress>
    <dm:networkMask>32</dm:networkMask>
    <dm:nextHopIp>21.21.21.21</dm:nextHopIp>
    <dm:distanceMetric>3</dm:distanceMetric>
  </dm:staticRoute>
</dm:staticRoutes>
```

Device Model Validation

If the cartridge registry entry <**dmValidation**> contains a dmValidation entry, the Network Processor will invoke this function to validate the transformed device model. This would capture logical faults as opposed to syntax faults which would be caught by the device model validation using **deviceModel.xsd**. If the transform is a java transform, then use <**dmValidation transformType="java"**> as the entry.

Sample Annotated Device Model

The Network Processor compares the target device model against the last pushed device model that was persisted after the last successful push to the device.

The target device model is compared or annotated against the last pushed device model yielding the annotated device model, which annotates the changes needed to the device's configuration to end up with the configuration desired.

Below is a sample of the annotated device model:

```
<lib:device xsi:type="dm:CiscoDevice" xmlns:lib="http://www.metasolv.com/
serviceactivator/devicemodel" xmlns:dm="http://www.metasolv.com/serviceactivator/
cisco" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <lib:appInfo>
    <lib:version>2.3.0</lib:version>
  </lib:appInfo>
  <lib:componentId>7020</lib:componentId>
  <lib:componentName>rotgsr-1.kanata.ca.oracle.com</lib:componentName>
  <lib:driverType>cisco</lib:driverType>
  <lib:configurationInfo>
    <lib:deviceType>Cisco 12008</lib:deviceType>
    <lib:deviceDescription>Cisco Internetwork Operating System Software
IOS (tm) GS Software (GSR-P-M), Version 12.0(32)S5, RELEASE SOFTWARE (fc2)
Tech</lib:deviceDescription>
    <lib:cartridgeObjid>49</lib:cartridgeObjid>
  </lib:configurationInfo>
  <lib:ipAddress>10.156.68.94</lib:ipAddress>
  <lib:generationId>5</lib:generationId>
  <lib:authenticationTacacs>
    <lib:userName/>
    <lib:password/>
    <lib:enablePassword/>
  </lib:authenticationTacacs>
  <lib:configlets/>
  <lib:interfaces>
    <lib:interface xsi:type="dm:CiscoInterface">
      <lib:id>7252</lib:id>
      <lib:name>Serial3/0</lib:name>
      <dm:bandwidth>44210</dm:bandwidth>
    </lib:interface>
  </lib:interfaces>
  <lib:configVersion dmId="1" changeType="ADD">
    <lib:assoc_id>7432</lib:assoc_id>
    <lib:assoc_id>7462</lib:assoc_id>
    <lib:timestamp>2007-11-07T15:15:23.177Z</lib:timestamp>
  </lib:configVersion>
  <lib:schedules/>
  <dm:prefixLists/>
<dm:staticRoutes>
```

```
    <dm:staticRoute smId="7460" dmId="2" changeType="ADD">
      <lib:assoc_id>7462</lib:assoc_id>
      <dm:networkAddress>20.20.20.20</dm:networkAddress>
      <dm:networkMask>32</dm:networkMask>
      <dm:nextHopIp>21.21.21.21</dm:nextHopIp>
      <dm:distanceMetric>3</dm:distanceMetric>
    </dm:staticRoute>
  </dm:staticRoutes>
  <dm:rtrs/>
<dm:netFlow>
    <dm:export-destinations>
      <dm:export-destination smId="1578" dmId="3" changeType="ADD">
        <lib:assoc_id>7432</lib:assoc_id>
        <dm:ipAddress>3.3.3.3</dm:ipAddress>
        <dm:port>32</dm:port>
      </dm:export-destination>
    </dm:export-destinations>
  </dm:netFlow>
  <dm:fwModuleGroups/>
  <dm:aceModuleGroups/>
</lib:device>
```

Sample Relating the Device Model to the Annotated Device Model

The Network Processor, after comparing the target device model with the last pushed device model, has annotated the device model with these attributes:

- smId - definition ID from the service model
- dmId - device model ID which is created for each command entry to be applied. This ID is generated by the Network Processor by invoking the annotation.
- changeType - attribute indicating the change type: adding, deleting or modifying configuration

The configuration data on the interfaces have been carried forward from previous examples; IP address and port from the netFlow configuration, and networkAddress, networkMask, nextHopIp and distanceMetric from the static route configuration.

```
<dm:netFlow>
    <dm:export-destinations>
      <dm:export-destination smId="1578" dmId="3" changeType="ADD">
        <lib:assoc_id>7432</lib:assoc_id>
        <dm:ipAddress>3.3.3.3</dm:ipAddress>
        <dm:port>32</dm:port>
      </dm:export-destination>
    </dm:export-destinations>
  </dm:netFlow>
. . .
<dm:staticRoutes>
    <dm:staticRoute smId="7460" dmId="2" changeType="ADD">
      <lib:assoc_id>7462</lib:assoc_id>
      <dm:networkAddress>20.20.20.20</dm:networkAddress>
      <dm:networkMask>32</dm:networkMask>
      <dm:nextHopIp>21.21.21.21</dm:nextHopIp>
      <dm:distanceMetric>3</dm:distanceMetric>
    </dm:staticRoute>
  </dm:staticRoutes>
```

> **Note:**
>
> The service model showed that the logical policies were applied to specific interfaces. When you read the device model it may appear that information is missing. Static routes and netflow configuration happen to be configured on the target device type as device level commands, rather than being configured against interfaces. Therefore the static route and netflow information don't show as child elements to interfaces in the device model.
>
> The application of these logical policies on interfaces at the GUI is simply a human viewable artifact that been translated to a device level artifact.

Sample CLI Commands and the Device Model to CLI Transform

The next step is for the annotated device model to be transformed to the CLI commands. To do this, the Network Processor invokes the <**dmToCli**> entry that is pointed to in the cartridge instance. If the transform is a java transform, then use <**dmToCli transformType="java"**> as the entry.

Using the cartridge-supplied transform, the Network Processor transforms the annotated device model to a CLI document, which is a list of native configuration commands to be sent to the device.

The annotated dm2Cli is an Java or XQuery file that navigates the annotated device model and reorganizes the information into the CLI document. In addition to the basic conversion to CLI, additional processing such as logic to invoke options, may be included at the developers discretion.

Sample CLI Document

```
<cmd:commandSession xmlns:cmd="http://www.metasolv.com/serviceactivator/
climodel">
  <cmd:configuration>
    <cmd:ipAddress>10.156.68.94</cmd:ipAddress>
    <cmd:prompt>
      <cmd:matchPattern>([^>#\n]*)[>#]</cmd:matchPattern>
      <cmd:prependPattern>\n</cmd:prependPattern>
      <cmd:appendPattern>(([>#])|(\(.*\)#))</cmd:appendPattern>
      <cmd:errorPattern>.*Username:</cmd:errorPattern>
      <cmd:hostPattern>[\w\.\-]+</cmd:hostPattern>
    </cmd:prompt>
    <cmd:errorPatterns>
      <!--Cisco and frameworkTest-cu1 Error Messages-->
      <cmd:errorPattern>
        <cmd:pattern>(?s).*Invalid input.*</cmd:pattern>
        <cmd:message>Invalid input</cmd:message>
      </cmd:errorPattern>
    </cmd:errorPatterns>
    <cmd:warningPatterns>
      <!--in theory, we could ignore this and continue. The problem is, we get
the same message when removing the policing rule. This means a user could see
the warning, correct the problem, and then see the warning pop back-->
      <cmd:warningPattern blocking="true">
        <cmd:pattern>(?s).*Illegal .* burst size.*Increasing .* burst size.*</
cmd:pattern>
```

```
      </cmd:warningPattern>
      <cmd:warningPattern>
        <cmd:pattern>(?s).*WARNING: \"ip multicast-routing\" is not configured.*</
cmd:pattern>
      </cmd:warningPattern>
      <cmd:warningPattern>
        <cmd:pattern>(?s).*WARNING: CGMP requires PIM enabled on interface</
cmd:pattern>
      </cmd:warningPattern>
      <cmd:warningPattern>
        <cmd:pattern>(?s).*WARNING: RGMP requires PIM enabled on interface</
cmd:pattern>
      </cmd:warningPattern>
      <!--This message indicates "copy running-config startup-config" has failed. This
will produce a warning instead of failing and rolling back commands.-->
      <cmd:warningPattern>
        <cmd:pattern>(?s).*startup-config file open failed.*</cmd:pattern>
      </cmd:warningPattern>
      <cmd:warningPattern>
        <!--cisco: message after Policy Map configured with bandwidth greater than
available"-->
        <cmd:pattern>(?s).*bandwidth is less than requested.*</cmd:pattern>
      </cmd:warningPattern>
      <cmd:warningPattern>
        <!--cisco: message after configuring two instances of the export destination
commands with the same IP address a warning is received (NetFlow)"-->
        <cmd:pattern>(?s).*Second destination address is the same as previous
address.*</cmd:pattern>
      </cmd:warningPattern>
      <cmd:warningPattern>
        <cmd:pattern>(?s).*Warning: portfast should only be enabled on ports connected
to a single.*</cmd:pattern>
        <cmd:message>portfast should only be enabled on ports connected to a single
host. Connecting hubs, concentrators, switches, bridges, etc... to this interface when
portfast is enabled, can cause temporary bridging loops.</cmd:message>
      </cmd:warningPattern>
      <cmd:warningPattern>
        <cmd:pattern>(?s).*Command rejected: .* is a routed port.*</cmd:pattern>
        <cmd:message>Command rejected: port is a routed port.</cmd:message>
      </cmd:warningPattern>
      <cmd:warningPattern>
        <cmd:pattern>(?s).*(w|W)arning: trust value is ignored in 'mls qos queueing-
only' mode.*</cmd:pattern>
        <cmd:message>Trust value is ignored in 'mls qos queueing-only' mode.</
cmd:message>
      </cmd:warningPattern>
      <cmd:warningPattern>
        <cmd:pattern>(?s).*Receive Threshold enabled.*</cmd:pattern>
        <cmd:message>Receive Threshold enabled.</cmd:message>
      </cmd:warningPattern>
    </cmd:warningPatterns>
    <cmd:successPatterns>
      <!--CISCO MESSAGES-->
      <cmd:successPattern>
        <!---reference a policy map that is already applied to this interface-->
        <cmd:pattern>(?s).*QoS policy .+ is already applied.(\\n.*|$)</cmd:pattern>
      </cmd:successPattern>
      <cmd:successPattern>
        <!---cisco : message after "conf t"-->
        <cmd:pattern>(?s).*Enter configuration commands, one per line\..*</cmd:pattern>
      </cmd:successPattern>
```

```
      <cmd:successPattern>
        <!---cisco : message after "no interface"-->
        <cmd:pattern>(?s).*Not all config may be removed.*</cmd:pattern>
      </cmd:successPattern>
      <cmd:successPattern>
        <!---cisco : message after "no ip vrf "-->
        <cmd:pattern>(?s).*IP addresses from all interfaces in VRF .* have been
removed.*</cmd:pattern>
      </cmd:successPattern>
    </cmd:successPatterns>
  </cmd:configuration>
  <cmd:authentication>
    <cmd:prompt>.*Username:</cmd:prompt>
    <cmd:response/>
  </cmd:authentication>
  <cmd:authentication>
    <cmd:prompt>.*Password:</cmd:prompt>
    <cmd:response/>
    <cmd:errorPrompt>.*Username:</cmd:errorPrompt>
  </cmd:authentication>
  <cmd:initialization>
    <cmd:command alwaysRetry="true">
      <cmd:assoc_id>7432</cmd:assoc_id>
      <cmd:assoc_id>7462</cmd:assoc_id>
      <cmd:commandString>terminal length 0</cmd:commandString>
    </cmd:command>
  </cmd:initialization>
  <cmd:precheck/>
  <cmd:preconfig>
    <cmd:command alwaysRetry="true">
      <cmd:assoc_id>7432</cmd:assoc_id>
      <cmd:assoc_id>7462</cmd:assoc_id>
      <cmd:retry>
        <cmd:retryNumber>2</cmd:retryNumber>
        <cmd:waitTime>20</cmd:waitTime>
      </cmd:retry>
      <cmd:commandString>conf t</cmd:commandString>
    </cmd:command>
  </cmd:preconfig>
<cmd:command dmId="3">
    <cmd:assoc_id>7432</cmd:assoc_id>
    <cmd:commandString>ip flow-export destination 3.3.3.3 32</cmd:commandString>
  </cmd:command>
  <cmd:command dmId="2">
    <cmd:assoc_id>7462</cmd:assoc_id>
    <cmd:commandString>ip route 20.20.20.20 32 21.21.21.21 3</cmd:commandString>
  </cmd:command>
  <cmd:postconfig>
    <cmd:command dmId="1" configVersion="true">
      <cmd:assoc_id>7432</cmd:assoc_id>
      <cmd:assoc_id>7462</cmd:assoc_id>
      <cmd:commandString>alias exec IpsaConfigVersion 2007-11-07T15:15:23.177Z</
cmd:commandString>
    </cmd:command>
    <cmd:command alwaysRetry="true">
      <cmd:assoc_id>7432</cmd:assoc_id>
      <cmd:assoc_id>7462</cmd:assoc_id>
      <cmd:commandString>end</cmd:commandString>
    </cmd:command>
  </cmd:postconfig>
  <cmd:postcheck/>
```

```
  <cmd:finalize>
    <cmd:command>
      <cmd:assoc_id>7432</cmd:assoc_id>
      <cmd:assoc_id>7462</cmd:assoc_id>
      <cmd:commandString>copy running-config startup-config</cmd:commandString>
      <cmd:conditionalPrompt>.*Destination filename.*</cmd:conditionalPrompt>
      <cmd:conditionalCommand>
        <cmd:assoc_id>7432</cmd:assoc_id>
        <cmd:assoc_id>7462</cmd:assoc_id>
        <cmd:commandString>startup-config</cmd:commandString>
        <cmd:conditionalPrompt>.*Overwrite the previous NVRAM configuration?.*</
cmd:conditionalPrompt>
        <cmd:conditionalCommand>
          <cmd:assoc_id>7432</cmd:assoc_id>
          <cmd:assoc_id>7462</cmd:assoc_id>
          <cmd:commandString>y</cmd:commandString>
        </cmd:conditionalCommand>
        <cmd:timeoutSeconds>600</cmd:timeoutSeconds>
      </cmd:conditionalCommand>
      <cmd:timeoutSeconds>600</cmd:timeoutSeconds>
    </cmd:command>
    <cmd:command>
      <cmd:assoc_id>7432</cmd:assoc_id>
      <cmd:assoc_id>7462</cmd:assoc_id>
      <cmd:commandString>logout</cmd:commandString>
      <cmd:conditionalPrompt>.*</cmd:conditionalPrompt>
    </cmd:command>
  </cmd:finalize>
</cmd:commandSession>
```

Key sections in the CLI document are:

- Configuration: information on network processor behavior to implement for this device
  - ipAddress: IP address of the device
  - prompt: the device prompt
  - errorPatterns: device response error pattern
  - warningPatterns: device response warning pattern
  - successPatterns: device response success pattern
- Authentication: information about login credentials of the device
- Initialization: commands that may need to be applied to the device as an extended part of the login/setup phase.
- Precheck: commands for checking and validating existing configuration on the device.
- Preconfig: commands needed before the configuration can take place.
- Command: this section contains the actual configuration commands for the changing services
- Postconfig: commands that must be issued once configuration is completed.
- Finalize: commands to be issued once all other configuration and checks are complete.

CLI Elements

Several new elements have been added to the CLI document to allow cartridges to handle some of the variations in communications with devices. Some of the new items added allow you to:

- Specify a terminal type (VT100 is used by default)
- Specify a line terminator (\n is used by default)
- Flush buffers

Sample Relating the Annotated Device Model to the CLI Document

In the sample, we can see that the configuration parameters have reached the CLI stage.

The snippet below contains the dmIDs assigned by the Network Processor, and the final configuration commands based on the original logical policies and interfaces they were applied to.

```
<cmd:command dmId="3">
    <cmd:assoc_id>7432</cmd:assoc_id>
    <cmd:commandString>ip flow-export destination 3.3.3.3 32</cmd:commandString>
  </cmd:command>
  <cmd:command dmId="2">
    <cmd:assoc_id>7462</cmd:assoc_id>
    <cmd:commandString>ip route 20.20.20.20 32 21.21.21.21 3</cmd:commandString>
  </cmd:command>
```

CLI Merging

The CLI models generated by the core/base and service cartridges will all be merged before commands are sent to the device. The CLI models can be broken up into separate sections, each of which is merged together. This provides a means of ordering the commands between the various cartridges.

Merge Section Descriptions

The following lists the merge sections available and what type of commands are expected to be in each:

- Initialization: commands that may need to be applied to the device as an extended part of the login/setup phase. This phase does not perform any configuration related activities.
- Pre-check: commands for checking and validating existing configuration on the device. This should not perform any configuration.
- Preconfig: commands needed before the configuration can take place. This is usually commands needed to put the device into a special mode for accepting new configuration.
- DeleteConfig: This section is for deleting configuration from the device. This configuration cannot include configuration that removes interfaces.
- CreateInterface: special section for creating interfaces.

- Command: special section used by the core cartridges for applying configuration. Should not be used by SDK developers.
- DeleteInterface: section for deleting interfaces. The configuration on these interfaces must be separately removed in the “deleteConfig" section.
- CreateConfig: section for creating or overwriting configuration. This section cannot create interfaces.
- Postconfig: commands that must be issued once configuration is completed. Normally, this is commands to get out of the configuration mode.
- Postcheck: commands for validating the configuration applied to the device.
- Finalize: commands to be issued once all other configuration and checks are complete. This usually includes commands for saving the configuration on the device.

Sample Relating the CLI Document to Configuration Commands

Continuing the example to the configuration stage, the Network Processor command executor (see "Command Executor") would send the following commands to provision the sample netflow collector and static route:

```
ip flow-export destination 3.3.3.3 32
ip route 20.20.20.20 32 21.21.21.21 3
```

The auditTrails would show the following:

```
2007-11-07 10:15:23|10.156.68.94|#Start Configuration
2007-11-07 10:15:23|10.156.68.94|file-interface|#Applying Configuration
2007-11-07 10:15:23|10.156.68.94|file-interface|terminal length 0
2007-11-07 10:15:23|10.156.68.94|file-interface|conf t
2007-11-07 10:15:23|10.156.68.94|file-interface|ip flow-export destination 3.3.3.3 32
2007-11-07 10:15:23|10.156.68.94|file-interface|ip route 20.20.20.20 32 21.21.21.21 3
2007-11-07 10:15:23|10.156.68.94|file-interface|alias exec IpsaConfigVersion
2007-11-07T15:15:23.177Z
2007-11-07 10:15:23|10.156.68.94|file-interface|end
2007-11-07 10:15:23|10.156.68.94|file-interface|copy running-config startup-config
2007-11-07 10:15:23|10.156.68.94|file-interface|logout
2007-11-07 10:15:23|10.156.68.94|#End Configuration
```

If the configuration is successful, a message is sent back to the policy server indicating that associations 7432 and 7462 have been successfully installed. The service model and the device model are then persisted to the database.

Command Executor

Once the CLI document is constructed the Network Processor command executor is invoked to read the CLI and execute the command elements. Its functionality includes managing logging in to a device, command delivery, recognizing success or failure of commands and other housekeeping related to command delivery.

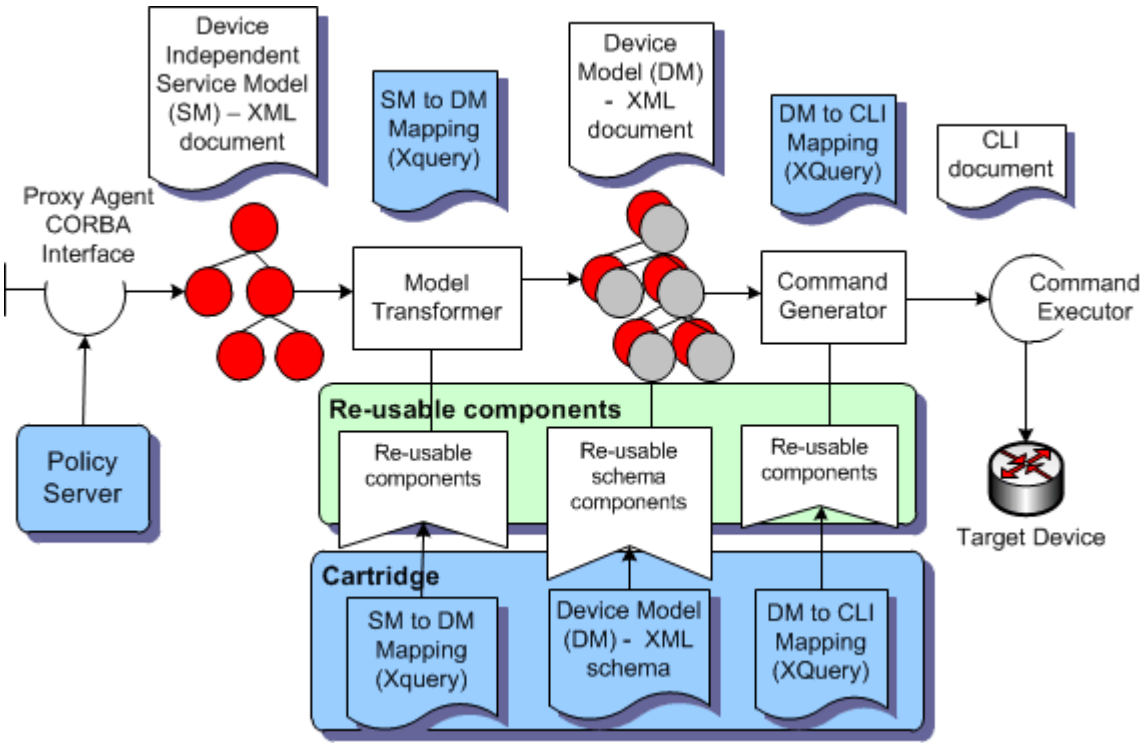
The default command executor provides standard telnet or ssh access to a device and delivers commandString (which is a child element of command) content to the device. For example:

```
<cmd:commandString>conf t</cmd:commandString>
```

Network Processor End to End Flow-Through Illustration

Figure 2-2 illustrates the completed data flow-through described in the preceding pages in this document.

Figure 2-2 Network Processor End to End Flow-Through



Device Model Extension

The Network Processor comes with a **base_devicemodel.xsd** which defines the base device model. Consider this to be a common set of device model information for all device models.

Cartridges generally extend this **base_deviceModel.xsd** to include details of new services that are to be administered.

Changeables and Identifiables

When writing commands to a device, there are three possible user actions:

- Add: adding a command to the device
- Delete: deleting a command from the device
- Modify: modifying an existing command on the device by overwriting it.

In order for a cartridge to support these actions, metadata (changeType) is added to the device model to indicate what action is taking place, which occurs in the annotate device model stage. The target device model shows what the intended device model

looks like. The annotated device model shows what is being added/deleted/modified in comparison to the previously pushed device model.

To identify if a command or a command field is addable, deleteable, and or modifiable, in extending the device model, you need to specify what aspects of the device extension are changeable.

Change Types

ChangeTypes are attributes inserted into the device model during the annotation process. They specify which configuration is to be added or deleted.

The cartridge implementation then uses these attributes to decide how to send the configuration to the device.

Generally speaking, when you write your device model, you will be adding content that models the configuration (commands) that you are administering. As part of that model, if the commands are add-able and delete-able you need to mark those commands as “changeable" This is done by extending “lib:Changeable" found in the base_devicemodel.xsd. This allows the Network Processor to annotate the device model with changeType attributes for the purpose of identifying what action is taking place.

Example

n the device model fragment seen below, if the target device model has redistributeStatic and the last device model does not, then redistributeStatic would be annotated as **add**.

If the target device model does not have redistributeStatic and the last device model does, then redistributeStatic would be annotated with **delete**.

If the target device model has redistributeStatic and the last device model has redistributeStatic and its value has changed, then redistributeStatic would be annotated with **delete** and **add**. The cartridge, in this case, would be expected to send both a delete command followed by an add command to the device to realize the target configuration.

If the target device model has redistributeStatic and the last device model has redistributeStatic and its value has not changed, then redistributeStatic would not be annotated with attribute changeType.

As example, all the elements in the complex type EigrpVrfAddressFamily are changeable.

```
<xs:complexType name="EigrpVrfAddressFamily">
  <xs:complexContent>
    <xs:extension base="lib:Changeable">
      <xs:sequence>
        <xs:element name="vrfRef" type="xs:string"/>
        <xs:element name="vrfRdRef" type="dm:VpnTag"/>
        <xs:element name="redistributeStatic" type="dm:RedistributeProtocol"
minOccurs="0"/>
        <xs:element name="redistributeConnected" type="dm:RedistributeProtocol"
minOccurs="0"/>
        <xs:element name="redistributeBgp" type="dm:RedistributeProtocol"
minOccurs="0"/>
        <xs:element name="redistributeRip" type="dm:RedistributeProtocol"
minOccurs="0"/>
        <xs:element name="networkStatements" type="dm:NetworkStatements"
minOccurs="0"/>
        <xs:element name="autoSummary" type="dm:ChangeableBoolean" minOccurs="0"/>
        <xs:element name="autonomousSystem" type="xs:unsignedInt"/>
        <xs:element name="maximumPaths" type="dm:ChangeableInteger" minOccurs="0"/>
```

```
        <xs:element name="logNeighbourChanges" type="dm:ChangeableBoolean"
minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

When the Network Processor executes the annotation, it will mark each of the above elements with **add** or **delete** or nothing based on the action taking place.

Changeable and Identifiable

If a command can be added, deleted and/or modified then you need to further model that command as **changeable and identifiable**.

Identifiable is terminology that conveys what aspect of the command is modifiable. This is done by extending **lib:Container** found in **base_devicemodel.xsd**.

Example

A complex type EigrpVrfAddressFamily extends lib:Changeable, so all these elements can be annotated as added/deleted.

```
<xs:complexType name="EigrpVrfAddressFamily">
  <xs:complexContent>
    <xs:extension base="lib:Changeable">
      <xs:sequence>
        <xs:element name="vrfRef" type="xs:string"/>
        <xs:element name="vrfRdRef" type="dm:VpnTag"/>
        <xs:element name="redistributeStatic" type="dm:RedistributeProtocol"
minOccurs="0"/>
        <xs:element name="redistributeConnected" type="dm:RedistributeProtocol"
minOccurs="0"/>
        <xs:element name="redistributeBgp" type="dm:RedistributeProtocol"
minOccurs="0"/>
        <xs:element name="redistributeRip" type="dm:RedistributeProtocol"
minOccurs="0"/>
        <xs:element name="networkStatements" type="dm:NetworkStatements"
minOccurs="0"/>
        <xs:element name="autoSummary" type="dm:ChangeableBoolean"
minOccurs="0"/>
        <xs:element name="autonomousSystem" type="xs:unsignedInt"/>
        <xs:element name="maximumPaths" type="dm:ChangeableInteger"
minOccurs="0"/>
        <xs:element name="logNeighbourChanges" type="dm:ChangeableBoolean"
minOccurs="0"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

EigrpVrfAddressFamily is used in Eigrp which extends lib:Container. This indicates that these three elements can be added, deleted, and modified.

```
<xs:complexType name="Eigrp">
  <xs:complexContent>
    <xs:extension base="lib:Container">
      <xs:sequence>
        <xs:element name="asn" type="xs:int"/>
        <xs:element name="logNeighbourChanges" type="dm:ChangeableBoolean"
minOccurs="0"/>
```

```
        <xs:element name="eigrpVrfAddressFamily" type="dm:EigrpVrfAddressFamily"
minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Eigrp is used in eigrp where a key is defined.

```
<xs:element name="eigrp" type="dm:Eigrp" minOccurs="0">
  <xs:key name="eigrpVrfAddressFamilyKey">
    <xs:selector xpath="dm:eigrpVrfAddressFamily"/>
      <xs:field xpath="dm:vrfRef"/>
      <xs:field xpath="dm:vrfRdRef"/>
      <xs:field xpath="dm:autonomousSystem"/>
    </xs:key>
</xs:element>
```

Reading backwards, there is an element **eigrp**, which defines a key composed of three entities called **vrfRef**, **vrfRdRef** and **autonomousSystem**.

If the target and previously pushed device model have the same values for the three elements defined as keys and a change of any type was detected on the other children elements, then eigrp would be annotated with **modify**. The cartridge implementation would then send the appropriate eigrp command taking into account that this is a modify action.

If the target and previously pushed device model have different values for any of the three elements defined as keys and a change of any type was detected on the other child elements, then eigrp would be annotated with **delete** and **add**. The cartridge implementation would then send the appropriate eigrp command taking into account that a delete command must be sent followed by an add command.

3
Cartridge Overview

This chapter outlines the makeup of Oracle Communications IP Service Activator cartridges, introduces base and service cartridges as well as configuration policies, and provides an overview of cartridge registration.

Introduction to Cartridges

A cartridge is a vendor-specific implementation of a set of services for a given family of device types running the same operating system. A cartridge is a set of files (shipped as a .jar file) used by the IP Service Activator Network Processor to implement these services.

Cartridges provide the required components for configuration to flow from the service model to the device for a particular service. The transforms required for the Network Processor to take the configuration from service model to device model, and annotated device model to CLI are included in the cartridge.

Cartridge components include:

- Device Model schema definition
- Service Model to Device Model transform
- Device Model validation
- Annotated Device Model to CLI transform
- Message (success/warning/error) pattern definitions
- Options definitions
- Audit command definitions
- Capabilities definitions
- Logging configuration
- Re-usable (shared library) definitions
- Java beans
- Uninstall information

Cartridges are built from a collection of source files generated by the SDK, which you then customize to implement the desired services.

Cartridge source files include:

- .xsd schema definition files
- .xml instance data files
- .xq (xQuery transformations) or .java (Java transformations)

There are two main types of cartridge: base and service cartridges.

Base Cartridges

Base cartridges provide a framework for allowing the Network Processor to perform basic communication functions with a device. These functions include logging in and out of the device, sending commands or configlets, performing audits, and interpreting responses from the device as successes, warnings, or failures.

Base cartridges do not contain implementations of services. Additional services targeting specific vendor device types are added through integrated service cartridges.

Refer to *IP Service Activator SDK Base Cartridge Developer Guide* for details about creating base cartridges.

Service Cartridges

A service cartridge provides the implementation of a logical service, such as a static route policy, for a specific vendor. The service cartridge also provides the ability to send commands related to its logical service. The service cartridge is also responsible for providing the necessary information to audit and interpret success, warning, or failure responses for commands it sends.

You can use the SDK to create service cartridges. Service cartridges are not independently deployable — they must be deployed as extensions to either a base or a core cartridge.

Refer to *IP Service Activator SDK Service Cartridge Developer Guide* for details about creating service cartridges.

Configuration Policies

A configuration policy provides a GUI form and a schema to collect data for a service. Configuration policies require service cartridges to implement the service on specific devices.

Refer to *IP Service Activator SDK Configuration Policy Extension Developer Guide* for details about creating configuration policies.

Cartridge Registration

Cartridges are identified to the IP Service Activator Network Processor through registry files.

When the Network Processor starts up, it loads all the cartridge registries and internally creates a map of which cartridges administer which devices. Therefore, when a device is edited in the IP Service Activator client, the correct cartridge to administer that change can be called.

The driver type, device type and operating system version attributes in the registry entry allow the Network Processor to map particular devices to a cartridge. When a service, which a cartridge supports, is applied to a device, the Network Processor determines the correct cartridge to use to configure the service based on this registration.

The cartridge operation support files are identified to the Network Processor as entries in the registry files. These files are discussed in Cartridge Overview .

The registry files that identify cartridges to the Network Processor include:

- **MIPSA_registry.xml**: core cartridge registry file
- **Registry.xml**: base cartridge registry file
- **Extension.xml**: service cartridge registry file
- **Customization.xml**: registry file used to override certain entries in the Registry.xml or Extension.xml files. These are typically placed in *Service_Activator_home***Config\networkProcessor\Custom\Registries**.
- **ConfigPolicyRegistry.xml**: registry information for the configuration policy used to integrate it with a service cartridge.

> **Note:**
>
> The Network Processor will detect and process any files ending in .xml from this directory.

Base Cartridge Registry.xml

The **Registry.xml** file identifies a base cartridge instance to the Network Processor and further indicates when the resources provided by that cartridge (i.e. transforms, validation script) are to be invoked.

A sample **Registry.xml** follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<registry
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://www.metasolv.com/service activator/networkprocessor/cartridgeregistry/"
  xmlns:cs="http://www.metasolv.com/service activator/networkprocessor/
cartridgesubscription/">

  <cartridge>
    <name>cisco</name>
    <driverType>cisco</driverType>
    <transforms>
      <smToDm>com/metasolv/service activator/cartridges/cisco/transforms/sm2dm.xq</
smToDm>
      <dmValidation>com/metasolv/service activator/cartridges/cisco/transforms/
dmValidation.xq</dmValidation>
      <dmToCli>com/metasolv/service activator/cartridges/cisco/transforms/
annotatedDm2Cli.xq</dmToCli>
    </transforms>
    <audit>
      <auditTemplate>
        <auditTemplateEntry>
          <auditTemplateFile>com/metasolv/service activator/cartridges/cisco/audit/
auditTemplate.xml</auditTemplateFile>
            <appliesTo>
            <deviceTypes useRegex="true">Cisco.*</deviceTypes>
            <osVersions useRegex="true">.*</osVersions>
          </appliesTo>
```

```
        </auditTemplateEntry>
      </auditTemplate>
    </audit>
    <messages>
      <success>com/metasolv/service activator/cartridges/cisco/messages/
successMessages.xml</success>
      <warning>com/metasolv/service
activator/cartridges/cisco/messages/warningMessages.xml</warning>
      <error>com/metasolv/service activator/cartridges/cisco/messages/
errorMessages.xml</error>
    </messages>
    <capabilities>
      <capabilitiesEntry>
        <capsFile>com/metasolv/service activator/cartridges/cisco/capabilities/
empty_caps.xml</capsFile>
          <appliesTo>
            <deviceTypes useRegex="true">Cisco.*</deviceTypes>
            <osVersions useRegex="true">.*</osVersions>
          </appliesTo>
        </capabilitiesEntry>
      </capabilities>
    <options/>
  <commandExecutor>com.metasolv.service
activator.networkprocessor.DefaultCommandExecutor</commandExecutor>
  </cartridge>
</registry>
```

The relevant sections are:

- **Name**: uniquely identifies a base cartridge instance
- **DriverType**: identifies to which family of devices this cartridge instance is applicable. The driverType value must be unique in an IP Service Activator Network Processor installation. For example, if a base cartridge is deployed to a Network Processor installation with an existing IP Service Activator core cartridge deployed, the driverType values must be different. The same is true if an additional base cartridge is deployed. The following driverType values are already assigned to existing core cartridges: Cisco, Huawei, Juniper, and CatOS. When base cartridge source files are generated, <sdk_global_deviceName> is assigned a lowercase version of <driverType> from the **Registry.xml** file. See the IP Service Activator online Help for further information.
- **Transforms**: identifies the location of transform and validation files
- **Audit**: identifies the location of the audit files. Indicates when an audit file is applicable using the appliesTo element, which specifies deviceType and osVersion characteristics.
- **Messages**: identifies locations of the success, warning and error message pattern files
- **Capabilities**: identifies the location of the capabilities file and indicates when this file is applicable, using the appliesTo element
- **Options**: identifies the location of the options file and indicates when this file is applicable, using the appliesTo element
- **CommandExecutor**: identifies the package location of the command executor

The above registry would apply to any device with the driver type cisco. The audit template specified will apply to any device types that begin with Cisco and any OS version because of the wildcard entry '.*'.

Service Cartridge Extension.xml

The **Extension.xml** file identifies a service cartridge instance to the Network Processor and indicates when the resources provided by that cartridge (i.e. transforms, validation scripts) are to be invoked.

A sample **Extension.xml** follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<registry
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://www.metasolv.com/service activator/networkprocessor/
cartridgeregistry/"
    xmlns:cs="http://www.metasolv.com/service activator/networkprocessor/
cartridgesubscription/">
    <extension>
      <name>ciscoBanner</name>
      <subscriptions>
        <cs:configPolicyContentTypeList>
          <cs:configPolicyContentType>banners</cs:configPolicyContentType>
        </cs:configPolicyContentTypeList>
      </subscriptions>
      <transforms>
        <smToDm>com/metasolv/service activator/cartridges/ciscobanner/transforms/
sm2dm.xq</smToDm>
        <dmValidation>com/metasolv/service activator/cartridges/ciscobanner/
transforms/dm-validation.xq</dmValidation>
        <dmToCli>com/metasolv/service activator/cartridges/ciscobanner/transforms/
annotated-dm2cli.xq</dmToCli>
        <dmMigration>com/metasolv/service activator/cartridges/ciscobanner/
xquerylib/dm-migration.xq</dmMigration>
      </transforms>
      <audit>
        <auditTemplate>
          <auditTemplateEntry>
            <auditTemplateFile>com/metasolv/service activator/cartridges/ciscobanner/
audit/auditTemplate.xml</auditTemplateFile>
            <appliesTo>
            <deviceTypes useRegex="true">Cisco.*</deviceTypes>
              <osVersions useRegex="true">.*</osVersions>
            </appliesTo>
          </auditTemplateEntry>
        </auditTemplate>
      </audit>
      <messages>
        <success>com/metasolv/service activator/cartridges/ciscobanner/messages/
successMessages.xml</success>
        <warning>com/metasolv/service activator/cartridges/ciscobanner/messages/
warningMessages.xml</warning>
        <error>com/metasolv/service activator/cartridges/ciscobanner/messages/
errorMessages.xml</error>
      </messages>
    </extension>
</registry>
```

The relevant sections are:

- **Name**: uniquely identifies a service cartridge instance

- **Subscriptions**: identifies a list of policy types that this cartridge will administer. In this case, this service cartridge provides an implementation of the banners configuration policy for cisco devices.
- **Transforms**: identifies the location of transform and validation files
- **Audit**: identifies the location of the audit files. Indicates when an audit file is applicable using the appliesTo element, which specifies deviceType and osVersion characteristics.
- **Messages**: identifies locations of the success, warning and error message pattern files
- **Capabilities**: identifies the location of the capabilities file and indicates when this file is applicable, using the appliesTo element
- **Options**: identifies the location of the options file and indicates when this file is applicable, using the appliesTo element
- **VrfReductionStrategy**: identifies location of a specific strategy file

A service cartridge extends a base cartridge by its placement in a child directory of the base cartridge called **ServiceCartridges**. The service cartridge does not contain an explicit reference indicating which base cartridge it extends.

The audit template specified in the example will apply to any device types that begin with 'Cisco' and any OS version because of the wildcard entry '.*'.

About Subscriptions

Subscriptions, which are elements in service cartridges' **Extension.xml** file, indicate the functionality provided by the service cartridge. This lets you subdivide the service administration functionality into many smaller service cartridges.

For example, a service cartridge instance could subscribe to netflowCollector policies. It would therefore be responsible for administering that policy. Another service cartridge instance could subscribe to staticRoute and would be responsible for administering that policy.

Alternatively, a service cartridge instance could subscribe to both netflowCollector and staticRoute policies. It would be responsible for administering both these policies.

When the Network Processor encounters a policy in the service model which has no service cartridge subscription, it falls back to the base cartridge to locate support for the policy.

Definition Type

Use definition type subscriptions to subscribe to IP Service Activator services implemented in the product core object model. The following is an example from an **Extension.xml** file:

```
<subscriptions>
  <cs:definitionTypeList>
    <cs:definitionType>PolicingRuleDefinitionType</cs:definitionType>
  </cs:definitionTypeList>
</subscriptions>
```

Configuration Policy Identification

Configuration policies are identified to the Network Processor through the service cartridges which implement their services. (Configuration policies require service cartridges to administer their services on specific devices.)

As mentioned above, the **Extension.xml** file contains a subscription section which is used to identify which policies the service cartridge supports.

In this example, the bannerSample configuration policy is specified as being supported by the service cartridge.

```
. . .
  <subscriptions>
    <cs:configPolicyContentTypeList>
      <cs:configPolicyContentType>bannerSample</cs:configPolicyContentType>
    </cs:configPolicyContentTypeList>
  </subscriptions>
. . .
```

Configuration Policy ConfigPolicyRegistry.xml

The **ConfigPolicyRegistry.xml** file identifies a configuration policy instance to the Network Processor.

A sample **ConfigPolicyRegistry.xml** looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<config_policy_registration
        xmlns="http://www.metasolv.com/serviceactivator/configurationpolicy">
    <name>bannersSample</name>
    <version>1.0</version>
</config_policy_registration>
```

The relevant sections are:

- **Name**: unique name to identify the policy
- **Version**: version number of this policy to distinguish this policy for upgrade purposes

The configuration policy is the GUI that the user would see. The administration of that GUI is handled within the service cartridge. So a given ConfigPolicyRegistration is only functional if there is a service cartridge that administers that config policy.

4
Cartridge Operations

This chapter describes how cartridge resources are used to support cartridge operations.

Cartridge operations supported by XML configuration files include:

- **Audits**: comparison of Oracle Communications IP Service Activator's internal representation of what is configured on a device, and the actual device configuration. An audit template file is used to configure the audit and its report. Audit synonyms are used to identify commands that are logically equivalent.
- **Options**: allows for variation in command syntax by certain device type and OS combinations within device families.
- **Capabilities**: indicates what supported capabilities should be reported back to the IP Service Activator policy server when a device supported by a cartridge is discovered.
- **Message Definition**: patterns defining responses from the device
- **Pre- and Post-Checks**: provide the ability verify information on a device when the annotated DM to CLI transform executes, before the general configuration is sent.
- **Cartridge Version**: The cartridge version is used as the version of the device model that the cartridge generates. If you make significant modifications which include device model schema changes to a cartridge that is already deployed, you will need to upgrade the device model.

In addition, the SDK supports the configuration of cartridges to enable the use of services from the Configuration Management product.

Audits

Audits identify discrepancies between IP Service Activator's representation of a device's configuration and the actual configuration currently on the device. For an in depth analysis of the configuration running on a device, an IP Service Activator user can perform a device audit or a per-service audit. You define auditTemplates and audit synonyms that tailor the audit report output.

Audit Template

Audit templates define filter patterns to be applied to commands to identify configuration of interest, and to affect their inclusion in the audit report, and to set attributes on the command results, which, when viewed using a stylesheet will affect how they are displayed to the IP Service Activator user.

Commands to show configuration, and commands to logout, and all commands that can be applied to the device by the cartridge should be listed. All commands that can be applied to the device by a service cartridge should be listed.

> **Note:**
>
> For a service cartridge, do not include commands to show configuration and commands to logout; such commands should be listed in the audit template file for the base cartridge that this service cartridge extends.

Audit templates for base cartridges have two command sets:

- First command set: the device commands to list a device's entire configuration
- Second command set: a complete set of the commands that the cartridge is capable of sending to the device

Audit templates for service cartridges have only the second command set.

For a service cartridge, the first command set is not needed; the first command set from the base cartridge's audit template file describes the commands that the Network Processor must issue to the device to retrieve the device's entire configuration for the base cartridge, and all of the service cartridges that extend the services of the base cartridge.

A sample **auditTemplate.xml** file follows:

```
<commandSession xsi:schemaLocation="http://www.metasolv.com/serviceactivator/
climodel  file:../../../../networkprocessor/climodel/cliModel.xsd" xmlns="http://
www.metasolv.com/serviceactivator/climodel" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance">
  <command>
    <!--commands to show configuration-->
    <commandId>show-commands</commandId>
    <commandString/>
    <command>
      <commandId>show-commands.terminal_length_0</commandId>
      <commandString>terminal length 0</commandString>
    </command>
    <command kind="show">
      <commandId>show-commands.show_running-config</commandId>
      <commandString>show running-config</commandString>
      <conditionalPrompt>.*(?m)^end$.*</conditionalPrompt>
    </command>
    <command>
      <commandId>show-commands.logout</commandId>
      <commandString>logout</commandString>
      <conditionalPrompt>.*</conditionalPrompt>
    </command>
  </command>
  <command>
    <!--context specific filter patterns to identify configuration of interest-->
    <commandId>cisco</commandId>
    <commandString/>
    <command kind="ALWAYSIGNORE">
      <commandId>^ignore^</commandId>
      <commandString>^Building configuration.*</commandString>
    </command>
    <command kind="ALWAYSIGNORE">
      <commandId>^ignore^</commandId>
      <commandString>Current configuration.*</commandString>
    </command>
```

```
<command kind="ALWAYSIGNORE">
  <commandId>^ignore^</commandId>
  <commandString>^exit$</commandString>
</command>
<command kind="IGNORE">
  <commandId>^ignore^</commandId>
  <commandString>^terminal length 0$</commandString>
</command>
<command kind="IGNORE">
  <commandId>^ignore^</commandId>
  <commandString>^show .*</commandString>
</command>
<command kind="IGNORE">
  <commandId>^ignore^</commandId>
  <commandString>^conf t$</commandString>
</command>
<command kind="IGNORE">
  <commandId>^ignore^</commandId>
  <commandString>^end$</commandString>
</command>
<command kind="IGNORE">
  <commandId>^ignore^</commandId>
  <commandString>^copy</commandString>
</command>
<command kind="IGNORE">
  <commandId>^ignore^</commandId>
  <commandString>^logout$</commandString>
</command>
<command>
  <commandId>cisco.vlan</commandId>
  <commandString>^vlan \d+$</commandString>
  <command kind="IGNORE">
    <commandId>^ignore^</commandId>
    <commandString>^$</commandString>
  </command>
</command>
<command kind="IGNORE">
  <commandId>^ignore^</commandId>
  <commandString>^default interface .*$</commandString>
</command>
<command>
  <commandId>cisco.interface</commandId>
  <commandString>^interface</commandString>
  <command>
    <commandId>cisco.interface.description</commandId>
    <commandString>^description .*$</commandString>
  </command>
  <command>
    <commandId>cisco.interface.mtu</commandId>
    <commandString>^mtu \d+$</commandString>
  </command>
  <command>
    <commandId>cisco.interface.ip_address</commandId>
    <commandString>^(no\s)?ip address</commandString>
  </command>
  <command>
    <commandId>cisco.interface.shutdown</commandId>
    <commandString>^shutdown$</commandString>
</command>
  <command kind="IGNORE">
    <commandId>^ignore^</commandId>
```

```
  <commandString>^no shutdown$</commandString>
</command>
<command kind="IGNORE">
  <commandId>^ignore^</commandId>
  <commandString>^no logging event bundle-status$</commandString>
</command>
<command kind="IGNORE">
  <commandId>^ignore^</commandId>
  <commandString>^no logging event trunk-status$</commandString>
</command>
<command kind="IGNORE">
  <commandId>^ignore^</commandId>
  <commandString>^no logging event link-status$</commandString>
</command>
<command>
  <commandId>cisco.interface.logging_event_bundle-status</commandId>
  <commandString>^logging event bundle-status$</commandString>
</command>
<command>
  <commandId>cisco.interface.logging_event_trunk-status</commandId>
  <commandString>^logging event trunk-status$</commandString>
</command>
<command>
  <commandId>cisco.interface.logging_event_link-status</commandId>
  <commandString>^logging event link-status$</commandString>
</command>
<command>
  <commandId>cisco.interface.mls_qos_trust</commandId>
  <commandString>^mls qos trust \w*$</commandString>
</command>
<command>
  <commandId>cisco.interface.switchport</commandId>
  <commandString>^switchport$</commandString>
</command>
<command>
  <commandId>cisco.interface.switchport_trunk_encapsulation</commandId>
  <commandString>^switchport trunk encapsulation \w+$</commandString>
</command>
<command>
  <commandId>cisco.interface.switchport_trunk_allowed_vlan</commandId>
  <commandString>^switchport trunk allowed vlan</commandString>
</command>
<coommand kind="IGNORE">
  <commandId>^ignore^</commandId>
  <commandString>^switchport access vlan 1$</commandString>
</command>
<command>
  <commandId>cisco.interface.switchport_access_vlan</commandId>
  <commandString>^switchport access vlan \d+$</commandString>
</command>
<command>
  <commandId>cisco.interface.switchport_mode</commandId>
  <commandString>^switchport mode \w+$</commandString>
</command>
<command>
  <commandId>cisco.interface.switchport_port-security</commandId>
  <commandString>^switchport port-security$</commandString>
</command>
<command>
  <commandId>cisco.interface.switchport_port-security_maximum</commandId>
  <commandString>^switchport port-security maximum \d+$</commandString>
```

```
      </command>
      <command>
        <commandId>cisco.interface.switchport_port-security_violation</commandId>
        <commandString>^switchport port-security violation \w+$</commandString>
      </command>
      <command>
        <commandId>cisco.interface.switchport_port-security_mac-address_sticky</
commandId>
        <commandString>^switchport port-security mac-address sticky$</commandString>
      </command>
      <command>
        <commandId>cisco.interface.switchport_nonegotiate</commandId>
        <commandString>^switchport nonegotiate.*$</commandString>
      </command>
      <command>
        <commandId>cisco.interface.spanning-tree_portfast</commandId>
        <commandString>^spanning-tree portfast.*$</commandString>
      </command>
      <command>
        <commandId>cisco.interface.spanning-tree_bpduguard</commandId>
        <commandString>^spanning-tree bpduguard.*$</commandString>
      </command>
      <command>
        <commandId>cisco.interface.duplex</commandId>
        <commandString>^duplex \w+$</commandString>
      </command>
      <command>
        <commandId>cisco.interface.speed</commandId>
        <commandString>^speed \w+$</commandString>
      </command>
      <command>
        <commandId>cisco.interface.media-type</commandId>
        <commandString>^media-type \S+$</commandString>
      </command>
    </command>
    <!--Alias command for configuration version -->
    <command configVersion="true" reportManualConfig="true">
      <commandId>cisco.alias_IpsaConfigVersion</commandId>
      <commandString>^alias exec IpsaConfigVersion \S+</commandString>
    </command>
  </command>
</commandSession>
```

In the above **auditTemplate.xml** sample, the section under the comment `<!--commands to show configuration-->` contains the commands to retrieve the device's current configuration.

The section under the comment `<!--context specific filter patterns to identify configuration of interest-->` identifies all the commands the cartridge is capable of sending to the device.

When you populate the **auditTemplate.xml** file as part of the cartridge development process, you must specify every command you intend to support, otherwise the audit function will not work correctly.

When an audit is requested for the device, the Network Processor sends the first command set to the device to retrieve its configuration, which is filtered against the commands in the second command set. Commands that match those configured by the cartridge (i.e. listed in the second command set) are converted into a CLI document.

The Network Processor then invokes the annotated device model to CLI transform from the cartridge on the last pushed (i.e. persisted) device model (which matches IP Service Activator's internal representation of what is configured on the device).

A comparison is made between the two CLI documents. This comparison can highlight discrepancies including:

- Commands sent to the device that are now missing
- Commands added to the device that are not in the Network Processor device model (i.e. manually configured outside of IP Service Activator)
- Commands that are in the wrong order on the device. (In some cases command ordering affects how the device works.)

Audit Template Command Attributes

Audit template attributes are used to accommodate variations in device responses and to customize the audit report. You can apply these attributes to each individual command element to tell the audit to treat this command in a particular manner.

For example, when you read device configuration, the device could:

- display a command in lower case where IP Service Activator provisioned the device in upper case
- display commands indented incorrectly (resulting in wrong context interpretation)

Attributes can be used to ensure the expected command format is displayed in the audit report.

You can use attributes to indicate that specific commands need to follow one another in a particular order.

The command attributes are:

- kind=*string*
- configVersion=*boolean*
- ordered=*string*
- autoIndentUtil=*string*
- ignoreCase=*string*
- reportManualConfig=*boolean*
- brokenIndentRulesOffset=*integer*

kind=*string*

The kind attribute affects how the command is used to match commands on the device. Table 4-1 lists the possible string values.

Table 4-1 Value Strings for Kind Attribute

Value	Effect
ALWAYSIGNORE	Excludes the command from the audit report when found in any context.

Table 4-1 (Cont.) Value Strings for Kind Attribute

Value	Effect
IGNORE	Excludes the command from the audit report when found in this context.
IGNORE_CASE	Performs a case insensitive comparison with the command on the device.
IGNORE_CHARS	Performs a comparison with commands on the device, ignoring characters that follow the attribute (e.g. <command kind="IGNORE_CHARS -">). A dash (-) appears in the device configuration, however, you want the audit to ignore it.
CHECK_DEFAULT	Performs a comparison with commands on the device marking missing commands as “potentially" missing. This is used for special handling in cases where devices don't display default values in a command string.
PARTIAL	Performs a comparison with commands on the device taking the command element name (as depicted in the audit template) as part of the full command. This allows the audit to determine equality between IP Service Activator provisioned commands and those seen on the device using a partial comparison only.

Table 4-2 shows how the audit template command kind attribute affects the inclusion of the command, and its kind attribute value in the resulting audit report.

Table 4-2 Effect of Kind Attribute on Audit Report

#	Cmd in IP Service Activator	Cmd on Device	Command in Audit Template	Kind in Resulting Audit Report for showAll=false (IPSA)	Kind in Resulting Audit Report for showAll=true (CM)
1.	C1	C1	Present	NORMAL	--
2.	C1	C1	Present + kind=CHECK_DEFAULT	NORMAL	--
3.	C1	C1	Present + kind=IGNORE	*exclude*	UNMANAGED
4.	C1	C1	Not Present	*exclude*	NORMAL
5.	C1	C1'	Present | Present + kind=PARTIAL	CHANGED	CHANGED (Missing: Conflickt)
6.	C1	C1'	Present + kind=PARTIAL_DISPLAY_ALL	MISSING	CHANGED (Missing: Conflict)
7.	C1	C1'	Present + kind="CHECK_DEFAULT"	CHANGED	CHANGED (Potential: Conflict)
8.	C1	C1'	Present + kind="IGNORE"	*exclude*	UNMANAGED
9.	C1	C1'	Not Present	*exclude*	CHANGED (Missing: Conflict)

Table 4-2 (Cont.) Effect of Kind Attribute on Audit Report

#	Cmd in IP Service Activator	Cmd on Device	Command in Audit Template	Kind in Resulting Audit Report for showAll=false (IPSA)	Kind in Resulting Audit Report for showAll=true (CM)
10.	C1	missing	Present	MISSING	--
11.	C1	missing	Present + kind="CHECK_DEFAULT"	POTENTIAL	--
12.	C1	missing	Present + kind="IGNORE"	*exclude*	*exclude*
13.	C1	missing	Not Present	*exclude*	MISSING
14.	--	C1	Present	CONFLICT	--
15.	--	C1	Present + kind="CHECK_DEFAULT"	POTENTIAL	--
16.	--	C1	Present + kind="IGNORE"	*exclude*	UNMANAGED
17.	--	C1	Not Present	*exclude*	UNMANAGED

Note:

In Table 4-2:

- C1' means the command is the same, but some of its arguments are different.
- CHANGED is a JuniperXML only tag; for others, this will show up as a MISSING:CONFLICT pair

configVersion=*boolean*

Set configVersion to **true** on the command that sets the IP Service Activator configuration version. The IP Service Activator configuration version is a timestamp of when the configuration was last modified by IP Service Activator.

ordered=*string*

The ordered attribute lets you control whether or not the sequencing of commands is relevant when determining whether or not discrepancies exist between the device and the IP Service Activator device model. Define a string value, and set the ordered attribute value to the same string value for all commands at the same nesting level for which the sequencing is relevant. The correct sequence of the commands will be the order they appear in the audit template.

For example, to specify that if commands "cmd B", and "cmd D" both appear under command "cmd X", then "cmd D" must appear after "cmd B", set ordered="OrderBD" for both "cmd B", and "cmd D".

In this case, no discrepancies will be found since B, and D appear in the correct sequence relative to each other:

```
cmd X
  cmd A
  cmd B
  cmd C
  cmd D
  cmd E
```

In this case, discrepancies will be found since B, and D do not appear in the correct sequence relative to each other:

```
cmd X
  cmd D
  cmd A
  cmd B
  cmd C
  cmd E
```

autoIndentUntil=*string*

The autoIntendUntil attribute tells the audit that every command under the current command is a child command until *string* is detected, then context of the current command will be exited. This attribute is used to deal with bad indenting problems.

For example, to tell the audit that commands B and C are children of command A, despite bad indenting:

```
autoIndentUntil="cmd D"
cmd A
cmd B
cmd C
cmd D
```

ignoreCase=*string*

Performs a case insensitive comparison with the command on the device.

reportManualConfig=*boolean*

The reportManualConfig attribute lets you explicitly control whether a command can be reported as a manual configuration. By default, global commands are not reported as manual configurations, but nested commands are.

brokenIndentRulesOffset=*integer*

The brokenIndentRulesOffset attribute tells the audit that this command has a bad indent and that it should be corrected to an absolute value. This attribute is used to deal with bad indenting problems and comes into play with when CLIParser will prematurely break out of context.

For example, use brokenIndentRulesOffset="2" for command D in the following list:

```
cmd A
  cmd B
  cmd C
  cmd D
  cmd E
```

In the above example, command D is suppose to be a child of command A but its relative position shows it is out of context for command A. The brokenIndentRulesOffset attribute will come into play when audit interrogates command D and is about break out of command A

context. The absolute position of command D will be reset to ensure that it is seen as a child of command A or peer to commands C and E.

Audit Synonyms

Some device IOS combinations display commands differently from the manner in which the commands were activated on the device. This can cause an audit to fail even though the commands reported are actually logically equivalent to those that were configured.

The cartridge architecture allows you to specify synonyms for commands on the target device. Synonyms use regular expressions to match and replace commands, so it can be determined if two commands are equivalent.

You can specify system synonyms on a per-cartridge basis. The discussion of audit synonyms in *IP Service Activator System Administrator's Guide* describes how audit synonyms work, and describes how an administrator can specify custom synonyms to create new synonyms or redefine system synonyms after IP Service Activator is deployed.

You can specify system synonyms in the **synonyms.xml** file included in the cartridge. The samples and descriptions of elements in a synonyms file, as described in *IP Service Activator System Administrator's Guide*, can be used as reference material for an administrator editing the content of a custom synonyms file or a cartridge developer editing the content of a system synonyms file.

Synonyms can be specified for both the expected command and for the configured command.

> **Note:**
>
> A given command can only have one synonym made up of one or more match and replace criteria. For example, A can be equivalent to A' but not A" simultaneously for equivalence testing within the scope of a single auditTemplate.

For further details on audit synonyms, refer to *IP Service Activator System Administrator's Guide*.

Options

There can be variations in the command syntax for certain commands within device families based on the specific device type and OS variant. Options allow you to specify the correct commands to handle these variants, without having to create a separate service cartridge.

Options allow you to set information in the device model which will render the correct syntax version for a given command.

To do this, you define and document configuration options for a cartridge, and implement the variations in the service model to device model transform, and the annotated device model to CLI transform, based on option values.

Option values for specific device type and IOS combinations are specified in option configuration files, which are registered by cartridge units in the appropriate registry entry file. The option configuration files, and the registry entries that reference them, may be customized by the system administrator once the cartridge is deployed.

For complete details on creating options for a cartridge, refer to *IP Service Activator SDK Base Cartridge Developer Guide*.

Capabilities

In IP Service Activator, when a device is discovered, the policy server fetches the device's capabilities from the Network Processor, so it can model the device appropriately for the user. The Network Processor returns capabilities based on the base and service cartridges that manage that device.

It is the capabilities files that specify the ability of the device and cartridge to support different types of policies at the device, interface and sub-interface level.

Capabilities for different service cartridges are ORed together to provide a single view of the overall device capabilities to the policy server. A fault is generated if different service cartridges specify conflicting capabilities.

It is very important when you are creating cartridges for use with IP Service Activator that the capabilities files be appropriately configured to report the capabilities you are providing policy and service support for.

A sample capabilities file follows:

```
<caps:capabilities
  xmlns:caps="http://www.metasolv.com/serviceactivator/capabilities"
  xmlns="http://www.metasolv.com/serviceactivator/capabilities"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.metasolv.com/serviceactivator/capabilities">
  <caps:device>
  </caps:device>
  <caps:interface>
    <caps:ifType>32</caps:ifType>
    <caps:inbound>
      <caps:access_rule_support>
        <caps:access_rules_supported>255</caps:access_rules_supported>
        <caps:classification_supported>
          <caps:supported>SRC_IP</caps:supported>
          <caps:supported>DST_IP</caps:supported>
          <caps:supported>SRC_PORT</caps:supported>
          <caps:supported>DST_PORT</caps:supported>
          <caps:supported>IP_PROTO</caps:supported>
          <caps:supported>TCP_HEADER_OPTIONS</caps:supported>
          <caps:supported>TCP_ESTABLISHED</caps:supported>
        </caps:classification_supported>
      </caps:access_rule_support>
      <caps:ip_unnumbered_support>0</caps:ip_unnumbered_support>
    </caps:inbound>
    <caps:outbound>
      <caps:access_rule_support>
        <caps:access_rules_supported>255</caps:access_rules_supported>
        <caps:classification_supported>
          <caps:supported>SRC_IP</caps:supported>
          <caps:supported>DST_IP</caps:supported>
          <caps:supported>SRC_PORT</caps:supported>
```

```
            <caps:supported>DST_PORT</caps:supported>
            <caps:supported>IP_PROTO</caps:supported>
            <caps:supported>TCP_HEADER_OPTIONS</caps:supported>
            <caps:supported>TCP_ESTABLISHED</caps:supported>
          </caps:classification_supported>
        </caps:access_rule_support>
        <caps:vce_support>
          <caps:supports_virtual_ce>1</caps:supports_virtual_ce>
          <caps:supports_RIP_virtual_ce>1</caps:supports_RIP_virtual_ce>
          <caps:supports_static_virtual_ce>1</caps:supports_static_virtual_ce>
          <caps:supports_OSPF_virtual_ce>0</caps:supports_OSPF_virtual_ce>
          <caps:supports_eBGP_virtual_ce>1</caps:supports_eBGP_virtual_ce>
          <caps:supports_EIGRP_virtual_ce>1</
caps:supports_EIGRP_virtual_ce>
        </caps:vce_support>
        <caps:ip_unnumbered_support>0</caps:ip_unnumbered_support>
      </caps:outbound>
      <caps:subInterface>
        <caps:inbound>
          <caps:access_rule_support>
            <caps:access_rules_supported>255</caps:access_rules_supported>
            <caps:classification_supported>
              <caps:supported>SRC_IP</caps:supported>
              <caps:supported>DST_IP</caps:supported>
              <caps:supported>SRC_PORT</caps:supported>
              <caps:supported>DST_PORT</caps:supported>
              <caps:supported>IP_PROTO</caps:supported>
              <caps:supported>TCP_HEADER_OPTIONS</caps:supported>
              <caps:supported>TCP_ESTABLISHED</caps:supported>
            </caps:classification_supported>
          </caps:access_rule_support>
          <caps:ip_unnumbered_support>1</caps:ip_unnumbered_support>
        </caps:inbound>
        <caps:outbound>
          <caps:access_rule_support>
            <caps:access_rules_supported>255</caps:access_rules_supported>
            <caps:classification_supported>
              <caps:supported>SRC_IP</caps:supported>
              <caps:supported>DST_IP</caps:supported>
              <caps:supported>SRC_PORT</caps:supported>
              <caps:supported>DST_PORT</caps:supported>
              <caps:supported>IP_PROTO</caps:supported>
              <caps:supported>TCP_HEADER_OPTIONS</caps:supported>
              <caps:supported>TCP_ESTABLISHED</caps:supported>
            </caps:classification_supported>
          </caps:access_rule_support>
          <caps:vce_support>
            <caps:supports_virtual_ce>1</caps:supports_virtual_ce>
            <caps:supports_RIP_virtual_ce>1</caps:supports_RIP_virtual_ce>
            <caps:supports_static_virtual_ce>1</caps:supports_static_virtual_ce>
            <caps:supports_OSPF_virtual_ce>0</caps:supports_OSPF_virtual_ce>
            <caps:supports_eBGP_virtual_ce>1</caps:supports_eBGP_virtual_ce>
            <caps:supports_EIGRP_virtual_ce>1</
caps:supports_EIGRP_virtual_ce>
          </caps:vce_support>
          <caps:ip_unnumbered_support>1</caps:ip_unnumbered_support>
      </caps:outbound>
      </caps:subInterface>
    </caps:interface>
    <caps:interface>
      <caps:ifType>23</caps:ifType>
```

```
      <caps:inbound>
        <caps:access_rule_support>
          <caps:access_rules_supported>255</caps:access_rules_supported>
          <caps:classification_supported>
            <caps:supported>SRC_IP</caps:supported>
            <caps:supported>DST_IP</caps:supported>
            <caps:supported>SRC_PORT</caps:supported>
            <caps:supported>DST_PORT</caps:supported>
            <caps:supported>IP_PROTO</caps:supported>
            <caps:supported>TCP_HEADER_OPTIONS</caps:supported>
            <caps:supported>TCP_ESTABLISHED</caps:supported>
          </caps:classification_supported>
        </caps:access_rule_support>
        <caps:ip_unnumbered_support>1</caps:ip_unnumbered_support>
      </caps:inbound>
      <caps:outbound>
        <caps:access_rule_support>
          <caps:access_rules_supported>255</caps:access_rules_supported>
          <caps:classification_supported>
            <caps:supported>SRC_IP</caps:supported>
            <caps:supported>DST_IP</caps:supported>
            <caps:supported>SRC_PORT</caps:supported>
            <caps:supported>DST_PORT</caps:supported>
            <caps:supported>IP_PROTO</caps:supported>
            <caps:supported>TCP_HEADER_OPTIONS</caps:supported>
            <caps:supported>TCP_ESTABLISHED</caps:supported>
          </caps:classification_supported>
        </caps:access_rule_support>
        <caps:vce_support>
          <caps:supports_virtual_ce>1</caps:supports_virtual_ce>
          <caps:supports_RIP_virtual_ce>1</caps:supports_RIP_virtual_ce>
          <caps:supports_static_virtual_ce>1</caps:supports_static_virtual_ce>
          <caps:supports_OSPF_virtual_ce>0</caps:supports_OSPF_virtual_ce>
          <caps:supports_eBGP_virtual_ce>1</caps:supports_eBGP_virtual_ce>
          <caps:supports_EIGRP_virtual_ce>1</
caps:supports_EIGRP_virtual_ce>
        </caps:vce_support>
        <caps:ip_unnumbered_support>1</caps:ip_unnumbered_support>
      </caps:outbound>
    </caps:interface>
    <caps:interface>
      <caps:ifType>22</caps:ifType>
      <caps:inbound>
        <caps:access_rule_support>
          <caps:access_rules_supported>255</caps:access_rules_supported>
          <caps:classification_supported>
            <caps:supported>SRC_IP</caps:supported>
            <caps:supported>DST_IP</caps:supported>
            <caps:supported>SRC_PORT</caps:supported>
            <caps:supported>DST_PORT</caps:supported>
            <caps:supported>IP_PROTO</caps:supported>
            <caps:supported>TCP_HEADER_OPTIONS</caps:supported>
            <caps:supported>TCP_ESTABLISHED</caps:supported>
          </caps:classification_supported>
        </caps:access_rule_support>
        <caps:ip_unnumbered_support>1</caps:ip_unnumbered_support>
      </caps:inbound>
      <caps:outbound>
        <caps:access_rule_support>
          <caps:access_rules_supported>255</caps:access_rules_supported>
          <caps:classification_supported>
```

```
            <caps:supported>SRC_IP</caps:supported>
            <caps:supported>DST_IP</caps:supported>
            <caps:supported>SRC_PORT</caps:supported>
            <caps:supported>DST_PORT</caps:supported>
            <caps:supported>IP_PROTO</caps:supported>
            <caps:supported>TCP_HEADER_OPTIONS</caps:supported>
            <caps:supported>TCP_ESTABLISHED</caps:supported>
          </caps:classification_supported>
        </caps:access_rule_support>
        <caps:vce_support>
          <caps:supports_virtual_ce>1</caps:supports_virtual_ce>
          <caps:supports_RIP_virtual_ce>1</caps:supports_RIP_virtual_ce>
          <caps:supports_static_virtual_ce>1</caps:supports_static_virtual_ce>
          <caps:supports_OSPF_virtual_ce>0</caps:supports_OSPF_virtual_ce>
          <caps:supports_eBGP_virtual_ce>1</caps:supports_eBGP_virtual_ce>
          <caps:supports_EIGRP_virtual_ce>1</
caps:supports_EIGRP_virtual_ce>
        </caps:vce_support>
        <caps:ip_unnumbered_support>1</caps:ip_unnumbered_support>
      </caps:outbound>
    </caps:interface>
</caps:capabilities>
```

The relevant sections are:

- **Device**: specifies device capabilities
- **Interface**
  - **ifType**: specifies interface capabilities for the interface type instance

By default, capabilities are set to false (i.e. 0) meaning the capability is not supported. Specifying true (i.e. 1) indicates that the service is supported. In some cases, support is specified using strings which indicate which aspect of a capability is supported. For example, the number of access rules supported on an interface of type 32:

```
<caps:access_rules_supported>255</caps:access_rules_supported>
```

In the sample above, there are no device-level capabilities. If a device for which this capabilities file applies was discovered, and had an interface of type 32, then that interface would have support for the capabilities marked true (1).

Message Definition

Success, warning and error message pattern files can be defined for service cartridges in the same way as for base cartridges. For example, device responses for commands sent by a service cartridge are analyzed and patterns are created in the message pattern files for that service cartridge. The difference is that for a base cartridge, the messages files are referenced from the **Registry.xml** file, and for a service cartridge, the messages files are referenced from the **Extension.xml** file.

The Network Processor compares device responses against sets of patterns stored in XML files to monitor its interactions with the device. By finding matches with stored messages patterns, the Network Processor can determine the results of commands sent to the device.

The Network Processor changes existing configuration on the device only if the device response matches a success pattern, a non-blocking warning pattern, or if there is no response at all (i.e. only the command prompt is returned).

As the Network Processor generates the needed commands, connects to the device, and starts changing the device configuration, the responses from the device are analyzed after each command. The following categories of responses are defined:

- **Success**: if the device response matches a success pattern or there is no response at all (i.e. only a prompt is returned) then the command is considered to be successful.
- **Warning** (blocking and non-blocking): if the response from the device matches a non-blocking warning pattern, a fault (i.e. `Warning`) is raised. If the response from the device matches a blocking warning pattern, a fault is raised, and all concretes affected by that transaction are rejected and the partially implemented configuration is rolled back.
- **Error**: if the response from the device matches one of the known error patterns, then a fault (i.e. `Error`) is raised against the device itself, all the concretes affected by that transaction are rejected and the partially implemented configuration is rolled back.

In addition, if the response from the device does not match any success, warning, or error pattern, then the response is considered to be an unknown error. A fault (i.e. `Error`) is raised against the device, all the concretes affected by that transaction are rejected, and the partially implemented configuration is rolled back. The fault message includes the rejected command and the actual message returned by the device. Note that successive white space and new line characters are removed from the device response. The configuration conflict is recorded in the audit log file, and the device state is marked Intervention Required for decision/action by the system administrator.

Overriding Message Definitions

Once the cartridge is deployed, the list of success patterns can be overridden by the system administrator. For this purpose, the cartridge .zip file includes a copy of the **successMessages.xml** file in the vendor specific sampleRegistry directory. For example, for a Cisco cartridge, the path would be *Service_Activator_home***Config\networkProcessor\ciscoSampleRegistry\messages\successMessages.xml**.

To override the success patterns, a system administrator can edit a copy of the **successMessage.xml** file and copy it to the same relative location referenced by the appropriate registry entry. If no **successMessages.xml** file is specified in the registry file entry, the global **SuccessMessages.xml** file will be used.

> **Note:**
>
> The global **SuccessMessages.xml** file is deprecated.

For example, to override the Cisco **successMessages.xml** file once the cartridge has been deployed:

1. Copy the **successMessages.xml** file found in the install directory at

   *Service_Activator_home***Config\networkProcessor\ciscoSampleRegistry\messages\successMessages.xml**

   to

   *Service_Activator_home***Config\networkProcessor\com\metasolv\serviceactivator\cartridges\cisco\successMessages.xml**

2. Add, modify, or delete success patterns as necessary.
3. Ensure that the location specified in the appropriate registry file reflects the actual location of the file relative to the install directory at *Service_Activator_home***Config\networkProcessor**

   For example:

   ```
   <successMessages>com\metasolv\serviceactivator\cartridges\cisco\messages\succ
   essMessages.xml</successMessages>
   ```

The same process can be applied for warning and error message files.

For details on message files as part of a base cartridge, refer to *IP Service Activator SDK Base Cartridge Developer Guide*.

Pre- and Post-Checks

Pre- and post-checks provide the ability verify information on a device when the annotated DM to CLI transform executes, before the general configuration is sent. This allows you to confirm that prerequisites to the configuration are met. For example, some configuration may rely on certain other values being configured on the device. Alternatively, some configuration may conflict with certain values that may be configured on the device.

After configuration is sent, you have the opportunity to have a post-check invoked to verify some aspect of the commands that were sent to the device.

Types of Pre- and Post-Checks

There are two basic types of checks: Type I and Type II.

For Type I checks, the framework handles the pass/fail decision, based on regular expression pattern matching of the result against the message files.

With Type II checks, the result of sending the check commands are passed back, and then programmatically inspected using a customized XQuery. Appropriate action is taken based on that result.

Type I Checks

In a Type I check, the check mechanism sends commands to the device, and then parses the results against the cartridges message files. If a match occurs, an appropriate action is taken to continue or abort the remaining configuration.

Type I check is imbedded in the CLI document executed by the Network Processor to send the CLI to the device.

Type II Checks

A Type II check is executed before the CLI document is executed by the Network Processor. The check mechanism sends commands to the device, and returns the result. This result is programmatically inspected based on your customized XQuery to determine whether to continue or abort the remaining configuration.

There are two types of Type II checks:

- **Type IIa checks**: uses a success/error pattern. The pattern match determines how the pass/fail condition is met.
- **Type IIb checks**: the command instance has no success/error pattern. The results are inspected using your customized XQuery to determine what action to take.

Cartridge Version

The cartridge version is used as the version of the device model that the cartridge generates. It is set in the cartridge skeleton properties file (**sdk_global_cartridgeVersion**) and the value of it is put into the **dm-version.xq** file when it is generated. Any device model that is generated by the cartridge is given this version number.

A sample fragment from **dm-version.xq** follows: ""

```
declare variable $dmver:version := "2.0";
```

The version format can be a, a.b, a.b.c (major, major.minor, major.minor.sub-minor), where a, b, c must all be numeric.

As a cartridge developer, you want to increment the cartridge version whenever a significant change to the device model has occurred which requires a non-trivial upgrade to an existing DM.

Device Model Upgrades

If you make modifications which include device model schema changes to a cartridge that is already deployed, you must be aware of the effect of these changes on your IP Service Activator installation, in particular, persistent data.

As the cartridge developer, you are responsible for analyzing whether or not a device model upgrade is required, and if so:

- Updating the cartridge version
- Updating and unit testing the upgrade source code related to the modified device model

Once the cartridge is deployed, non-trivial changes to **devicemodel.xsd** will require a device model upgrade and may or may not require a custom transform for the upgrade.

Examples of non-trivial device model changes are:

- Introduction of mandatory attributes
- Removal of attributes
- Substantial modification of an existing attribute
- Re-ordering of attributes

An example of a trivial device model change is:

- Introduction of optional attributes

The device model transform is required if the change made to the schema impacts the use of the last device model during the service model to device model transform. By default, the last device model is not used in the generated cartridge source files. It would only be used by custom XQuery code. If it is not used, then the upgrade will regenerate the new version of the last device model without the need for a custom upgrade transform.

Should you make a non-trivial change to a device model, then you must write a transform that allows the Network Processor upgrade tool (NpUpgrade) to upgrade the existing device model.

Network Processor NpUpgrade

The NpUpgrade does a partial upgrade of all device models, including those for the service cartridges. It looks for the **xquerylib\DmUpgrade.xq** in the same classpath as **sm2dm.xq**, and if this file exists, it uses the XQuery to perform the upgrade. The upgraded device model is used as the last device model in the upgrade process.

- An editable **DmUpgrade.xq** is added to the xquerylib directory for each base and service cartridge upon source file generation.
- For each cartridge, an *SDK_home***\...\test\models\upgradeFrom\sampleDeviceModel.xml** file is provided for testing purposes.
- A test file, *SDK_home***\...\test\DmUpgradeTests.java**, is provided to transform the sample device model by running the **DmUpgrade.xq** xquery.

Configuration Version

The IpsaConfigVersion is removed from a device when the last concrete is removed from the device in IP Service Activator. Typically, this coincides with the removal of the last IP Service Activator configured command from the device.

An exception to this occurs when a configuration policy, which does not generate commands, is the last configuration to be removed from the device in IP Service Activator. For example, schedule is an example of a configuration policy that does not generate commands. If IP Service Activator is running with Configuration Management, and you have installed a schedule on a device, along with other concretes which do generate commands, the device will maintain an IpsaConfigVersion, until all configuration is removed including the schedule.

Configuration Management Support

The SDK supports the configuration of cartridges to enable the use of services from the Configuration Management product.

The following services provided by Configuration Management can be supported in the Network Processor:

- **Audit**: supported through base and service cartridges
- **Configuration activation**: supported from configuration policies through service cartridges in conjunction with a base cartridge.
- **Restore**: can be implemented in base cartridges

For More Information on Cartridge Operations

For extensive information on the following points, see *IP Service Activator System Administrator's Guide*:

- Audit types and reports

- The syntax of audit template file entries
- Running a device audit
- Contents of synonyms files
- Tools to help create synonyms

For information on configuring cartridges to support the use of services from the Configuration Management product, see *IP Service Activator SDK Configuration Management Developer Guide*.

5

Testing, Monitoring, and Error Handling

This chapter explains how to perform unit and end-to-end tests, logging, and error handling.

Test Environments

Basic tests of cartridges and configuration policies created with the SDK can be performed without an Oracle Communications IP Service Activator or Oracle Communications Configuration Management system.

To perform end-to-end tests of installed cartridges and configuration policies that you have developed, it is recommended that you have a IP Service Activator installation dedicated to the purpose. This system should have access to some actual devices.

Testing on a live IP Service Activator installation is not recommended.

Unit Test

A unit test script is automatically created when you compile a cartridge. The purpose of the unit test is to verify that the main transform stages of the cartridge (service model to device model and annotated device model to CLI) generate the output documents correctly.

DM Validation

DM validation is used to validate the device model generated by the cartridge. It is called after the cartridge has generated the device model from the service model. The validation is more specific to the vendor and service.

Logging

Each Network Processor maintains one current log file and one current audit trail log file. The Network Processor logging facilities are based on the log4j utility.

For more information on network processor logging, please refer to the discussion of Network Processor administration and maintenance in *IP Service Activator System Administrator's Guide*.

Audit Trail Logging

Audit trail logging records the commands sent to devices by the base cartridge, and any service cartridges that extend the services of the base cartridge.

The audit trail log files for the Network Processor are located in the directory: *Service_Activator_home***AuditTrails**.

Audit trail logging properties are set up on a per-cartridge basis and include:

- The name of the audit logging file

- Audit trail logging level
- File rollover strategy

Handling Faults and Errors

The Network Processor has a number of strategies for handling faults and errors when communication with the target device.

Rollback

Any time an error is encountered while sending commands to a device, a rollback will be initiated. The rollback's purpose is to restore the device to the same state it was in before configuration changes were applied. This ensures the device is returned to a known state.

The rollback works by reversing the last and target device models before annotating. This will cause configuration that was created to instead be marked as deleted. Likewise, anything removed would instead be added. Also any modifications would be modified back to their original state.

Once this reverse annotation is completed, the DM to CLI is rerun. This result is a complete list of commands to undo the changes.

Device Model IDs

The remaining component to the rollback involves the device model IDs. DM IDs are used to correlate commands back to their corresponding device model elements. They must be implemented correctly for rollbacks to function properly. Incorrectly implemented DM IDs will not be detected or noticed during the successful running of the system. They will only cause problems after specific failures sending commands to a device.

DM IDs are used to track changes applied to the device, and provide a means for knowing which changes have been successfully applied before the error was encountered. These IDs remain the same for the reverse annotation, and are used to filter out configuration that was not applied. This is done as part of the annotations. Any configuration that did not get applied to the device will not be marked as changed.

Every device model element extending the base type of Changeable will be marked with a different DM ID by the framework before the DM to CLI transform. Every element marked as changed (changetype="ADD", “DELETE" or “MODIFY") will need to have its DM ID in a command to be sent to the device. The framework can then determine which configuration elements have been successfully applied to the device while sending commands. This is needed when a failure occurs, as the framework will redo the annotations but will exclude any configuration that was not successfully applied.

Rollback Failures

If any failure occurs during the rollback, the system is forced to put the device in the intervention required state. This is an indication that the system could not put the device into a known state and a user must investigate.

For more information on how to deal with a failed rollback, refer to the discussion of recovering from rollback failure in *IP Service Activator System Administrator's Guide*.

Quarantine

The rollback mechanism will remove all configuration changes if any failure is encountered while sending the commands to the device. There could be many different changes in a given transaction, not all of which could fail. Quarantine provides a means of separating the failed configuration so other changes can be applied to the device.

If any failure occurs where the association IDs are known (e.g. sending commands, DM validation XQuery faults), the system will quarantine that configuration after the rollback has been completed. This is done by restoring the failed parts of the service model to the last service model. Once this has been completed, the transforms are run again with the new service model. This effectively reverts the failed configuration and then retries any other configuration changes that could have occurred.

6
Best Practices

This chapter offers some best practices to help you create high quality cartridges.

Choosing Whether to Extend a Cartridge or to Plan a New Cartridge

If you are considering adding support for a new service type using the SDK, you can either add that to one of your pre-existing service cartridges, or you can choose to create a new service cartridge.

In general, the guidelines are as follows:

- Services that have inter-dependencies (other than interface/sub-interface creation) must have their implementations co-exist in the same cartridge.
- Service types that will have many instances on a device (for instance, VPLS service compared to SNMP setup service) must exist in their own cartridge.
- Go with the minimal set of cartridges to implement the desired set of services within the above constraints (inter-dependencies, model size, choice of Java vs. XQuery). This is because there is overhead when you add each service cartridge. The actual overhead will depend on the set of cartridges and the set of services on a given device.

Choosing the Cartridge Implementation

You can choose to create either Java or XQuery-based cartridges. This section provides some information to help you choose.

Developer Knowledge

Java knowledge (and experience with additional libraries such as XMLBeans, SAX, DOM) is more commonly available than XQuery knowledge.

Transformation Complexity

XQuery is very good at mining data from the source document and putting it into the destination document. However, it does not excel at complex flow logic, modularization, or computation. If you require any of these, you have two options:

- Write your basic flow in XQuery, as well as any simple transforms, and write the more complex parts in Java. Call the Java methods from XQuery.

  or

- Write everything in Java.

Model Size

The size of the Service Model and Device Model depend on a number of factors:

- The number of instances of services of this kind that will go on one device.
- The size of an individual instance (for example, some PHB instances can get very large).
- The number of commands generated for an instance.

Typically, the larger the model, the more Java has to figure in the transformation, either through an XQuery framework seeded with Java callouts, or by using only Java.

Time to Complete

With knowledgeable developers, XQuery typically delivers working cartridges earlier than Java. Prototyping is typically easier with XQuery.

XQuery Advantages and Disadvantages

The advantages of using XQuery are:

- Faster development cycles (at least, initially)
- Very rich language for the functionality it covers
- Easily field modifiable

The disadvantages to using XQuery are:

- Very primitive modularization concepts (and object orientation concepts)
- No schema awareness, limited type checking
- Control flow structures are difficult to use, especially beyond simple flow control
- Runtime exceptions come from Saxon and are difficult to debug
- Additional IDE for full developer convenience required (e.g. oXygen, XMLspy)
- Very easy to write sub-optimal code and XPath expressions (both in terms of memory use and execution time)
- Code flow must mimic the sequence and structure of the output XML document which leads to convoluted (spaghetti) code without a lot of constant effort

Java Advantages and Disadvantages

The advantages of using Java are:

- Full fledged programming language
- Multiple ways to deal with XML data
- Easier to layout responsibilities for behavior and data abstraction
- Can be written much faster
- Standard Java development, debugging, and profiling tools are enough

- Allows developer to greatly optimize the generation of the Device Model from the Service Model by viewing it as a set of modifications to be performed on (a copy of) the Last Device Model, rather than as a complete re-generation of the new Device Model

XQuery Transform Best Practices

This section offers best practices to create high quality XQuery transforms for cartridges.

XQuery Performance Optimization

XQuery is not a procedural language, so care must be taken to not write code in a procedural fashion. The XQuery engines implement performance optimizations that can often cause side-effects when attempting to write procedural code with XQuery.

Consider the code below:

```
declare function myFunction($source as element()*) as xs:boolean
  {
  let $list := for $item in $source
    return
    $item/itemvalue
  return
    if (fn:count($list) > 0) then
      true()
    else
      false()
  ;
```

This code returns the correct Boolean value, but the inner loop may not execute on all possible items in the source. Optimizations may cause the loop to terminate once the following ‘if' expression is fully satisfied (i.e. there is something in the list).

This is just one example of how the optimizations take place. Many more can occur and most are driven by determining dependencies and the engine only evaluating things that need to be evaluated.

Another optimization example follows:

```
declare function myFunction2($source as element()*) as xs:boolean
  {
  let $list := myFunctionA ($source/element)
  let $list2 := myFunctionB ($source/element2)
  return
    $list
```

This can evaluate **$list** by calling the function **myFunctionA**, but the return value specifies only **$list** and not **$list2** so **myFunctionB** may never be called.

XQuery Searches

Care must be taken while creating XQuery searches. XQuery always checks for all possible search combinations. In some cases, the application designer is aware that there is only a single match for something for a search item, but XQuery does not provide an accurate representation of it. Instead, it evaluates all the possible combinations. The easiest way to reduce the performance cost of a search is to be careful about the scope of all searches.

For more efficient searches, avoid searching with “//" as this causes two problems:

- Performance impact
- Can unexpectedly return results at different levels in the document. It is always better to be more explicit about scope.

Best Practices for Coding XQuery

The following points define the best practices while writing codes in XQuery:

- Care should be taken while writing loops, especially nested loops. Nested looping, even when constrained with a ‘where' clause, can be very risky. The behavior is very similar to general searching where all the possible combinations are evaluated.
- It is advisable to break code into separate modules, and use smaller functions wherever possible.
- Some functions are not always appropriate for implementing in XQuery. Very complex methods are more appropriate to do in Java. For example, cases where data can be held and stored in Maps instead of continually searching, are more appropriately done in Java.

Syntax For Entering Control Characters in XQuery

You can enter any characters from the ASCII character set (0-255) in any text editor using the following procedure:

1. On your keyboard, press the Num Lock key so that Num Lock is turned on
2. Press and hold the Alt key
3. Using the keypad on your keyboard, enter the decimal code of the character as a 3 digit number, padded on the left with zeros, if necessary (for example, 009 for tab)
4. Release the Alt key

Alternatively, many text editors allow you to select control characters from a list for insertion. For example, in TextPad, use the command **View->Clip Library**.

In addition to entering ASCII characters on Windows editors like Notepad and Wordpad, there are ways to enter these characters in UNIX editors like vi and emacs.

For example:

- Ctrl + I = horizontal TAB (numerical decimal value = 9)
- Ctrl + J = Line Feed (10)
- Ctrl + M = carriage return (13)

Java Transform Best Practices

his section provides some best practices for creating quality Java transforms.

Java Searches

Similar to the best practices for XQuery, the same care should be taken when using XPath statements to search for objects. This is most important when constructing the device model. Since you are creating the device model using Java, unlike with XQuery, you have random access to any part of the device model that you have already created. To improve search performance, use XmlCursor or even better, XmlBookmarks as you are creating the device model. The bookmarks allows you to quickly go back to a part of the device model you have created without using XPath to search for it.

Service Model to Device Model Java Transform

When implementing an Sm2Dm transform in Java, you will need to create a class that extends the ModelTransformer class. Then you will need to provide two methods. The first is **transform()**, which the Network Processor will look for and call to perform the service model to device model transform.

The second method you must supply is **getAppInfo()**. This method is used to get the version of your device model. This value is used to determine if any upgrade procedures need to be done to your device model between releases of your cartridge.

An example of a skeleton Sm2Dm class follows:

```
public class Sm2dm extends ModelTransformer {
  public XmlObject transform(
    XmlObject doc,
    Map params,
    Map uris) throws Exception {

    // get the last DM
      com.metasolv.serviceactivator.devicemodel.DeviceDocument lastDm;
      try {
        lastDm = (com.metasolv.serviceactivator.devicemodel.DeviceDocument)
((com.metasolv.serviceactivator.devicemodel.DeviceDocument)uris.get("last_dm")).copy();
      } catch (Exception e) {
      lastDm = null;
    }

    com.metasolv.serviceactivator.servicemodel.DeviceDocument sm;
    sm = (com.metasolv.serviceactivator.servicemodel.DeviceDocument)doc.copy();

    com.metasolv.serviceactivator.devicemodel.DeviceDocument targetDm;

    // Your transform here.

    return targetDm;
  }

  public XmlObject getAppInfo() {

    AppInfo appInfo = AppInfo.Factory.newInstance();
    appInfo.setVersion("1.0.0");
    return appInfo;
  }
}
```

Annotated Device Model to CLI Document Java Transform

When implementing a Dm2Cli transform in Java, you will need to create a class that extends the ModelTransformer class. Then, you will need to provide a method called **transform()** that the network processor will look for and call to perform the device mAn example of a skeleton Dm2Cli class is shown below:

An example of a skeleton Dm2Cli class is shown below:

```
public class AnnotatedDm2Cli extends ModelTransformer {
  public XmlObject transform(XmlObject doc, Map params, Map uris) throws
Exception {
    com.metasolv.serviceactivator.devicemodel.DeviceDocument lastDm;
    try {
      lastDm =
(com.metasolv.serviceactivator.devicemodel.DeviceDocument)uris.get("last_dm");
    } catch (Exception e) {
    lastDm = null;
  }

  XmlObject annotatedDm = doc.copy();

  CommandSessionDocument commandSessionDoc;

  // Your transform here.

  return commandSessionDoc;
}odel to CLI transform.
```

Best Practices for Extending the Device Model

Some best practices for creating device model extension .xsd files are:

- Define one element per command and one subordinate element for each command parameter. For example, keyword param1 param2 below:

  ```
  <keyword>
      <param1>value_1</param1>
      <param2>value_2</param2>
  </keyword>
  ```

- When defining the type for an element that maps to a command, extend **lib:Changeable**. This will allow **annotate()** (the DM compare processor) to mark it `@changeType="ADD"|"DELETE"`.
- Make an element Changeable and Identifiable if the command supports modifications (as opposed to delete and re-add). This will allow **annotate()** to mark it `@changeType="ADD"|"DELETE"|"MODIFY"`. An element is Identifiable if it belongs to a **lib:Container** and some of its subordinated elements make up a key within the scope of the container.
- Make an element Identifiable but not Changeable if it is only used as context for configuration items managed by Oracle Communications IP Service Activator (such as interfaces).
- Use ***Ref** in the name of an element that references other elements.
- Use XML schema validation as much as possible. For example, use min/maxOccurs, min/maxInclusive, pattern.

- Do not define unnecessary elements or types. Use containers only for defining scope for Identifiable elements.
- Do not use any unnecessary abstraction in the DM. Instead of generating multiple command types from a single DM element type, define one element type for each CLI command that needs to be generated. The SM2DM transformation should take care of the abstract-concrete mapping.